ORACLE

Java™

Java

Przewodnik dla początkujących

Wydanie VII

Herbert Schildt

Helion

Oracle Press™

Tytuł oryginału: Java: A Beginner's Guide, Seventh Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-4611-6

HELION SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javpp7.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javpp7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ

17

Wprowadzenie do JavaFX

W tym rozdziale poznasz:

- podstawowe pojęcia JavaFX, takie jak obszar roboczy, scena, węzeł oraz graf sceny,
- metody cyklu życia aplikacji JavaFX,
- ogólną postać aplikacji JavaFX,
- sposób uruchamiania aplikacji JavaFX,
- tworzenie etykiet,
- stosowanie przycisków,
- obsługę zdarzeń,
- stosowanie pól wyboru,
- stosowanie list,
- tworzenie pól tekstowych,
- sposoby dodawania efektów,
- sposoby stosowania transformacji.

W szybkim świecie komputerów jedynie zmiany są czymś stałym, a sztuka i nauka programowania bezustannie się rozwijają i idą naprzód. Nic zatem dziwnego, że pakiety do tworzenia w języku Java aplikacji o graficznym interfejsie użytkownika także biorą udział w tym procesie. Przypomnijmy sobie, że początkowym pakietem do tworzenia aplikacji o graficznym interfejsie użytkownika (w skrócie aplikacji GUI) w Javie był pakiet AWT. Niedługo po wprowadzeniu został on zastąpiony pakietem Swing, który udostępnił znacznie lepsze rozwiązania. Choć pakiet Swing odniósł ogromny sukces, to jednak dosyć trudno w nim tworzyć „wizualne cuda", których niejednokrotnie wymagają dzisiejsze aplikacje. Co więcej, rozwinęły się także koncepcje leżące u podstaw pakietów do tworzenia graficznego interfejsu użytkownika. Aby lepiej obsługiwać wymogi nowoczesnych interfejsów użytkownika oraz odzwierciedlić postęp w sposobach ich projektowania, konieczne było wprowadzenie nowego rozwiązania. W efekcie powstał system JavaFX — platforma do tworzenia graficznych interfejsów użytkownika nowej generacji. Niniejszy rozdział stanowi wprowadzenie prezentujące możliwości i sposoby stosowania tego systemu.

Koniecznie należy wspomnieć, że prace nad JavaFX były realizowane w dwóch fazach. Początkowa wersja JavaFX bazowała na języku skryptowym o nazwie **JavaFX Script**. Niestety, prace nad nim zostały zarzucone. Zaczynając od wersji JavaFX 2.0, platformy tej używa się bezpośrednio w kodzie Javy i udostępnia ona rozbudowany i wyczerpujący interfejs programowania aplikacji. JavaFX dostarczana jest wraz z JDK 7, zaczynając od jego wersji 4. Najnowszą wersją tej platformy jest JavaFX 9,

która stanowi jeden z elementów JDK 9. Ponieważ w momencie pisania tej książki JavaFX 9 była najnowszą wersją platformy, dlatego właśnie ona została opisana w tym rozdziale.

Zanim zaczniemy poznawać możliwości JavaFX, przyda się odpowiedzieć na jedno pytanie, które w naturalny sposób pojawia się w jego kontekście: Czy platforma JavaFX powstała jako zamiennik pakietu Swing? Odpowiadając na nie, należy stwierdzić, że w zasadzie tak właśnie było. Niemniej jednak pakiet Swing jeszcze przez jakiś czas pozostanie używanym elementem Javy. Wynika to z faktu, że wciąż istnieje bardzo dużo kodu, który z niego korzysta. Co więcej, istnieją całe legiony programistów, którzy wiedzą, jak go używać. Pomimo to nie ulega wątpliwości, że to właśnie JavaFX jest platformą, która w przyszłości będzie preferowanym narzędziem do tworzenia graficznych interfejsów użytkownika aplikacji pisanych w Javie. Oczekuje się, że w ciągu kilku najbliższych lat JavaFX zastąpi Swing w nowych projektach oraz że wiele aplikacji, które do tej pory używały pakietu Swing, zostanie zmodyfikowanych i zacznie używać biblioteki JavaFX. Najprościej rzecz ujmując: JavaFX to coś, czego żaden programista Javy nie może zignorować.


Uwaga

W tym rozdziale zakładam, że dysponujesz podstawową znajomością zagadnień związanych z tworzeniem graficznych interfejsów użytkownika w języku Java, w tym także związanych z obsługą zdarzeń, przedstawioną w rozdziale 16.

Podstawowe pojęcia JavaFX

Zanim będziesz mógł stworzyć swoją pierwszą aplikację JavaFX, musisz poznać i zrozumieć kilka kluczowych pojęć i możliwości. Choć JavaFX jest pod pewnymi względami podobna do pakietu Swing (który przedstawiłem w poprzednim rozdziale), to jednak występują pomiędzy nimi pewne znaczące różnice. Na przykład komponenty JavaFX, podobnie jak komponenty Swing, są w całości napisane w Javie, a zdarzenia są obsługiwane w przejrzysty i prosty sposób. Niemniej jednak ogólny sposób organizacji JavaFX oraz powiązania pomiędzy jej głównymi komponentami znacząco się różnią od tych w pakiecie Swing. Dlatego też sugeruję, byś uważnie przeczytał kolejne punkty rozdziału.

Pakiety JavaFX

Cała zawartość platformy JavaFX została umieszczona w pakietach, których nazwy zaczynają się od `javafx`. Gdy pisałem tę książkę, w skład całej biblioteki API JavaFX wchodziło ponad 30 różnych pakietów. Oto cztery przykładowe z tych pakietów: `javafx.application`, `javafx.stage`, `javafx.scene` oraz `javafx.scene.layout`. Choć w tym rozdziale będziemy używać jedynie kilku pakietów JavaFX, to jednak warto poświęcić trochę czasu na dokładniejsze ich poznanie. JavaFX oferuje bowiem bardzo szeroki zakres możliwości. Zaczynając od JDK 9, pakiety JavaFX zostały zorganizowane w formie modułów, takich jak `javafx.base`, `javafx.graphics` oraz `javafx.controls`.

Klasy Stage oraz Scene

Główną metaforą zaimplementowaną w JavaFX jest **obszar roboczy** (ang. *stage*). W przypadku prawdziwych sztuk scenicznych obszar roboczy zawiera **scenę** (ang. *scene*). A zatem potocznie mówiąc: obszar roboczy definiuje przestrzeń, a scena określa, co się w tej przestrzeni pojawi. Albo, wyrażając to samo w nieco inny sposób, obszar roboczy jest pojemnikiem dla sceny, a scena — dla elementów, które się na nią składają. W efekcie wszystkie aplikacje JavaFX zawierają przynajmniej jeden obszar roboczy oraz przynajmniej jedną scenę. Elementy te są reprezentowane w JavaFX API odpowiednio przez klasy `Stage` oraz `Scene`. Przyjrzyjmy się nim nieco dokładniej.

Obiekt `Stage` jest pojemnikiem najwyższego poziomu. Wszystkie aplikacje JavaFX automatycznie mają dostęp do jednego obiektu `Stage`, nazywanego **głównym obszarem roboczym** (ang. *primary stage*). Ten główny obszar roboczy jest dostarczany przez system uruchomieniowy podczas uruchamiania aplikacji JavaFX. Choć w wielu aplikacjach można tworzyć własne obszary robocze, to jednak ten główny będzie jedynym, który jest niezbędny.

Jak już wspominałem, obiekt `Scene` jest pojemnikiem dla wszystkich elementów, które składają się na scenę. Są nimi kontrolki, takie jak przyciski, pola wyboru, pola tekstowe oraz inne elementy graficzne. W celu utworzenia sceny dodasz te wszystkie elementy do obiektu klasy `Scene`.

Węzły i graf sceny

Pojedyncze elementy sceny są nazywane **węzłami** (ang. *node*). Na przykład takim węzłem będzie kontrolka przycisku. Jednak węzły mogą także zawierać grupy innych węzłów. Co więcej, węzeł może także zawierać inny węzeł potomny. W takim przypadku węzeł zawierający jakieś dziecko jest nazywany **węzłem rodzica** (ang. *parent node*) lub **węzłem gałęzi** (ang. *branch mode*). Z kolei węzły, które nie zawierają węzłów potomnych, są nazywane **węzłami końcowymi** (ang. *terminal node*) lub **liśćmi**. Kolekcja wszystkich węzłów tworzących scenę jest określana jako **graf sceny** (ang. *scene graph*) i stanowi **drzewo**.

W grafie sceny występuje jeden, specjalny typ węzła — **korzeń** (ang. *root node*). Jest to najwyższy węzeł w grafie sceny i jednocześnie jedyny, który nie ma żadnego rodzica. A zatem wszystkie inne węzły należące do grafu sceny mają rodzica i wszystkie bezpośrednio lub pośrednio są potomkami korzenia.

Klasą bazową dla wszystkich węzłów jest `Node`. Istnieje kilka różnych klas, które bezpośrednio lub pośrednio są jej klasami pochodnymi; między innymi należą do nich: `Parent`, `Group`, `Region` oraz `Control`.

Układy

JavaFX udostępnia kilka paneli układu obsługujących proces rozmieszczania innych elementów na scenie. Na przykład klasa `FlowPane` tworzy układ rozmieszczający elementy jeden za drugim, a klasa `GridPane` — układ pozwalający na umieszczanie elementów w wierszach i kolumnach. Dostępnych jest także kilka innych układów, takich jak `BorderPane` (przypominający nieco układ `BorderLayout` pakietu AWT). Każdy z nich dziedziczy po klasie `Node`. Wszystkie klasy układów zostały umieszczone w pakiecie `javafx.scene.layout`.

Klasa Application oraz metody cyklu życia

Aplikacja JavaFX musi być klasą pochodną klasy `Application` zdefiniowanej w pakiecie `javafx.application`. Oznacza to, że także Twoja aplikacja będzie rozszerzać klasę `Application`. Klasa ta definiuje trzy metody cyklu życia, które aplikacja może przesłaniać. Noszą one odpowiednio nazwy: `init()`, `start()` oraz `stop()`. Poniżej przedstawiłem postać tych metod w kolejności, w jakiej są one wywoływane:

```
void init()

abstract void start(Stage scenaGlowna)

void stop()
```

Metoda `init()` jest wywoływana na samym początku działania aplikacji. Służy ona do wykonywania różnych czynności inicjalizacyjnych. Niemniej jednak, jak wyjaśnię w dalszej części rozdziału, *nie można* jej używać do tworzenia obiektu sceny ani do jej konstruowania. Jeśli aplikacja nie potrzebuje żadnych czynności inicjalizacyjnych, to metody `init()` nie trzeba przesłaniać, gdyż istnieje jej wersja domyślna.

Po metodzie `init()` wywoływana jest metoda `start()`. To właśnie ona stanowi początek działania aplikacji i *może* posłużyć do skonstruowania i przygotowania sceny. Zwróć uwagę, że parametrem tej metody jest obiekt klasy `Scene`. Jest to obiekt sceny dostarczony przez środowisko uruchomieniowe i stanowi scenę główną aplikacji. Zauważ także, że jest to metoda abstrakcyjna — czyli aplikacja musi ją przesłonić.

Kiedy aplikacja kończy działanie, wywoływana jest z kolei metoda `stop()`. Pozwala ona na wykonanie wszelkich czynności porządkowych związanych z zamykaniem aplikacji. W przypadkach, gdy wykonywanie takich czynności nie jest potrzebne, można skorzystać z pustej, domyślnej wersji tej metody.

Uruchamianie aplikacji JavaFX

Aby uruchomić niezależną aplikację JavaFX, należy wywołać metodę `launch()` klasy `Application`. Istnieją dwie wersje tej metody; poniżej przedstawiłem tę, której będziemy używali w tym rozdziale:

```
public static void launch(String ... args)
```

Parametr *args* reprezentuje listę (być może pustą) łańcuchów znakowych, które zazwyczaj będą określały argumenty wiersza poleceń. Wywołanie metody `launch()` powoduje utworzenie aplikacji, a następnie wywołanie jej metod `init()` i `start()`. Wywołanie metody `launch()` zakończy się dopiero po zakończeniu aplikacji. Ta wersja metody `launch()` uruchamia klasę pochodną klasy `Application`, w której kodzie metoda ta została wywołana. Druga wersja metody `launch()` pozwala wskazać klasę pochodną klasy `Application`, którą należy uruchomić.

Zanim przejdziemy dalej, koniecznie trzeba zwrócić uwagę na jeszcze jedno ważne zagadnienie: aplikacje JavaFX, które zostały spakowane przy użyciu narzędzia *javafxpackager* (bądź też przy wykorzystaniu analogicznych narzędzi zintegrowanych środowisk programistycznych), nie muszą zawierać wywołania metody `launch()`. Niemniej jednak skorzystanie z niej niejednokrotnie upraszcza proces testowania i uruchamiania aplikacji, a oprócz tego pozwalają one na używanie programu bez konieczności tworzenia pliku JAR. Z tego powodu w przykładach przedstawionych w tym rozdziale metoda ta będzie używana.

Szkielet aplikacji JavaFX

Wszystkie aplikacje JavaFX posiadają tę samą, podstawową strukturę. Dlatego zanim poznasz jakiekolwiek inne możliwości platformy JavaFX, warto zobaczyć, jak wygląda szkielet takiej aplikacji. Oprócz przedstawienia struktury aplikacji przykład z listingu 17.1 pokazuje także, jak taką aplikację można uruchomić oraz kiedy wywoływane są poszczególne metody cyklu jej życia. Komunikaty generowane przez poszczególne metody będą wyświetlane w oknie konsoli. Poniżej przedstawiony został pełny szkielet aplikacji JavaFX.

Listing 17.1. *JavaFXSkel.java*

```
// Szkielet aplikacji JavaFX.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

  public static void main(String[] args) {

    System.out.println("Uruchamiamy aplikację JavaFX.");

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody init().
  public void init() {
    System.out.println("Wewnątrz metody init().");
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    System.out.println("Wewnątrz metody start().");

    // Określa tytuł sceny.
```

```
    myStage.setTitle("Szkielet aplikacji JavaFX.");

    // Tworzy korzeń. W tym przypadku będzie to układ
    // FlowPane, choć istnieje także kilka innych.
    FlowPane rootNode = new FlowPane();   <-- Utworzenie korzenia.

    // Tworzy scenę.
    Scene myScene = new Scene(rootNode, 300, 200);   <-- Utworzenie sceny.

    // Określa scenę używaną przez obiekt obszaru roboczego.
    myStage.setScene(myScene);   <-- Ustawienie sceny w obiekcie obszaru roboczego.

    // Wyświetla obszar roboczy oraz scenę.
    myStage.show();   <-- Wyświetlenie obszaru roboczego.
  }

  // Przesłania metodę stop().
  public void stop() {
    System.out.println("Wewnątrz metody stop().");
  }
}
```

Choć ten szkielet aplikacji jest całkiem prosty, można go skompilować i uruchomić. Spowoduje to wyświetlenie na ekranie pustego okna. Oprócz tego aplikacja wyświetli także w oknie konsoli następujące komunikaty:

```
Uruchamiamy aplikację JavaFX.
Wewnątrz metody init().
Wewnątrz metody start().
```

Kiedy zamkniesz okno aplikacji, w oknie konsoli zostanie wyświetlony kolejny komunikat:

```
Wewnątrz metody stop().
```

Oczywiście w rzeczywistych aplikacjach metody cyklu życia nie wyświetlają żadnych komunikatów w standardowym strumieniu wyjściowym `System.out`. W tym przykładzie komunikaty są wyświetlane tylko po to, by pokazać, kiedy poszczególne metody są wywoływane. Co więcej, zgodnie z podanymi wcześniej informacjami przesłanianie metod `init()` oraz `stop()` jest konieczne wyłącznie w przypadkach, gdy aplikacja musi wykonywać jakieś niestandardowe czynności związane z jej inicjalizacją lub zamykaniem. W pozostałych przypadkach z powodzeniem można użyć domyślnych implementacji tych metod, dostarczanych przez klasę `Application`.

A teraz przeanalizujmy ten program bardziej szczegółowo. Zaczyna się on od zaimportowania czterech pakietów. Pierwszym z nich jest `javafx.application`, który zawiera klasę `Application`. Klasa `Scene` należy do pakietu `javafx.scene`, a klasa `Stage` do pakietu `javafx.stage`. Ostatni z pakietów, `javafx.scene.layout`, zawiera kilka paneli układów. W powyższym przykładzie używany jest układ `FlowPane`.

Następnie tworzona jest klasa `JavaFXSkel`. Zwróć uwagę, że rozszerza ona klasę `Application`. Jak już wyjaśniałem, `Application` to klasa, po której dziedziczą wszystkie aplikacje JavaFX. Przedstawiona w przykładzie klasa `JavaFXSkel` zawiera cztery metody. Pierwszą z nich jest `main()`. Służy ona do uruchamiania aplikacji poprzez wywołanie metody `launch()`. Zwróć uwagę, że w wywołaniu metody `launch()` zostaje przekazany parametr `args` metody `main()`. Choć takie rozwiązanie jest często stosowane, to w wywołaniu metody `launch()` można także przekazać inne argumenty, można też nie przekazywać żadnych argumentów. I jeszcze jedna uwaga: stosowanie metody `launch()` jest konieczne w przypadku tworzenia niezależnych aplikacji, lecz nie w innych przypadkach. Jeśli metoda `launch()` nie będzie potrzebna, to nie trzeba także definiować metody `main()`. Niemniej jednak z podanych już wcześniej powodów w przykładowych programach przedstawionych w tym rozdziale będą używane obie te metody.

Kiedy aplikacja zaczyna działanie, system uruchomieniowy JavaFX wywołuje metodę `init()`. Dla celów demonstracyjnych metoda `init()` przedstawiona w powyższym programie wyświetla jedynie komunikat w standardowym strumieniu wyjściowym — `System.out`. Jednak zazwyczaj będzie ona

używana do wykonania czynności związanych z inicjalizacją aplikacji. Oczywiście jeśli takie czynności nie będą potrzebne, to przesłanianie metody `init()` nie będzie konieczne, gdyż dostępna jest jej pusta, domyślna implementacja. Koniecznie należy zapamiętać, że metody `init()` nie można używać do tworzenia obszaru roboczego czy sceny. Te elementy aplikacji należy tworzyć i wyświetlać w metodzie `start()`.

Po zakończeniu metody `init()` zostaje wywołana metoda `start()`. To właśnie w niej jest tworzona scena początkowa, która następnie zostaje dodana do głównego obszaru roboczego. Przeanalizujmy tę metodę wiersz po wierszu. W pierwszej kolejności zwróć uwagę, że parametrem tej metody jest obiekt klasy `Stage`. W momencie jej wywoływania parametr ten posłuży do przekazania referencji do głównego obszaru roboczego aplikacji. To właśnie w tym obszarze roboczym zostanie także zastosowana scena prezentowana w aplikacji.

Po wyświetleniu w oknie konsoli komunikatu informującego o rozpoczęciu wykonywania metody `start()` określa ona tytuł sceny, wywołując w tym celu metodę `setTitle()`:

```
myStage.setTitle("Szkielet aplikacji JavaFX.");
```

Choć ta czynność nie jest w zasadzie wymagana, to jednak jest ona zwyczajowo wykonywana w większości niezależnych aplikacji JavaFX. Ten tytuł staje się nazwą głównego okna aplikacji.

Kolejną czynnością jest utworzenie korzenia sceny. Korzeń jest jedynym węzłem w grafie sceny, który nie ma rodzica. W przedstawionej aplikacji korzeniem jest obiekt klasy `FlowPane`, jednak można w tym celu zastosować obiekty kilku innych klas.

```
FlowPane rootNode = new FlowPane();
```

Jak już wspominałem, układ `FlowPane` rozmieszcza elementy kolejno, jeden za drugim, a w razie konieczności są przenoszone do następnego wiersza. (A zatem działa on analogicznie jak klasy `FlowLayout` pakietu Swing). W typ przypadku został zastosowany układ rozmieszczający elementy w poziomie, lecz oprócz niego istnieje także układ, który rozmieszcza elementy w pionie. Choć w tej aplikacji nie jest to konieczne, to jednak można także określić dodatkowe opcje układu, takie jak wyrównanie elementów oraz pionowe i poziome odległości pomiędzy nimi.

W poniższym wierszu korzeń zostaje użyty do utworzenia obiektu `Scene`:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

Dostępnych jest kilka wersji konstruktora klasy `Scene`. Ta zastosowana w przedstawionym przykładzie tworzy scenę, która używa określonego korzenia i ma podane wymiary (odpowiednio szerokość i wysokość). Poniżej przedstawiłem tę wersję konstruktora klasy `Scene`.

```
Scene(Parent korzen, double szerokosc, double wysokosc)
```

Zwróć uwagę, że typem korzenia jest `Parent`. `Parent` to klasa pochodna klasy `Node`, reprezentująca węzły, które mogą mieć potomków. Oprócz tego zwróć także uwagę na to, że szerokość i wysokość zostały określone w formie liczb typu `double`. Dzięki temu mogą to być wartości posiadające części ułamkowe. W przedstawionym przykładzie korzeniem jest obiekt `rootNode`, a szerokość i wysokość sceny będą odpowiednio wynosić 300 i 200.

Następny wiersz kodu określa, że sceną używaną przez obiekt obszaru roboczego, `myStage`, będzie `myScene`:

```
myStage.setScene(myScene);
```

Użyta w tej instrukcji metoda `setScene()` jest zdefiniowana w klasie `Stage`; ustawia ona używaną scenę na tę, która została przekazana w jej wywołaniu.

Jeśli obiekt sceny nie będzie już więcej używany, dwie powyższe instrukcje można połączyć w jedną:

```
myStage.setScene(new Scene(rootNode, 300, 200));
```

Ze względu na zwartą postać powyższa instrukcja będzie używana w większości przykładów zamieszczonych w dalszej części rozdziału.

Ostatnia instrukcja metody `start()` wyświetla obszar roboczy i scenę:

```
myStage.show();
```

Metoda `show()` w zasadzie wyświetla okno utworzone przez obszar roboczy i scenę.

Po zamknięciu aplikacji jej okno jest usuwane z ekranu, a system uruchomieniowy JavaFX wywołuje metodę `stop()`. W powyższym przykładzie działanie tej metody ogranicza się do wyświetlenia komunikatu w oknie konsoli, dzięki czemu możemy się przekonać, że faktycznie została ona wywołana. Jednak w standardowych aplikacjach metoda ta nie wyświetla żadnych komunikatów. Co więcej, jeśli aplikacja nie musi wykonywać żadnych czynności porządkowych, to bezcelowe jest także przesłanianie metody `stop()`, gdyż klasa `Application` udostępnia jej domyślną, pustą implementację.

Kompilacja i uruchamianie programów JavaFX

Jedną z ważnych zalet platformy JavaFX jest możliwość uruchamiania tego samego programu w wielu różnych środowiskach wykonawczych. Na przykład program JavaFX można uruchamiać jako niezależną aplikację na komputerze biurowym oraz przy użyciu technologii Java Web Start. Niemniej jednak w niektórych przypadkach mogą być potrzebne różne pliki pomocnicze, na przykład plik HTML lub plik Java Network Launch Protocol (JNLP).

Ogólnie rzecz biorąc, programy JavaFX są kompilowane tak samo jak wszystkie inne programy pisane w Javie. Niemniej jednak w zależności od docelowego środowiska wykonawczego może się pojawić konieczność wykonania określonych czynności dodatkowych. Z tego względu czasami najprostszym sposobem kompilacji aplikacji JavaFX jest wykorzystanie zintegrowanego środowiska programistycznego (IDE), które potrafi w pełni obsługiwać platformę JavaFX. Jeśli chcesz kompilować i sprawdzać przykładowe aplikacje przedstawione w tym rozdziale, to bez trudu możesz to robić, używając narzędzi obsługiwanych z poziomu wiersza poleceń. Wystarczy skompilować i uruchomić aplikację w standardowy sposób — posługując się programami `javac` i `java`.

Wątek aplikacji

We wcześniejszej części rozdziału wspominałem, że nie można używać metody `init()` do tworzenia obszaru roboczego ani sceny. Obiektów tych nie można także tworzyć w konstruktorze aplikacji. Powodem tych ograniczeń jest konieczność tworzenia obu tych obiektów w **wątku aplikacji**. A jak się okazuje, zarówno metoda `init()`, jak i konstruktor aplikacji są wywoływane w wątku głównym, nazywanym także **wątkiem uruchomieniowym**. I właśnie z tego powodu ani konstruktora aplikacji, ani metody `init()` nie można używać do tworzenia obiektów `Stage` i `Scene`. Jak pokazał program przedstawiony na listingu 17.1, inicjalizację graficznego interfejsu użytkownika należy wykonywać w metodzie `start()`, gdyż ona jest wywoływana w wątku aplikacji.

Co więcej, także wszelkie zmiany w aktualnie wyświetlanym interfejsie użytkownika muszą być wykonywane w wątku aplikacji. Dodatkowo także wszystkie zdarzenia w aplikacjach JavaFX są generowane w wątku aplikacji. Dzięki temu procedur obsługi zdarzeń można używać do interakcji z graficznym interfejsem użytkownika aplikacji. Także metoda `stop()` jest wywoływana w wątku aplikacji.

Prosta kontrolka JavaFX: Label

Podstawowymi elementami wszystkich graficznych interfejsów użytkownika są kontrolki, gdyż właśnie one pozwalają użytkownikom na prowadzenie interakcji z aplikacją. Jak można się spodziewać, JavaFX udostępnia bogaty zestaw kontrolek. Najprostszą z nich jest etykieta, która pozwala wyświetlić komunikat lub obrazek. Choć etykiety są bardzo prostymi kontrolkami, to jednak doskonale nadają się do przedstawienia technik służących do tworzenia grafu sceny.

W JavaFX etykiety są instancjami klasy `Label` należącej do pakietu `javafx.scene.control`. Klasa `Label` dziedziczy między innymi po klasach `Labeled` oraz `Control`. Klasa `Labeled` definiuje kilka możliwości wspólnych dla wszystkich elementów zawierających etykiety (czyli prezentujących teksty), natomiast klasa `Control` definiuje możliwości wspólne dla wszystkich kontrolek.

Poniżej przedstawiłem konstruktor klasy `Label`, którego użyjemy w kolejnym przykładzie:

```
Label(String str)
```

Wyświetlany łańcuch znakowy jest określony przez parametr *str*.

Po utworzeniu etykiety (bądź jakiejkolwiek innej kontrolki) należy ją dodać do zawartości sceny, czyli do grafu sceny. W tym celu w pierwszej kolejności należy wywołać metodę `getChildren()` obiektu korzenia grafu sceny. Zwraca ona listę węzłów potomnych w formie obiektu typu `ObservableList<Node>`. Interfejs `ObservabelList` należy do pakietu `javafx.collections` i dziedziczy po `java.util.List`, który z kolei jest elementem biblioteki Collections Framework — platformy kolekcji języka Java. Typ `List` definiuje kolekcję reprezentującą listę obiektów. Omówienie interfejsu `List` oraz biblioteki Collections Framework wykracza poza ramy tematyczne tej książki, na szczęście jednak zastosowanie obiektu `ObservableList` do dodawania węzłów potomnych jest całkiem proste. Wystarczy wywołać metodę `add()` obiektu listy zwróconego przez metodę `getChildren()` i przekazać do niej dodawany węzeł — w naszym przypadku będzie nim obiekt etykiety.

Poniższy program (listing 17.2) stanowi praktyczną prezentację opisywanych zagadnień — tworzy on prostą aplikację JavaFX, która wyświetla etykietę.

Listing 17.2. *JavaFXLabelDemo.java*

```
// Prezentacja zastosowania etykiety w aplikacji JavaFX.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Etykiety w aplikacjach JavaFX");

    // Używa obiektu FlowPane jako korzenia.
    FlowPane rootNode = new FlowPane();

    // Tworzy scenę.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Ustawia scenę w obszarze roboczym.
    myStage.setScene(myScene);

    // Tworzy etykietę.                          Utworzenie etykiety.
    Label myLabel = new Label("JavaFX ma duże możliwości");

    // Dodaje etykietę do grafu sceny.
    rootNode.getChildren().add(myLabel);  <-- Dodanie etykiety do grafu sceny.

    // Wyświetla obszar roboczy i scenę.
    myStage.show();
  }
}
```

Ekspert odpowiada

Pytanie: Opisałeś, jak można dodawać węzeł do grafu sceny. A czy można go z tego grafu usunąć?

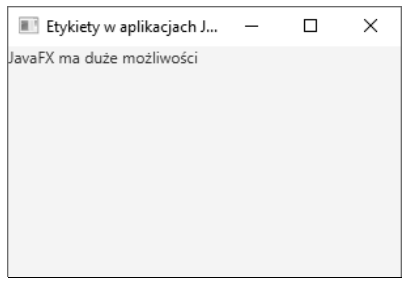
Odpowiedź: Owszem, można. Aby usunąć węzeł z grafu sceny, należy wywołać metodę `remove()` obiektu `ObservableList`. Na przykład poniższa instrukcja:

```
rootNode.getChildren().remove(myLabel);
```

usuwa kontrolkę `myLabel` ze sceny. Ogólnie rzecz biorąc, typ `ObservableList` obsługuje wiele operacji związanych z zarządzaniem listami. Poniżej podam dwa przykłady. Korzystając z metody `isEmpty()`, można sprawdzić, czy lista jest pusta; a liczbę węzłów aktualnie znajdujących się na liście można pobrać przy użyciu metody `size()`. Kiedy zaczniesz dokładniej poznawać możliwości platformy JavaFX, zapewne zechcesz także przyjrzeć się bliżej interfejsowi `ObservableList`.

Ten program wyświetli okno przedstawione na rysunku 17.1.

Rysunek 17.1. *Okno aplikacji JavaFX z etykietą*



W przedstawionym programie zwróć szczególną uwagę na poniższy wiersz kodu:

```
rootNode.getChildren().add(myLable)
```

Ta instrukcja dodaje etykietę do listy potomków, których rodzicem jest obiekt `rootNode`. Choć w razie konieczności można ją rozdzielić i zapisać w postaci dwóch odrębnych wywołań, to jednak w praktyce bardzo często spotyka się właśnie taki zapis.

Zanim zajmiemy się kolejnym zagadnieniem, warto zwrócić uwagę, że typ `ObservableList` udostępnia także metodę o nazwie `addAll()`, która pozwala dodać do grafu sceny kilka obiektów w jednym wywołaniu. Już niebawem zobaczysz przykład jej zastosowania.

Stosowanie przycisków i zdarzeń

Choć program z listingu 17.2 stanowi prosty przykład demonstrujący zastosowanie kontrolek JavaFX i tworzenie grafu sceny, to jednak nie pokazuje, w jaki sposób można obsługiwać zdarzenia. A jest to bardzo istotne zagadnienie, gdyż większość kontrolek używanych do tworzenia graficznego interfejsu użytkownika generuje zdarzenia, które są następnie obsługiwane przez aplikację. Na przykład kiedy użytkownik korzysta z takich kontrolek jak przyciski, pola wyboru oraz listy, generują one zdarzenia. Pod wieloma względami obsługa zdarzeń w JavaFX jest dosyć podobna do obsługi zdarzeń w aplikacjach Swing, opisanych w poprzednim rozdziale; choć została dodatkowo usprawniona. Jedną z bardzo często używanych kontrolek jest przycisk. Oznacza to, że zdarzenia generowane przez przyciski są jednymi z częściej obsługiwanych. Dlatego też napisanie aplikacji z przyciskiem będzie doskonałą okazją do przedstawienia sposobu obsługi zdarzeń w aplikacjach JavaFX. Z tego względu jednocześnie przedstawię podstawy obsługi zdarzeń i sposób korzystania z przycisków JavaFX.

Podstawy obsługi zdarzeń

Podstawową klasą związaną ze zdarzeniami JavaFX jest klasa `Event` zdefiniowana w pakiecie `javafx.event`. Klasa `Event` dziedziczy po `java.util.EventObject`, a to oznacza, że zdarzenia JavaFX mają te same podstawowe możliwości funkcjonalne co inne zdarzenia w języku Java. Dostępnych jest kilka klas pochodnych klasy `Event`. W tym punkcie rozdziału zajmiemy się jedną z nich, a konkretnie `ActionEvent`.

Ogólnie rzecz biorąc, model obsługi zdarzeń stosowany przez platformę JavaFX bazuje na ich przekazywaniu. Aby obsłużyć zdarzenie, w pierwszej kolejności należy zarejestrować jego obiekt nasłuchujący, który będzie oczekiwać na zdarzenia. W momencie zgłoszenia zdarzenia zostaje wywołany odpowiedni obiekt nasłuchujący, który musi odpowiedzieć na nie i zakończyć działanie. Pod tym względem zdarzenia JavaFX są obsługiwane tak samo jak zdarzenia Swing.

Zdarzenia są obsługiwane poprzez zaimplementowanie interfejsu `EventHandler`, zadeklarowanego w pakiecie `javafx.event`. Jest to interfejs sparametryzowany o następującej postaci:

```
interface EventHanderl<T extends Event>
```

Gdzie `T` jest typem obsługiwanego zdarzenia. Interfejs ten deklaruje tylko jedną metodę, `handle()`, której parametrem jest obiekt zdarzenia. Poniżej przedstawiłem postać tej metody:

```
void handle(T eventObj)
```

W tym przypadku parametr *`eventObj`* jest zgłoszonym zdarzeniem. Zazwyczaj obiekty nasłuchujące zdarzeń są implementowane przy wykorzystaniu anonimowych klas wewnętrznych lub wyrażeń lambda; choć można w tym celu także definiować niezależne klasy, jeśli takie rozwiązanie będzie odpowiednie dla tworzonej aplikacji (na przykład jeśli jeden obiekt nasłuchujący będzie obsługiwał zdarzenia pochodzące z kilku źródeł).

Przedstawienie kontrolki Button

W aplikacjach JavaFX przyciski są tworzone przy użyciu klasy `Button`, zdefiniowanej w pakiecie `javafx.scene.control`. Klasa ta dziedziczy po stosunkowo dużej liczbie klas bazowych, do których należą między innymi `ButtonBase`, `Labeled`, `Region`, `Control`, `Parent` oraz `Node`. Analizując dokumentację klasy `Button`, można się przekonać, że jej możliwości funkcjonalne pochodzą od jej klas bazowych. Co więcej, udostępnia ona wiele różnych opcji. Niemniej jednak w kolejnym przykładzie ograniczymy się do domyślnej postaci przycisków. Przyciski mogą zawierać zarówno tekst, jak i grafikę, jak również oba te elementy jednocześnie. W tym przypadku skorzystamy z przycisków prezentujących zwyczajny łańcuch znakowy.

Poniżej przedstawiłem konstruktor klasy `Button`, którego użyjemy w kolejnym przykładzie:

```
Button(String str)
```

W tym przypadku parametr *`str`* jest napisem, który zostanie wyświetlony na przycisku.

Naciśnięcie przycisku spowoduje wygenerowanie zdarzenia `ActionEvent`. Klasa `ActionEvent` została zdefiniowana w pakiecie `javafx.event`. Obiekt nasłuchujący tego zdarzenia można zarejestrować, wywołując metodę `setOnAction()` obiektu `Button`. Poniżej przedstawiłem jej ogólną postać:

```
final void setOnAction(EventHandler<ActionEvent> handler)
```

przy czym parametr *`handler`* jest rejestrowanym obiektem nasłuchującym. Jak już wspominałem, obiekty nasłuchujące często są implementowane jako anonimowe klasy wewnętrzne lub wyrażenia lambda. Metoda `setOnAction()` ustawia wartość właściwości `onAction` przechowującej referencję obiektu nasłuchującego. Podobnie jak we wszystkich innych przypadkach obsługi zdarzeń w języku Java obiekt nasłuchujący zdarzeń musi odpowiadać na nie możliwie jak najszybciej, a następnie kończyć działanie. Jeśli obsługa zdarzeń będzie zajmować zbyt wiele czasu, doprowadzi ona do zauważalnego zwolnienia działania aplikacji. W sytuacjach, gdy obsługa zdarzenia zabiera dużo czasu, należy ją realizować w odrębnym wątku.

Przedstawienie obsługi zdarzeń i stosowania przycisków

Poniższy program zamieszczony na listingu 17.3 przedstawia obsługę zdarzeń oraz sposób korzystania z przycisków w aplikacjach JavaFX. Przedstawiona aplikacja wyświetla dwa przyciski oraz etykietę. Przyciski noszą odpowiednio nazwy: *Góra* i *Dół*. Za każdym razem, gdy użytkownik kliknie przycisk, w etykiecie zostanie wyświetlony odpowiadający mu tekst. Jak widać, aplikacja ta działa podobnie do programu `JButton` przedstawionego w poprzednim rozdziale. Całkiem interesujące może być porównanie kodu obu tych aplikacji.

Listing 17.3. *JavaFXEventDemo.java*

```
// Prezentacja stosowania przycisków i obsługi zdarzeń
// w aplikacji JavaFX.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Przyciski i zdarzenia JavaFX");

    // Jako korzeń zostaje użyty panel FlowPane. W tym
    // przypadku pionowe i poziome odstępy pomiędzy umieszczonymi
    // w nim kontrolkami będą wynosić 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Wyrównuje kontrolki do środka.
    rootNode.setAlignment(Pos.CENTER);

    // Tworzy obiekt Scene.
    Scene myScene = new Scene(rootNode, 300, 100);

    // Dodaje obiekt Scene do obiektu Stage.
    myStage.setScene(myScene);

    // Tworzy etykietę.
    response = new Label("Przycisk");

    // Tworzy dwa przyciski.
    Button btnUp = new Button("Góra");
    Button btnDown = new Button("Dół");

    // Obsługa zdarzenia ActionEvent przycisku Góra.
    btnUp.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Nacisnąłeś przycisk Góra.");
      }
    });

    // Obsługa zdarzenia ActionEvent przycisku Dół.
    btnDown.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Nacisnąłeś przycisk Dół.");
      }
    });
```

Utworzenie dwóch przycisków.

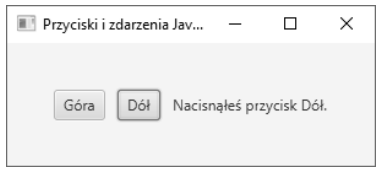
Zdefiniowanie obiektów nasłuchujących zdarzeń `ActionEvent` przycisków.

Zdefiniowanie obiektów nasłuchujących zdarzeń `ActionEvent` przycisków.

```
    // Dodaje etykietę i przyciski do grafu sceny.
    rootNode.getChildren().addAll(btnUp, btnDown, response);

    // Wyświetla scenę i obszar roboczy.
    myStage.show();
  }
}
```

Na rysunku 17.2 przedstawiłem okno tej aplikacji.

Rysunek 17.2. *Okno aplikacji z dwoma przyciskami i etykietą*



Przeanalizujmy teraz kluczowe fragmenty tego programu. W pierwszej kolejności zwróć uwagę na sposób, w jaki są tworzone oba przyciski:

```
Button btnUp = new Button("Góra");
Button btnDown = new Button("Dół");
```

Te dwie instrukcje tworzą dwa przyciski prezentujące komunikaty tekstowe. Na pierwszym z nich wyświetlany jest napis `Góra`, a na drugim napis `Dół`.

Następnie tworzone są obiekty nasłuchujące zdarzeń `ActionEvent` generowanych przez oba przyciski. Poniższy fragment kodu pokazuje tworzenie tego obiektu nasłuchującego dla przycisku `Góra`:

```
// Obsługa zdarzenia Action przycisku Góra.
btnUp.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    response.setText("Nacisnąłeś przycisk Góra.");
  }
});
```

Jak już zaznaczałem, przyciski odpowiadają na zdarzenia typu `ActionEvent`. Aby zarejestrować obiekt nasłuchujący tych zdarzeń, należy wywołać metodę `setOnAction()` przycisku. W powyższym przykładzie do zaimplementowania interfejsu `EventHandler` zastosowaliśmy anonimową klasę wewnętrzną. (Przypomnij sobie, że interfejs ten deklaruje tylko jedną metodę — `handle()`.) Wewnątrz metody `handle()`, w etykiecie `response`, zostanie zapisany tekst odpowiadający kliknięte mu przyciskowi. Zwróć uwagę, że w tym celu wywołujemy metodę `setText()` kontrolki etykiety. Zdarzenia przycisku `Dół` są obsługiwane w analogiczny sposób.

Po określeniu obiektów nasłuchujących zdarzeń etykieta `response` oraz przyciski `btnUp` oraz `btnDown` zostają dodane do grafu sceny. W tym celu wywołujemy metodę `addAll()`.

```
rootNode.getChildren().addAll(btnUp, btnDown, response);
```

Metoda `addAll()` dodaje listę węzłów do węzła rodzica, na rzecz którego została wywołana. Oczywiście nic nie stoi na przeszkodzie, by dodać węzły, używając trzech niezależnych wywołań metody `add()`; jednak w tej sytuacji zastosowanie metody `addAll()` jest wygodniejsze.

W powyższym programie warto też zwrócić uwagę na dwa inne, interesujące zagadnienia związane ze sposobem wyświetlania kontrolek w oknie aplikacji. Pierwszym z nich jest instrukcja użyta do utworzenia korzenia:

```
FlowPane rootNode = new FlowPane(10, 10);
```

Jak widać, w wywołaniu konstruktora klasy `FlowPane` zostały przekazane dwa argumenty. Określają one odpowiednio wielkość poziomego i pionowego odstępu pomiędzy elementami sceny. Jeśli odstępy te nie zostaną określone, to dwa elementy sceny (na przykład dwa przyciski) są rozmieszczane w taki sposób, że nie będzie pomiędzy nimi żadnego odstępu. A zatem w takim przypadku elementy zostaną wyświetlone bezpośrednio jeden przy drugim, tworząc interfejs użytkownika o bardzo nieatrakcyjnym wyglądzie. Problem ten można rozwiązać, podając wielkość odstępów.

Interesujący jest także poniższy wiersz kodu, określający wyrównanie kontrolek umieszczanych w układzie FlowPane:

```
rootNode.setAlignment(Pos.CENTER);
```

W tym przypadku zawartość elementu ma zostać wyśrodkowana. Efekt ten uzyskujemy, wywołując metodę setAlignment() obiektu FlowPane. Zastosowanie wartości Pos.CENTER oznacza, że zawartość zostanie wyśrodkowana zarówno w poziomie, jak i w pionie. Dostępne są także inne opcje wyrównania. Pos jest typem wyliczeniowym, zawierającym stałe określające różne wyrównania. Został on zdefiniowany w pakiecie javafx.geometry.

Zanim przejdziemy dalej, chciałbym zwrócić uwagę na jeszcze jedno zagadnienie. Otóż w powyższym programie do obsługi zdarzeń przycisków zostały wykorzystane anonimowe klasy wewnętrzne. Jednak ponieważ interfejs EventHandler deklaruje tylko jedną metodę abstrakcyjną, handle(), zatem w wywołaniu metody setOnAction() można także używać wyrażeń lambda. Na przykład poniższy fragment kodu pokazuje, jak wyglądałaby obsługa przycisku Up, gdyby wcześniejszy kod przepisać z użyciem wyrażenia lambda:

```
btnUp.setOnAction( (ae) ->
                    response.setText("Nacisnąłeś przycisk Góra");
                 );
```

Zwróć uwagę, że zapis wyrażeń lambda do anonimowych klas wewnętrznych jest znacząco krótszy i prostszy. (Wyrażeń lambda użyjesz także na samym końcu tego rozdziału, kiedy będziesz modyfikował powyższy program w ramach odpowiedzi na pytanie nr 10 z testu sprawdzającego).

Trzy kolejne kontrolki JavaFX

Platforma JavaFX definiuje obszerny zestaw kontrolek, które zostały umieszczone w pakiecie javafx.
↪scene.control. W poprzednich punktach rozdziału poznałeś już dwie takie kontrolki: Label oraz Button. W tym punkcie rozdziału przyjrzymy się trzem kolejnym: CheckBox, ListView oraz TextField. Zgodnie z tym, co sugerują ich nazwy, pozwalają one na tworzenie odpowiednio: pól wyboru, list oraz pól tekstowych. Wszystkie razem stanowią reprezentatywny przykład kontrolek JavaFX. Pozwalają także przedstawić kilka powszechnie stosowanych technik. Kiedy opanujesz te techniki, pozostałe kontrolki będziesz mógł poznawać samodzielnie.

Przedstawione tu kontrolki udostępniają możliwości funkcjonalne podobne do analogicznych kontrolek pakietu Swing, opisanych w poprzednim rozdziale. Podczas lektury tego punktu rozdziału ciekawe może być porównanie sposobów implementacji odpowiadających sobie kontrolek obu pakietów.

Pola wyboru

W JavaFX pola wyboru są reprezentowane przez klasę CheckBox. Jej bezpośrednią klasą bazową jest ButtonBase. Innymi słowy, pola wyboru są specjalnym rodzajem przycisków. Choć bez wątpienia doskonale znasz pola wyboru, gdyż są one powszechnie stosowanymi elementami interfejsu użytkownika, to jednak w JavaFX są one nieco bardziej wyszukane, niż można by początkowo sądzić. Wynika to z faktu, że pola wyboru reprezentowane przez klasę CheckBox mogą mieć trzy stany. Pierwsze dwa — pole zaznaczone i niezaznaczone — są zgodne z oczekiwaniami i odpowiadają domyślnemu sposobowi działania. Dostępny jest jednak także trzeci stan — **nieznany** (nazywany także **niezdefiniowanym**). Zazwyczaj jest on używany do zasygnalizowania, że stan pola wyboru nie został jeszcze określony bądź też nie ma on odniesienia do bieżącej sytuacji. Aby móc korzystać z tego stanu pośredniego, trzeba go jawnie włączyć. Niezbędna procedura została przedstawiona w przykładzie 17.1. W przykładowym programie zamieszczonym na listingu 17.4 przedstawione zostały pola wyboru działające w tradycyjny sposób.

Poniżej przedstawiłem konstruktor CheckBox, którego będziemy używać:

```
CheckBox(String str)
```

Tworzy on pole wyboru wraz z etykietą, której treść jest określona przez parametr *str*. Podobnie jak w przypadku przycisków zaznaczenie pola wyboru powoduje zgłoszenie zdarzenia ActionEvent.

Poniższy program (listing 17.4) przedstawia tworzenie i stosowanie pól wyboru. Wyświetla on cztery takie pola reprezentujące różne typy komputerów. Noszą one następujące nazwy: *Smartfon*, *Tablet*, *Netbook* oraz *Stacjonarny*. Za każdym razem, gdy zmienia się stan pola wyboru, generowane jest zdarzenie. Zdarzenie to jest obsługiwane poprzez wyświetlenie nowego stanu pola (zaznaczenie go lub usunięcie zaznaczenia) oraz przez wyświetlenie listy wszystkich zaznaczonych pól.

Listing 17.4. *CheckboxDemo.java*

```
// Program demonstrujący stosowanie pól wyboru.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

  CheckBox cbSmartphone;
  CheckBox cbTablet;
  CheckBox cbNotebook;
  CheckBox cbDesktop;

  Label response;
  Label selected;

  String computers;

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Prezentacja pól wyboru");

    // Jako korzeń zostaje użyty panel FlowPane. W tym
    // przypadku pionowe i poziome odstępy pomiędzy umieszczonymi
    // w nim kontrolkami będą wynosić 10.
    FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);

    // Wyrównuje kontrolki do środka.
    rootNode.setAlignment(Pos.CENTER);
    // Tworzy obiekt Scene.
    Scene myScene = new Scene(rootNode, 230, 200);

    // Dodaje obiekt Scene do obiektu Stage.
    myStage.setScene(myScene);

    Label heading = new Label("Jakie komputery posiadasz?");

    // Tworzy etykietę, która będzie nas informować o zmianie
    // stanu pola wyboru.
    response = new Label("");

    // Tworzy etykietę, która będzie nas informować o tym,
    // które pola są zaznaczone.
```

```
    selected = new Label("");

    // Tworzy cztery pola wyboru.
    cbSmartphone = new CheckBox("Smartfon");
    cbTablet = new CheckBox("Tablet");
    cbNotebook = new CheckBox("Notebook");
    cbDesktop = new CheckBox("Stacjonarny");

    // Obsługa zdarzeń ActionEvent pól wyboru.
    cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbSmartphone.isSelected())
          response.setText("Zaznaczono pole 'smartfon'.");
        else
          response.setText("Usunięto zaznaczenie pola 'smartfon'.");

        showAll();
      }
    });

    cbTablet.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbTablet.isSelected())
          response.setText("Zaznaczono pole 'tablet'.");
        else
          response.setText("Usunięto zaznaczenie pola 'tablet'.");

        showAll();
      }
    });

    cbNotebook.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbNotebook.isSelected())
          response.setText("Zaznaczono pole 'notebook'.");
        else
          response.setText("Usunięto zaznaczenie pola 'notebook'.");

        showAll();
      }
    });

    cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbDesktop.isSelected())
          response.setText("Zaznaczono pole 'stacjonarny'.");
        else
          response.setText("Usunięto zaznaczenie pola 'stacjonarny'.");

        showAll();
      }
    });

    // Dodaje kontrolki do grafu sceny.
    rootNode.getChildren().addAll(heading, cbSmartphone, cbTablet,
                                  cbNotebook, cbDesktop, response, selected);

    // Wyświetla scenę i obszar roboczy.
    myStage.show();

    showAll();
  }

  // Aktualizuje zawartość okna aplikacji.
```

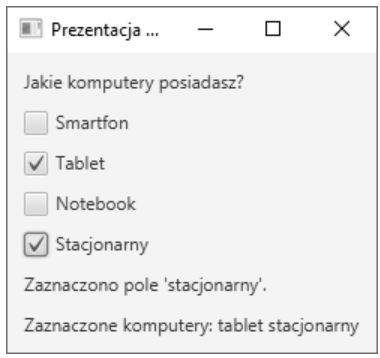
Utworzenie pól wyboru.

Obsługa zdarzeń pól wyboru.

```
  void showAll() {
    computers = "";
    if(cbSmartphone.isSelected()) computers = "smartfon ";
    if(cbTablet.isSelected()) computers += "tablet ";
    if(cbNotebook.isSelected()) computers += "notebook";
    if(cbDesktop.isSelected()) computers += "stacjonarny";

    selected.setText("Zaznaczone komputery: " + computers);
  }
}
```

Sprawdzenie stanu pól wyboru przy użyciu metody `isSelected()`.

Przykładowy wygląd okna tego programu przedstawia rysunek 17.3.



Rysunek 17.3. *Okno aplikacji demonstrującej działanie pól wyboru*

Sposób działania tego programu jest wyjątkowo prosty. Za każdym razem, gdy zostanie zmieniony stan któregoś z pól wyboru, generowane jest zdarzenie `ActionEvent`. Obiekty nasłuchujące tych zdarzeń w pierwszej kolejności informują, czy dane pole zostało zaznaczone, czy też zaznaczenie zostało usunięte. W tym celu na rzecz kontrolki, która wygenerowała zdarzenie, wywoływana jest metoda `isSelected()`. Zwraca ona wartość `true`, jeśli pole wyboru zostało właśnie zaznaczone, bądź wartość `false`, jeśli zaznaczenie zostało z niego właśnie usunięte. Następnie wywoływana jest metoda `showAll()`, która wyświetla wszystkie zaznaczone pola wyboru.

W powyższym programie warto zwrócić uwagę na jeszcze jedno, interesujące rozwiązanie. Zauważ, że panel działa w układzie pionowym i jest tworzony w następujący sposób:

```
FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);
```

Domyślnie panele `FlowPane` działają w układzie poziomym. Zmiana jego działania nastąpiła w wyniku przekazania stałej `Orientation.VERTICAL` jako pierwszego argumentu wywołania konstruktora.

Przykład 17.1. Zastosowanie stanu nieznanego pól wyboru

CheckboxDemo.java

Jak już wcześniej napisałem, domyślnie kontrolki `CheckBox` implementują dwa stany: zaznaczony i niezaznaczony. Kontrolka ta udostępnia jednak także trzeci stan — nieznany — którego można użyć, by zasygnalizować, że stan pola nie został jeszcze określony bądź też nie nadaje się on do użycia w bieżącej sytuacji. Możliwość korzystania z tego trzeciego stanu należy jawnie włączyć, gdyż stan ten nie jest domyślnie stosowany. Oprócz tego także obiekt nasłuchujący zdarzeń pola wyboru musi obsługiwać trzeci stan. Ten przykład ilustruje proces korzystania ze stanu nieznanego. W tym celu możliwość stosowania tego stanu jest włączana w polu wyboru `smartfon` przedstawionego wcześniej programu `CheckboxDemo`.

1. Aby włączyć możliwość stosowania stanu nieznanego w polu wyboru, wywołaj metodę `setAllowIndeterminate()` odpowiedniej kontrolki `CheckBox`:

   ```
   final void setAllowIndeterminate(boolean enable)
   ```

Jeśli parametr *enable* przyjmie wartość `true`, to stan nieznany będzie można stosować; w przeciwnym razie będzie on niedostępny. Po włączeniu tego trzeciego stanu użytkownik będzie mógł wybierać pomiędzy trzema stanami pola: zaznaczonym, niezaznaczonym i nieznanym. Aby włączyć możliwość stosowania stanu nieznanego w polu `smartfon`, musisz wykonać poniższą instrukcję:

```
cbSmartphone.setAllowIndeterminate(true);
```

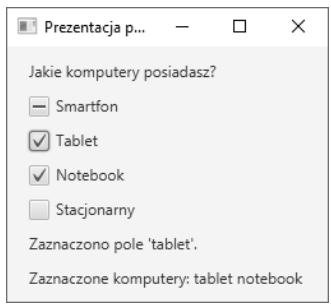
2. Aby określić, czy pole wyboru znajduje się w stanie nieznanym, należy wywołać przedstawioną poniżej metodę `isIndeterminate()`:

```
final boolean isIndeterminated();
```

Metoda ta zwraca wartość `true`, jeśli pole wyboru znajduje się w stanie nieznanym, a wartość `false` w przeciwnym razie. Obiekt nasłuchujący, który obsługuje zdarzenia pola wyboru, będzie musiał sprawdzać, czy pole znajduje się w tym trzecim stanie. W tym celu zmodyfikuj obiekt nasłuchujący zdarzeń generowanych przez pole `smartfon` w następujący sposób:

```
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    if(cbSmartphone.isIndeterminate())
      response.setText("Stan pola 'smartfon' jest nieznany");
    else if (cbSmartphone.isSelected())
      response.setText("Zaznaczono pole 'smartfon'.");
    else
      response.setText("Usunięto zaznaczenie pola 'smartfon'.");

    showAll();
  }
});
```

3. Po wprowadzeniu tych zmian skompiluj i uruchom program. Teraz już będziesz mógł zmienić stan pola wyboru `smartfon` na nieznany, jak pokazałem na rysunku 17.4.



Rysunek 17.4. *Stosowanie stanu nieznanego pól wyboru*

Listy

Kolejną często stosowaną kontrolką są listy. W JavaFX listy są reprezentowane przez obiekty klasy `ListView`. Kontrolki te wyświetlają listę elementów i pozwalają na zaznaczenie jednego bądź kilku z nich. Jedną z bardzo przydatnych cech kontrolek `ListView` jest automatyczne wyświetlanie pasków przewijania, jeśli liczba wszystkich elementów listy przewyższa liczbę elementów, które mogą być w niej jednocześnie wyświetlone. Ze względu na tę możliwość efektywnego wykorzystania ograniczonego obszaru na ekranie kontrolki `ListView` są często stosowaną alternatywą dla innych kontrolek pozwalających na wybieranie opcji.

`ListView` jest klasą sparametryzowaną, zadeklarowaną w następujący sposób:

```
class ListView<T>
```

Przy czym `T` określa typ elementów przechowywanych na liście; często są to obiekty klasy `String`, choć mogą to być także obiekty innych typów.

Poniżej przedstawiłem postać konstruktora `ListView`, którego użyjemy w kolejnym przykładzie.

```
ListView(ObservableList<T> list)
```

Listę elementów, które należy wyświetlić w kontrolce, określa parametr *list*. Jak widać, jest to obiekt typu `ObservableList`. Jak już napisałem wcześniej, `ObservableList` jest typem reprezentującym listę obiektów. Domyślnie kontrolki `ListView` pozwalają na zaznaczanie tylko jednego elementu listy w danej chwili. Możliwość zaznaczania wielu elementów można włączyć, zmieniając tryb wyboru kontrolki; niemniej jednak w przedstawionych tu przykładach będziemy używali domyślnego trybu, pozwalającego na zaznaczenie tylko jednego elementu listy.

Chyba najprostszym sposobem utworzenia obiektu `ObservableList`, którego będzie można użyć do utworzenia kontrolki `ListView`, jest skorzystanie z metody fabrycznej `observableArrayList()`, która jest statyczną metodą klasy `FXCollections` (zdefiniowanej w pakiecie `javafx.collections`). Poniżej przedstawiłem wersję, której użyjemy:

```
static <E> ObservableList<E> observableArrayList(E ... elements)
```

gdzie `E` określa typ elementów przekazany jako parametr *elementes*.

Choć kontrolka `ListView` ma swoją domyślną wielkość, to jednak może się zdarzyć, że będziemy chcieli dostosować jej wysokość bądź szerokość do bieżących potrzeb. Jednym ze sposobów, by to zrobić, jest skorzystanie z przedstawionych poniżej metod `setPrefHeight()` oraz `setPrefWidth()`:

```
final void setPrefHeight(double height)

final void setPrefWidth(double width)
```

Ewentualnie można także użyć metody `setPrefSize()`, by w jednym wywołaniu określić oba wymiary kontrolki:

```
void setPrefSize(double width, double height)
```

Kontrolki `ListView` można używać na dwa podstawowe sposoby. Pierwszy z nich polega na zignorowaniu zdarzeń generowanych przez listę i pobieraniu jej zaznaczonych elementów wtedy, gdy program będzie ich potrzebować. Drugi sposób bazuje na monitorowaniu zmian w zaznaczonych elementach listy poprzez zarejestrowanie odpowiednich obiektów nasłuchujących. To rozwiązanie pozwala reagować na zmiany za każdym razem, gdy użytkownik zmodyfikuje zaznaczony element lub elementy listy. I właśnie to drugie rozwiązanie przedstawię w kolejnym przykładzie.

Obiekt nasłuchujący zdarzeń generowanych przez listy korzysta z interfejsu `ChangeListener` zdefiniowanego w pakiecie `javafx.beans.value`. Interfejs ten definiuje tylko jedną metodę: `change()`. Poniżej przedstawiłem jej deklarację:

```
void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)
```

W tym przypadku parametr *changed* jest obiektem typu `ObservableValue<T>`, który można obserwować w celu sprawdzania zmian. Parametry *oldVal* i *newVal* zawierają odpowiednio poprzednią i nową wartość. A zatem parametr *newVal* zawiera referencję do elementu listy, który właśnie został zaznaczony.

Aby odbierać zdarzenia związane ze zmianami zaznaczonych elementów listy, w pierwszej kolejności należy pobrać model zaznaczania używany przez daną kontrolkę `ListView`. Służy do tego przedstawiona poniżej metoda `getSelectionModel()`:

```
final MultipleSelectionModel<T> getSelectionModel()
```

Zwraca ona referencję obiektu modelu zaznaczania. `MultipleSelectionModel` jest klasą definiującą model pozwalający na zaznaczanie wielu elementów listy, która dziedziczy po klasie `SelectionModel`. Jednak zaznaczanie wielu elementów listy jest możliwe dopiero po włączeniu odpowiedniego modelu zaznaczania.

Używając obiektu modelu zwróconego przez wywołanie `getSelectionModel()`, należy pobrać referencję do właściwości wybranego elementu, definiującej, co ma się stać, kiedy element listy zostanie zaznaczony. Do tego celu służy przedstawiona poniżej metoda `selectedItemProperty()`:

```
final ReadOnlyObjectProperty<T> selectedItemProperty()
```

Obiekt nasłuchujący dodaje się właśnie do tej właściwości, wywołując jej metodę `addListener()`. Poniżej przedstawiłem deklarację tej metody:

```
void addListener(ChangeListener<? super T> listener)
```

W tym przypadku `T` określa typ właściwości.

Program pokazany na listingu 17.5 przedstawia praktyczne zastosowanie opisanych wcześniej informacji. Tworzy on kontrolkę wyświetlającą listę typów komputerów i daje użytkownikowi możliwość wybrania jej jednego elementu. Po wybraniu elementu program wyświetla jego nazwę.

Listing 17.5. *ListViewDemo.java*

```
// Prezentacja kontrolki ListView.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;

public class ListViewDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Prezentacja kontrolki ListView");

    // Jako korzeń zostaje użyty panel FlowPane. W tym
    // przypadku pionowe i poziome odstępy pomiędzy umieszczonymi
    // w nim kontrolkami będą wynosić 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Wyrównuje kontrolki do środka.
    rootNode.setAlignment(Pos.CENTER);

    // Tworzy obiekt Scene.
    Scene myScene = new Scene(rootNode, 200, 120);

    // Dodaje obiekt Scene do obiektu Stage.
    myStage.setScene(myScene);

    // Tworzy etykietę.
    response = new Label("Wybierz typ komputera");

    // Tworzy listę ObservableList, zawierającą elementy, które
    // mają być wyświetlone w kontrolce listy.
    ObservableList<String> computerTypes =
      FXCollections.observableArrayList("Smartfon", "Tablet", "Notebook",
                                        "Stacjonarny" );

    // Tworzy kontrolkę ListView.
    ListView<String> lvComputers = new ListView<String>(computerTypes);

    // Określa preferowaną wysokość i szerokość.
    lvComputers.setPrefSize(100, 70);
```

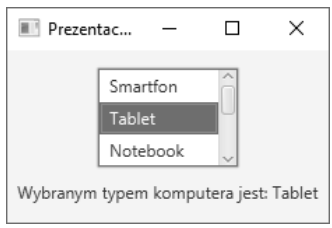
Utworzenie kontrolki `ListView`, prezentującej elementy umieszczone na liście `computerTypes`.

```
    // Pobiera model zaznaczania elementów listy.
    MultipleSelectionModel<String> lvSelModel =
                                    lvComputers.getSelectionModel();

    // Tworzy obiekt nasłuchujący zdarzeń, który będzie odpowiadał na
    // zmiany zaznaczenia elementów listy.
    lvSelModel.selectedItemProperty().addListener(
                                    new ChangeListener<String>() {
      public void changed(ObservableValue<? extends String> changed,
                          String oldVal, String newVal) {

        // Wyświetla nazwę wybranego elementu listy.
        response.setText("Wybranym typem komputera jest: " + newVal);
      }
    });

    // Dodaje etykietę i kontrolkę listy do grafu sceny.
    rootNode.getChildren().addAll(lvComputers, response);

    // Wyświetla scenę i obszar roboczy.
    myStage.show();
  }
}
```

Obsługa zdarzeń zmiany zaznaczonego elementu listy.

Przykładową postać okna tego programu przedstawia rysunek 17.5.

Rysunek 17.5.
Przykład kontrolki ListView



Zwróć uwagę, że lista posiada pionowy pasek przewijania, dzięki któremu można wyświetlić wszystkie jej elementy. Jak już wspominałem, kiedy wielkość zawartości kontrolki `ListView` przekracza jej wymiary, to automatycznie jest do niej dodawany pasek przewijania. Dzięki temu korzystanie z kontrolki `ListView` jest bardzo wygodne.

W powyższym programie trzeba zwrócić szczególną uwagę na sposób tworzenia kontrolki `ListView`. W pierwszej kolejności tworzony jest obiekt `ObservableList`:

```
ObservableList<String> computerTypes =
  FXCollections.observableArrayList("Smartfon", "Tablet", "Notebook",
                                    "Stacjonarny" );
```

Powyższa instrukcja korzysta z metody `observableArrayList()`, aby utworzyć listę łańcuchów znakowych. Następnie utworzony obiekt `ObservableList` zostaje użyty do zainicjowania kontrolki `ListView`:

```
ListView<String> lvComputers = new ListView<String>(computerTypes);
```

Następnie program określa preferowaną szerokość i wysokość kontrolki.

Kolejną czynnością, na którą należy zwrócić uwagę, jest sposób pobierania modelu zaznaczania używanego w kontrolce `lvComputers`:

```
MultipleSelectionModel<String> lvSelModel =
                                lvComputers.getSelectionModel();
```

Jak już wspominałem, kontrolka `ListView` używa modelu zaznaczania `MultipleSelectionModel` nawet wtedy, gdy dozwolone jest zaznaczenie tylko jednego elementu listy. Kolejną czynnością jest wywołanie metody `selectedItemProperty()` obiektu modelu i zarejestrowanie obiektu nasłuchującego; operacje te realizuje poniższy blok kodu:

```
lvSelModel.selectedItemProperty().addListener(
                                    new ChangeListener<String>() {
  public void changed(ObservableValue<? extends String> changed,
                      String oldVal, String newVal) {

    // Wyświetla nazwę wybranego elementu listy.
    response.setText("Wybranym typem komputera jest: " + newVal);
  }
});
```

W ramach ciekawostki należy zaznaczyć, że ten sam podstawowy mechanizm zastosowany w tym przykładzie do obsługi zdarzeń zmiany wybranego elementu listy może zostać użyty w dowolnych innych kontrolkach generujących te zdarzenia.

Ekspert odpowiada

Pytanie: W jaki sposób można włączyć możliwość zaznaczania wielu elementów w kontrolkach `ListView`?

Odpowiedź: Jeśli podczas korzystania z kontrolek `ListView` chcesz mieć możliwość zaznaczania więcej niż jednego elementu listy, musisz tego jawnie zażądać. W tym celu musisz ustawić odpowiedni tryb zaznaczania, wywołując metodę `setSelectionMode()` obiektu modelu zaznaczania. Metoda ta ma następującą postać:

```
final void setSelectionMode(SelectionMode mode)
```

W tym przypadku parametr *mode* musi być wartością `SelectionMode.MULTIPLE` lub `SelectionMode.SINGLE`. Aby włączyć możliwość zaznaczania wielu elementów listy, trzeba użyć wartości `SelectionMode.MULTIPLE`.

Jednym ze sposobów pobrania listy aktualnie zaznaczonych elementów kontrolki `ListView` jest wywołanie metody `getSelectedItems()` obiektu modelu zaznaczania. Poniżej przedstawiłem deklarację tej metody:

```
ObservableList<T> getSelectedItems()
```

Zwraca ona obiekt typu `ObservableList`. Po uzyskaniu obiektu `ObservableList` można sprawdzić zaznaczone elementy, używając rozszerzonej pętli `for`.

Pola tekstowe

Kontrolki takie jak `Button`, `CheckBox` oraz `ListView` są oczywiście całkiem przydatne, jednak wszystkie zapewniają możliwość wyboru jednej z predefiniowanych opcji bądź akcji. Jednak czasami będziemy chcieli, by użytkownik sam wpisał jakiś łańcuch znakowy. Aby udostępnić ten rodzaj wprowadzania danych, JavaFX zawiera kilka kontrolek tekstowych. W tym rozdziale przyjrzymy się jednej z nich, a konkretnie kontrolce `TextField`. Pozwala ona na wpisywanie jednego wiersza tekstu. Można jej zatem używać do pobierania imion, identyfikatorów, adresów i tym podobnych informacji. Kontrolka `TextField`, podobnie jak wszystkie pozostałe tekstowe kontrolki JavaFX, dziedziczy po klasie `TextInputControl`, która definiuje znaczną część ich możliwości funkcjonalnych.

Klasa `TextField` definiuje dwa konstruktory. Pierwszym z nich jest konstruktor domyślny, który tworzy puste pole tekstowe o domyślnej wielkości. Z kolei drugi konstruktor pozwala określić początkową zawartość pola. W tym rozdziale będziemy używać konstruktora domyślnego.

Choć czasami domyślna wielkość pola tekstowego może odpowiadać naszym wymaganiom, to jednak zazwyczaj będziemy chcieli ją określić. Można to zrobić, wywołując przedstawioną poniżej metodę `setPrefColumnCount()`:

```
final void setPrefColumnCount(int columns)
```

Kontrolka używa wartości *columns* do określenia swojej szerokości.

Tekst wyświetlany w kontrolce można określić przy użyciu metody `setText()`. Z kolei jej bieżącą zawartość można pobrać, wywołując metodę `getText()`. Oprócz tych podstawowych operacji kontrolki `TextField` udostępniają także kilka innych funkcji, które warto poznać, takich jak: wycinanie, wklejanie oraz dołączanie tekstu. Istnieje także możliwość programowego zaznaczenia fragmentu tekstu wpisanego w kontrolce.

Jednym ze szczególnie użytecznych zadań kontrolki `TextField` jest określanie komunikatu informacyjnego wyświetlanego, gdy jest ona pusta. Można go określić, używając przedstawionej poniżej metody `setPromptText()`:

```
final void setPromptText(String str)
```

W tym przypadku parametr *str* jest łańcuchem znakowym, który początkowo zostanie wyświetlony w polu tekstowym, jeszcze zanim użytkownik coś w nim wpisze. Tekst ten jest wyświetlany kolorem o niskiej intensywności (czyli zazwyczaj na szaro).

Kiedy podczas wpisywania tekstu w kontrolce `TextField` użytkownik naciśnie klawisz *Enter*, generowane jest zdarzenie `ActionEvent`. Choć czasami obsługa tych zdarzeń może się przydać, to jednak zazwyczaj zamiast je obsługiwać, programy po prostu pobierają tekstową zawartość pola, kiedy będzie to potrzebne. Kolejny program, przedstawiony na listingu 17.6, demonstruje oba te sposoby korzystania z kontrolek `TextField`. Program ten tworzy pole tekstowe, w którym użytkownik ma wpisać swoje imię. Kiedy podczas wpisywania imienia użytkownik naciśnie klawisz *Enter* lub kliknie przycisk *Pobierz imię*, łańcuch znakowy wpisany w polu zostanie pobrany i wyświetlony. Zwróć także uwagę na wyświetlany w polu komunikat informacyjny.

Listing 17.6. *TextFieldDemo.java*

```
// Prezentacja stosowania pól tekstowych.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

  TextField tf;
  Label response;

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Prezentacja pola tekstowego");

    // Jako korzeń zostaje użyty panel FlowPane. W tym
    // przypadku pionowe i poziome odstępy pomiędzy umieszczonymi
    // w nim kontrolkami będą wynosić 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Wyrównuje kontrolki do środka.
    rootNode.setAlignment(Pos.CENTER);

    // Tworzy obiekt Scene.
    Scene myScene = new Scene(rootNode, 230, 140);

    // Dodaje obiekt Scene do obiektu Stage.
    myStage.setScene(myScene);
```

```
        // Tworzy etykietę pola tekstowego.
        response = new Label("Jak masz na imię: ");

        // Tworzy przycisk do pobierania tekstu.
        Button btnGetText = new Button("Pobierz imię");

        // Tworzy pole tekstowe.
        tf = new TextField();
        // Określa komunikat informacyjny.
        tf.setPromptText("Wpisz imię.");
        // Podaje preferowaną liczbę kolumn — wielkość pola.
        tf.setPrefColumnCount(15);
        // Używamy wyrażenia lambda do obsługi zdarzeń ActionEvent
        // generowanych przez pole tekstowe. Zdarzenia te są
        // generowane, gdy użytkownik naciśnie klawisz ENTER
        // podczas wpisywania tekstu w polu. W tym przypadku obsługa
        // zdarzenia sprowadza się do pobrania i wyświetlenia
        // wpisanego imienia.
        tf.setOnAction( (ae) -> response.setText("Naciśnięto ENTER. " +
                                 "Wpisane imię to: " + tf.getText()));
        // Wyrażenia lambda używamy także do pobrania i wyświetlenia
        // imienia po kliknięciu przycisku.
        btnGetText.setOnAction((ae) ->
                               response.setText("Kliknięto przycisk. " +
                                   "Wpisane imię to: " + tf.getText()));

        // Używa separatora, aby ładniej rozmieścić elementy
        // w układzie.
        Separator separator = new Separator();
        separator.setPrefWidth(180);

        // Dodaje etykietę i kontrolkę listy do grafu sceny.
        rootNode.getChildren().addAll(tf, btnGetText, separator, response);

        // Wyświetla scenę i obszar roboczy.
        myStage.show();
    }
}
```

Utworzenie pola tekstowego.

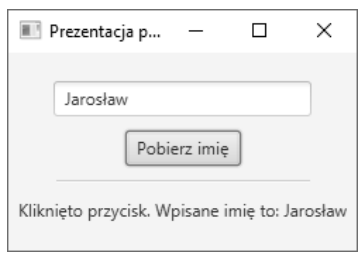
Określenie treści komunikatu informacyjnego.

Określenie szerokości pola w kolumnach.

Obsługa zdarzeń generowanych przez pole tekstowe.

Postać okna tego programu przedstawia rysunek 17.6.

W powyższym programie warto zwrócić uwagę na zastosowanie wyrażeń lambda jako obiektów nasłuchujących zdarzeń. Obsługa każdego ze zdarzeń sprowadza się do jednego wywołania metody, dzięki czemu doskonale nadaje się do zaimplementowania przy użyciu wyrażeń lambda.



Rysunek 17.6. *Aplikacja JavaFX z polem tekstowym*

Ekspert odpowiada

Pytanie: Jakie są inne kontrolki tekstowe dostępne w JavaFX?

Odpowiedź: Są to kontrolki: `TextArea` dająca możliwość wpisywania wielu wierszy tekstu oraz `PasswordField` służąca do wpisywania haseł. Czasami może się także przydać kontrolka `HTMLEditor` oraz `TextInputDialog`.

Przedstawienie efektów i transformacji

Główną zaletą platformy JavaFX jest możliwość modyfikowania wyglądu każdej kontrolki (czy też dowolnego węzła w grafie sceny) przy wykorzystaniu **efektów** oraz **transformacji**. Zarówno efekty, jak i transformacje pozwalają tworzyć graficzne interfejsy użytkownika o wyszukanym, nowoczesnym wyglądzie, którego użytkownicy zwykli oczekiwać. Jak się niebawem przekonamy, łatwość stosowania efektów i transformacji jest jednym z najmocniejszych punktów platformy JavaFX. Choć stosowanie efektów i transformacji jest dosyć szerokim zagadnieniem, to jednak ich krótka prezentacja zamieszczona w tym rozdziale pozwoli Ci zorientować się, jak wielkie korzyści zapewnia możliwość ich stosowania.

Efekty

Efekty są obsługiwane przez abstrakcyjną klasę `Effect` oraz jej konkretne klasy potomne, zdefiniowane w pakiecie `javafx.scene.efect`. Korzystając z tych efektów, można modyfikować wygląd węzłów umieszczonych w grafie sceny. Dostępnych jest kilka predefiniowanych efektów, a wybrane z nich przedstawiłem poniżej:

`Bloom`	Zwiększa jasność jasnej części węzła.
`BoxBlur`	Zamazuje węzeł.
`DropShadow`	Wyświetla cień umieszczony poniżej węzła.
`Glow`	Tworzy efekt rozświetlenia.
`InnerShadow`	Wyświetla cień wewnątrz węzła.
`Lighting`	Tworzy efekt cienia wywoływanego przez źródło światła.
`Reflection`	Wyświetla odbicie.

Zarówno te, jak i wszystkie pozostałe efekty są łatwe w użyciu i można je stosować we wszystkich węzłach, czyli obiektach klasy `Node` i klas pochodnych — w tym także w kontrolkach. Oczywiście w zależności od kontrolki niektóre z tych efektów będą bardziej odpowiednie, a inne mniej.

Aby zastosować efekt w węźle, należy wywołać metodę `setEffect()`, zdefiniowaną w klasie `Node`:

```
final void setEffect(Effect effect)
```

W tym przypadku parametr *`effect`* określa efekt, który należy zastosować. Aby nie stosować żadnego efektu, w wywołaniu metody `setEffect()` należy przekazać wartość `null`. Aby więc użyć jakiegoś efektu, w pierwszej kolejności należy utworzyć jego instancję, a następnie przekazać ją w wywołaniu metody `setEffect()`. Kiedy to zrobimy, dany efekt będzie używany zawsze podczas wyświetlania danego węzła (o ile tylko efekt będzie obsługiwany przez środowisko uruchomieniowe). Aby zademonstrować możliwości efektów, zastosujemy dwa z nich: `Reflection` oraz `BoxBlur`. Niemniej jednak proces dodawania efektów jest w zasadzie taki sam, niezależnie od tego, który z nich wybierzemy.

Efekt `BoxBlur` zamazuje węzeł, w którym został zastosowany. Nosi on tę nazwę, gdyż korzysta z techniki zamazywania (ang. *blurring*), bazującej na modyfikowaniu pikseli w pewnym prostokątnym (ang. *box*) obszarze. Aby skorzystać z tego efektu, w pierwszej kolejności trzeba utworzyć instancję klasy `BoxBlur`. Definiuje ona dwa konstruktory. Poniżej przedstawiłem ten, którego użyjemy:

```
BoxBlur(double width, double height, int iterations)
```

Parametry *width* oraz *height* określają wymiary obszaru, wewnątrz którego zostaną rozprzestrzenione informacje o kolorze danego piksela. Parametry te mogą przyjmować wartości z zakresu od 0 do 255. Zazwyczaj jednak wartości te są bliskie dolnej granicy tego zakresu.

Ostatni parametr, *iterations*, określa, ile razy zostanie zastosowany efekt. Jego wartość musi się mieścić w zakresie od 0 do 3 włącznie. Klasa BoxBlur udostępnia także konstruktor domyślny, w którym pierwsze dwa parametry przyjmują wartość 5.0, a trzeci — wartość 1.

Po utworzeniu instancji klasy BoxBlur szerokość i wysokość efektu można zmieniać przy użyciu metod setWidth() oraz setHeight():

```
final void setWidth(double width)

final void setHeight(double height)
```

Z kolei metoda setIterations() pozwala określić, ile razy efekt ma zostać zastosowany:

```
final void setIterations(int iterations)
```

Posługując się tymi metodami, można zmieniać postać efektu w trakcie działania programu.

Z kolei efekt Reflection symuluje efekt odbicia węzła, w którym został zastosowany. Jest on szczególnie użyteczny w przypadku węzłów zawierających teksty, na przykład w etykiecie. Klasa Reflection zapewnia bardzo duże możliwości kontroli wyglądu generowanego efektu. Na przykład można określać poziom nieprzezroczystości zarówno początkowego, jak i końcowego obszaru odbicia. Można także określać odstęp pomiędzy obrazem i odbiciem, jak również wielkość odbijanego obszaru. Wszystkie te informacje można podać w konstruktorze klasy Reflection:

```
Reflection(double offset, double fraction, double topOpacity, dobule bottomOpacity)
```

Parametr *offset* określa odstęp pomiędzy dolną krawędzią obrazu oraz jego odbiciem. Wielkość wyświetlanego odbicia jest określana przez parametr *fraction*. Jego wartość musi być liczbą z zakresu od 0 do 1.0. Parametry *topOpacity* oraz *bottomOpacity* określają odpowiednio nieprzezroczystość górnej oraz dolnej części odbicia. Wartości obu tych parametrów muszą się mieścić w zakresie od 0 do 1.0. Klasa Reflection udostępnia także konstruktor domyślny, który przypisuje parametrowi offset wartość 0, parametrowi fraction wartość 0.75, parametrowi topOpacity wartość 0.5, a parametrowi bottomOpacity wartość 0.

Wszystkie te wartości można także zmieniać podczas działania programu. Na przykład do modyfikacji poziomów nieprzezroczystości służą przedstawione poniżej metody setTopOpacity() oraz setBottomOpacity():

```
final void setTopOpacity(double opacity)

final void setBottomOpacity(double opacity)
```

Odstęp pomiędzy obrazem i odbiciem można określać przy użyciu metody setTopOffset():

```
final void setTopOffset(double offset)
```

Z kolei metoda setFraction() pozwala określać wielkość odbicia:

```
final void setFraction(double amount)
```

Te cztery metody pozwalają na modyfikowanie efektu odbicia podczas działania programu.

Transformacje

Transformacje są obsługiwane przez abstrakcyjną klasę Transform, zdefiniowaną w pakiecie javafx.
↪scene.transform. Ma ona cztery klasy pochodne: Rotate, Scale, Shear oraz Translate. Każda z nich realizuje przekształcenie odpowiadające jej nazwie, czyli: obrót, zmianę wielkości, pochylenie oraz przesunięcie. (Istnieje jeszcze jedna klasa pochodna klasy Transform, Affine, jednak jest ona stosowana znacznie rzadziej od czterech wymienionych wcześniej). W węźle można zastosować więcej niż jedną transformację. Na przykład można go obrócić i przeskalować. Jak wyjaśnię już zaraz w tekście zamieszczonym poniżej, transformacje są obsługiwane przez klasę Node.

Jednym ze sposobów dodawania transformacji do węzła jest dodanie jej do listy używanych w nim transformacji. Listę tę można pobrać, wywołując metodę `getTransform()` zdefiniowaną w klasie `Node`. Poniżej przedstawiłem jej deklarację:

```
final ObservableList<Transform> getTransforms()
```

Wywołanie tej metody zwraca listę transformacji. Aby dodać do niej kolejną transformację, wystarczy skorzystać z metody `add()`. Można także wyczyścić tę listę, wywołując metodę `clear()`. Istnieje także możliwość usunięcia konkretnego elementu listy; służy do tego metoda `remove()`.

W niektórych sytuacjach transformację można określić bezpośrednio, korzystając z jednej z właściwości klasy `Node`. Na przykład wywołując metodę `setRotate()` (i przekazując do niej wielkość kąta), można określić kąt obrotu węzła, przy czym oś obrotu będzie położona w środku danego węzła. Aby przeskalować węzeł, można skorzystać z metod `setScaleX()` oraz `setScaleY()`; podobnie przesunięcie węzła można określić przy użyciu metod `setTranslateX()` oraz `setTranslateY()`. (Dane środowisko uruchomieniowe może także obsługiwać przekształcenia wzdłuż osi *Z*). Niemniej jednak korzystanie z listy transformacji zapewnia większą elastyczność i dlatego to właśnie ten sposób modyfikowania węzłów przedstawię w tym rozdziale.

W ramach prezentacji użycia transformacji zastosujemy dwie klasy: `Rotate` i `Scale`. (Pozostałe klasy są stosowane w taki sam ogólny sposób). Klasa `Rotate` pozwala obrócić węzeł o zadany kąt względem wskazanego punktu obrotu. Obie te wartości można określać podczas tworzenia instancji klasy. Poniżej przedstawiłem postać jednego z konstruktorów tej klasy:

```
Rotate(double angle, double x, double y)
```

Parametr *angle* określa kąt obrotu wyrażony w stopniach. Punkt, względem którego węzeł będzie obracany, nazywany także **punktem obrotu**, jest określany przez parametry *x* i *y*.

Można także skorzystać z konstruktora domyślnego i określić parametry obrotu po utworzeniu obiektu klasy `Rotate`; właśnie takie rozwiązanie zostało zastosowane w programie przedstawionym na listingu 17.7. Informacje określające postać transformacji można podać przy użyciu metod: `setAngle()`, `setPivotX()` oraz `setPivotY()`; poniżej przedstawiłem postać ich deklaracji:

```
final void setAngle(double angle)

final void setPivotX(double x)

final void setPivotY(double y)
```

Podobnie jak wcześniej w konstruktorze, także w tym przypadku parametr *angle* określa kąt obrotu wyrażony w stopniach, a parametry *x* i *y* podają współrzędne punktu stanowiącego oś obrotu. Używając tych metod, można określić lub zmienić parametry obrotu podczas działania programu. Pozwalają one uzyskiwać naprawdę spektakularne efekty.

Klasa `Scale` pozwala przeskalować węzeł o zadany współczynnik. Oznacza to, że za jej pomocą możemy zmieniać wielkość węzła. Klasa `Scale` definiuje kilka konstruktorów; poniżej przedstawiłem ten, którego użyjemy w przykładowym programie.

```
Scale(double widthFactor, double heightFactor)
```

Parametr *widthFactor* określa współczynnik przeskalowania szerokości węzła, natomiast parametr *heightFactor* — współczynnik przeskalowania jego wysokości. Oba te współczynniki można zmieniać także po utworzeniu instancji klasy `Scale`; służą do tego odpowiednio przedstawione poniżej metody `setX()` oraz `setY()`:

```
final void setX(double widthFactor)

final void setY(double heightFactor)
```

Podobnie jak w przypadku konstruktora, także w tych metodach parametr *widthFactor* określa współczynnik przeskalowania szerokości węzła, a parametr *heightFactor* współczynnik przeskalowania wysokości węzła. Tych dwóch metod można używać w celu modyfikowania wielkości węzła w trakcie działania programu, na przykład w celu zwrócenia na dany węzeł uwagi użytkownika.

Prezentacja zastosowania efektów i transformacji

Program zamieszczony poniżej na listingu 17.7 przedstawia zastosowanie efektów oraz transformacji. W tym celu tworzy trzy przyciski oraz etykietę. Przyciski noszą odpowiednio nazwy: *Obróć*, *Przeskaluj* oraz *Zamaż*. Każde kliknięcie jednego z tych przycisków powoduje zastosowanie w nim odpowiedniego efektu lub transformacji. Konkretnie rzecz biorąc, każde kliknięcie przycisku *Obróć* sprawi, że zostanie on obrócony o 15 stopni. Podobnie każde kliknięcie przycisku *Przeskaluj* spowoduje zmianę jego wielkości, a kliknięcie przycisku *Zamaż* — jego coraz większe zamazanie. Z kolei etykieta prezentuje postać efektu odbicia. Testując ten program, przekonasz się, jak łatwo można zmodyfikować wygląd interfejsu użytkownika aplikacji. Bardzo interesujące może być przeprowadzenie kilku eksperymentów i wypróbowanie różnych transformacji i efektów; możesz także sprawdzić, jak wyglądają te przekształcenia na węzłach innych typów niż przyciski.

Listing 17.7. *EffectsAndTransformsDemo.java*

```
// Prezentacja obrotu, skalowania, odbicia i zamazania.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {

  double angle = 0.0;
  double scaleFactor = 0.4;
  double blurVal = 1.0;

  // Tworzy i inicjuje efekty i transformacje.
  Reflection reflection = new Reflection();
  BoxBlur blur = new BoxBlur(1.0, 1.0, 1);
  Rotate rotate = new Rotate();
  Scale scale = new Scale(scaleFactor, scaleFactor);

  // Tworzy przyciski.
  Button btnRotate = new Button("Obróć");
  Button btnBlur = new Button("Zamaż");
  Button btnScale = new Button("Przeskaluj");

  Label reflect = new Label("Odbicie jest zachwycające");

  public static void main(String[] args) {

    // Uruchamia aplikację JavaFX, wywołując metodę launch().
    launch(args);
  }

  // Przesłonięcie metody start().
  public void start(Stage myStage) {

    // Określa tytuł obszaru roboczego.
    myStage.setTitle("Prezentacja efektów i transformacji");

    // Jako korzeń zostaje użyty panel FlowPane. W tym
    // przypadku pionowe i poziome odstępy pomiędzy umieszczonymi
    // w nim kontrolkami będą wynosić 10.
```

Utworzenie efektów i transformacji.

```
FlowPane rootNode = new FlowPane(20, 20);

// Wyrównuje kontrolki do środka.
rootNode.setAlignment(Pos.CENTER);

// Tworzy obiekt Scene.
Scene myScene = new Scene(rootNode, 300, 120);

// Dodaje obiekt Scene do obiektu Stage.
myStage.setScene(myScene);

// Dodaje obrót do listy transformacji przycisku Obrót.
btnRotate.getTransforms().add(rotate);
```

Dodanie obrotu do przycisku `btnRotate`.

```
// Dodaje skalowanie do listy transformacji przycisku Skalowanie.
btnScale.getTransforms().add(scale);
```

Dodanie skalowania do przycisku `btnScale`.

```
// Dodaje odbicie do listy efektów etykiety.
reflection.setTopOpacity(0.7);
reflection.setBottomOpacity(0.3);
reflect.setEffect(reflection);
```

Dodanie odbicia do etykiety `reflect`.

```
// Obsługa zdarzeń przycisku Obrót.
btnRotate.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Każde kliknięcie przycisku powoduje powiększenie kąta
    // obrotu o 15 stopni.
    angle += 15.0;

    rotate.setAngle(angle);
    rotate.setPivotX(btnRotate.getWidth()/2);
    rotate.setPivotY(btnRotate.getHeight()/2);
  }
});

// Obsługa zdarzeń przycisku Skalowanie.
btnScale.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Każde kliknięcie przycisku powoduje powiększenie
    // współczynnika przeskalowania.
    scaleFactor += 0.1;
    if(scaleFactor > 2.0) scaleFactor = 0.4;

    scale.setX(scaleFactor);
    scale.setY(scaleFactor);

  }
});

// Obsługa zdarzeń przycisku Zamazanie.
btnBlur.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Każde kliknięcie przycisku powoduje zmianę stopnia zamazania.
    if(blurVal == 10.0) {
      blurVal = 1.0;
      btnBlur.setEffect(null);
```

Usunięcie efektu zamazania z przycisku `btnBlur`.

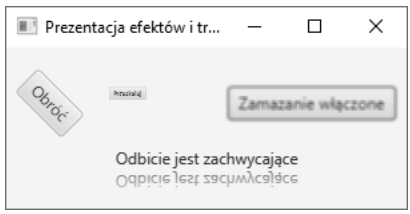
```
      btnBlur.setText("Zamazanie wyłączone");
    } else {
      blurVal++;
      btnBlur.setEffect(blur);
```

Dodanie efektu zamazania do przycisku `btnBlur`.

```
      btnBlur.setText("Zamazanie włączone");
    }
    blur.setWidth(blurVal);
    blur.setHeight(blurVal);
```

```
        }
    });

    // Dodaje etykietę i kontrolkę listy do grafu sceny.
    rootNode.getChildren().addAll(btnRotate, btnScale, btnBlur, reflect);

    // Wyświetla scenę i obszar roboczy.
    myStage.show();
  }
}
```

Przykładową postać okna tej aplikacji przedstawia rysunek 17.7.



Rysunek 17.7. *Prezentacja efektów i transformacji*

Zanim zakończymy temat efektów i transformacji, warto zaznaczyć, że kilka z nich daje szczególnie atrakcyjne efekty wizualne w przypadku stosowania na węzłach klasy `Text`. Klasa ta została zdefiniowana w pakiecie `javafx.scene.text`. Jej obiekty tworzą węzły zawierające tekst. A ponieważ są to węzły, oznacza to, że tekst można traktować jako element interfejsu użytkownika i stosować w nim efekty i transformacje.

Co dalej?

Gratuluję! Jeśli przeczytałeś wszystkie 17 rozdziałów tej książki, przeanalizowałeś przykłady i wykonałeś testy sprawdzające, to możesz się nazwać programistą używającym Javy. Oczywiście wciąż pozostaje jeszcze bardzo wiele rzeczy, których musisz się dowiedzieć na temat tego języka, jego bibliotek i podsystemów, jednak teraz dysponujesz już solidnym fundamentem, na którym możesz oprzeć swoją wiedzę i doświadczenia.

Poniżej podałem listę paru zagadnień, które być może będziesz chciał dokładniej poznać:

- JavaFX oraz Swing — obie technologie są ważnymi narzędziami stosowanymi podczas tworzenia nowoczesnych aplikacji pisanych w Javie.
- Obsługa zdarzeń.
- Klasy do obsługi komunikacji sieciowej.
- Klasy narzędziowe języka Java, a w szczególności biblioteka kolekcji — Collections Framework — która znacząco upraszcza wiele powszechnie wykonywanych zadań programistycznych.
- Interfejs API do obsługi współbieżności, zapewniający dokładną kontrolę nad wydajnymi aplikacjami wielowątkowymi.
- Java Beans — technologia umożliwiająca tworzenie komponentów programowych w języku Java.
- Serwlety. Jeśli masz zamiar pisać wydajne aplikacje internetowe, to na pewno będziesz chciał poznać serwlety.

Jeśli chcesz kontynuować naukę języka Java, polecam książkę *Java. Kompendium programisty*, wydaną przez Wydawnictwo Helion. Znajdziesz w niej wyczerpujące opracowanie samego języka Java oraz jego bibliotek, jak również wiele przykładów.

Test sprawdzający

1. Jaki jest główny pakiet platformy JavaFX?
2. Dwoma podstawowymi pojęciami stosowanymi w JavaFX są: obszar roboczy oraz scena. Jakie klasy stanowią implementację tych pojęć?
3. Graf sceny składa się z _____?
4. Klasą bazową węzłów jest _____?
5. Po której klasie dziedziczą wszystkie aplikacje JavaFX?
6. Jakie są trzy metody cyklu życia aplikacji JavaFX?
7. W której z tych trzech metod można tworzyć obiekt obszaru roboczego i elementy interfejsu użytkownika?
8. W celu uruchomienia niezależnych aplikacji JavaFX należy wywołać metodę `launch()`. To prawda czy fałsz?
9. Jak nazywają się klasy JavaFX służące do tworzenia etykiet i przycisków?
10. Jednym ze sposobów zakończenia niezależnej aplikacji JavaFX jest wywołanie metody `Platform.exit()`. Klasa `Platform` należy do pakietu `javafx.Application`. Wywołanie metody `exit()` powoduje natychmiastowe zakończenie aplikacji. Pamiętając o tym, zmodyfikuj przedstawiony w tym rozdziale program `JavaFXEventDemo` w taki sposób, by wyświetlał dwa przyciski: *Uruchom* i *Zakończ*. Kliknięcie pierwszego z nich powinno powodować wyświetlenie w etykiecie odpowiedniego komunikatu, a kliknięcie drugiego — zakończenie aplikacji. Zdarzenia generowane przez oba przyciski obsługuj przy użyciu wyrażeń lambda.
11. Jak nazywa się kontrolka JavaFX służąca do tworzenia przycisków opcji?
12. `ListView` to kontrolka służąca do wyświetlania listy plików i katalogów na lokalnym dysku twardym. Czy to prawda, czy fałsz?
13. Zmodyfikuj program przedstawiony w przykładzie 16.1 w taki sposób, by zamiast z pakietu Swing korzystał z JavaFX. Wykorzystaj w nim kolejną możliwość platformy JavaFX: programowe zgłaszanie zdarzeń kierowanych do przycisku. Można to zrobić, wywołując metodę `fire()` na rzecz instancji przycisku. Na przykład przy założeniu, że dysponujemy obiektem klasy `Button` o nazwie `myButton`, wywołanie `myButton.fire()` wygeneruje zdarzenie `ActionEvent` skierowane do tego przycisku. Skorzystaj z tej możliwości, implementując obsługę zdarzeń dwóch pól tekstowych zawierających nazwy porównywanych plików. Kiedy użytkownik naciśnie w którymś z nich klawisz *Enter*, zgłoś zdarzenie `ActionEvent` skierowane do przycisku *Porównaj*. Dzięki temu samo porównanie będzie mogło być wykonane przez kod obsługujący kliknięcia przycisku *Porównaj*.
14. Zmodyfikuj program `EffectsAndTransformsDemo` w taki sposób, by także przycisk *Obróć* był zamazany. Szerokość i wysokość zamazywanego obszaru ma wynosić 5, a liczba powtórzeń — 2.
15. Spróbuj samodzielnie poeksperymentować z różnymi efektami i transformacjami. Na przykład wypróbuj efekt `Glow` i transformację `Translate`.
16. Kontynuuj poznawanie języka Java. Dobrym punktem wyjścia będzie poznanie podstawowych pakietów biblioteki Javy, takich jak `java.lang`, `java.util` oraz `java.net`. Pisz przykładowe programy demonstrujące możliwości i wykorzystanie różnych klas i interfejsów tych pakietów. Ogólnie rzecz biorąc, najlepszym sposobem, by zostać doskonałym programistą używającym Javy, jest pisanie jak największej ilości kodu.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

Notatki

Java: zdobądź solidne podstawy i twórz kod mistrzów!

Java jest jednym z kilku języków programowania, które niemal od początku istotnie wpływały na kształt programowania jako takiego. Już w początkowej fazie swojej historii, w 1995 roku, Java spowodowała prawdziwą rewolucję w programowaniu. Stała się jedną z przyczyn skokowego rozwoju technologii internetowych i określiła nowe standardy projektowania języków programowania. Do dzisiaj zaliczana jest do awangardy — i skupia wokół siebie społeczność osób ciągle poszukujących innowacji. Oznacza to, że solidne podstawy programowania w Javie są znakomitą inwestycją dla każdego programisty, który swoją przyszłość wiąże z rozwojem najnowszych technologii informatycznych.

Ta książka jest kolejnym wydaniem wyjątkowego podręcznika, zaktualizowanym o informacje dotyczące Javy SE 9. Dzięki lekturze zdobędziesz solidne podstawy programowania w Javie, nawet jeśli nie masz żadnego przygotowania w tym kierunku. Poszczególne zagadnienia przedstawiono tu bardzo klarownie i przejrzyście, krok po kroku, uzupełniając je o liczne przykłady, testy sprawdzające i projekty do samodzielnej pracy. Najpierw zapoznasz się z podstawowymi informacjami, takimi jak kompilacja i uruchomienie programu w Javie. Następnie nauczysz się stosować słowa kluczowe i konstrukcje, które tworzą rdzeń tego języka. Stopniowo będziesz przechodzić do zaawansowanych tematów, włączając w to programowanie wielowątkowe, typy sparametryzowane, wyrażenia lambda oraz moduły. Na końcu zapoznasz się z biblioteką Swing.

Najważniejsze zagadnienia:

- **historia Javy, jej cechy szczególne i przygotowanie środowiska pracy**
- **podstawowe elementy kodu i zasady programowania obiektowego**
- **dziedziczenie, pakiety i interfejsy**
- **praca na plikach i operacje wejścia-wyjścia**
- **moduły i usługi**
- **technologia JavaFX**

Herbert Schildt — jest autorem niezwykle popularnych książek o programowaniu, które wielokrotnie stawały się prawdziwymi bestsellerami. To także uznany autorytet w dziedzinie nauczania takich języków jak Java, C++, C i C#. Zgłębianie ich tajników jest jego pasją od kilkudziesięciu lat. W latach 90. zeszłego stulecia brał udział w tworzeniu standardów języka C++ w ramach prac komitetu ANSI/ISO.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-4611-6
9 788328 346116

Oracle Press

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł