Wykład XI

Java - środowisko GUI
Biblioteka JavaFX

Wymagania dotyczące aplikacji konsumenckich, a zwłaszcza aplikacji na urządzenia mobilne; wiążą się z bardzo dużą atrakcyjnością wizualną i funkcjonalnością interfejsu użytkownika . Do uzyskanie lepszego wsparcia i uniwersalności tworzenie graficznych interfejsów użytkownika, konieczne było zastosowanie nowego rozwiązania i to właśnie ta potrzeba doprowadziła do powstania JavaFX. JavaFX to platforma kliencka oraz pakiet do tworzenia graficznego interfejsu użytkownika następnej generacji.
JavaFX udostępnia bardzo potężną, łatwą w użyciu i elastyczną platformę ułatwiającą tworzenie nowoczesnych i bardzo atrakcyjnych wizualnie interfejsów użytkownika.
W tym wykładzie celem będzie przedstawienie możliwości nowego pakietu oraz środowiska graficznego do tworzenia oprogramowania z wykorzystaniem platformy Java-FX.
JavaFX stanowi docelową platformę, która w przyszłości będzie używana do tworzenia graficznego interfejsu użytkownika aplikacji pisanych w Javie. Oczekuje się, że w ciągu kilku najbliższych lat JavaFX zastąpi Swing w nowych projektach.
Rozwój platformy JavaFX miał dwie główne fazy. Początkowo JavaFX bazowała na języku skryptowym o nazwie **JavaFX Script**. Ale przestał on być rozwijany. Zaczynając od wersji JavaFX 2.0, platforma JavaFX jest tworzona bezpośrednio w języku Java i udostępnia rozbudowane API. JavaFX obsługuje także język FXML, który może być używany do tworzenia interfejsu użytkownika.

Podstawowe pojęcia z zakresu JavaFX

Platforma JavaFX ma wszystkie dobre cechy pakietu Swing, została ona napisana bezpośrednio w Javie. Oprócz tego zapewnia wsparcie dla architektury MVC.
Przeważająca większość pojęć związanych z tworzeniem graficznego interfejsu użytkownika w pakiecie Swing odnosi się także do JavaFX. Niemniej jednak pomiędzy oboma rozwiązaniami występują znaczące różnice.
Z punktu widzenia programisty pierwszą różnicą pomiędzy JavaFX i pakietem Swing, którą można zauważyć, jest organizacja platformy JavaFX oraz wzajemne powiązania pomiędzy jej głównymi komponentami , JavaFX umożliwia tworzenie interfejsów użytkownika, które są bardziej dynamiczne pod względem wizualnym.

Pakiety JavaFX

Elementy JavaFX zostały umieszczone w pakietach, których nazwy rozpoczynają się od javafx. Biblioteka API platformy JavaFX składała się z ponad 30 pakietów. Oto cztery przykładowe z nich: javafx.application, javafx.stage, javafx.scene oraz javafx.scene.layout.

Klasy Stage oraz Scene

Podstawową elementem zaimplementowanym przez JavaFX jest **obszar roboczy** (ang. *scene*) zawierający **scenę**. Obszar roboczy definiuje przestrzeń, natomiast scena określa, co się w tej przestrzeni znajduje, inaczej: obszar roboczy jest kontenerem dla sceny, natomiast scena — dla elementów, które się na nią składają. W efekcie wszystkie aplikacje JavaFX zawierają przynajmniej jeden obszar roboczy oraz jedną scenę. W JavaFX elementy te są reprezentowane przez klasy Stage oraz Scene. W celu utworzenia aplikacji JavaFX należy w najprostszym przypadku dodać do obiektu Stage przynajmniej

jeden obiekt Scene. Przyjrzyjmy się nieco dokładniej tym dwóm klasom.
Klasa Stage jest pojemnikiem głównego poziomu. Wszystkie aplikacje JavaFX automatycznie mają dostęp do jednego obiektu Stage, nazywanego **głównym obszarem roboczym**. Ten główny obszar roboczy jest dostarczany przez środowisko wykonawcze podczas uruchamiania aplikacji JavaFX. Choć można także tworzyć inne obszary robocze, to w wielu aplikacjach ten główny będzie jedynym niezbędnym.
Zgodnie z podanymi wcześniej informacjami obiekt Scene jest pojemnikiem dla wszystkich elementów tworzących scenę. Elementami tymi mogą być różnego rodzaju kontrolki, takie jak przyciski, pola wyboru, pola tekstowe czy też obrazki. W celu utworzenia sceny wszystkie te elementy należy dodać do obiektu Scene.

Węzły i graf sceny

Poszczególne elementy sceny są nazywane **węzłami** (ang. *node*). Węzłem jest na przykład kontrolka przycisku. Jednak oprócz tego węzłami mogą także być grupy węzłów. Co więcej, węzeł może zawierać węzły potomne. W takim przypadku mówi się, że węzeł zawierający węzły potomne jest ich **rodzicem** lub **węzłem gałęzi** (ang. *branch node*). Z kolei węzły, które nie zawierają węzłów potomnych, są węzłami końcowymi, czyli tak zwanymi **liśćmi.**
W grafie sceny istnieje jeden węzeł o szczególnym znaczeniu, jest to tak zwany **korzeń.** Jest to węzeł głównego poziomu i jako jedyny w całym grafie sceny nie ma rodzica. A zatem z wyjątkiem korzenia wszystkie inne węzły w grafie sceny mają węzeł rodzica i wszystkie, bezpośrednio lub pośrednio, są potomkami korzenia.
Klasą bazową wszystkich węzłów jest klasa Node. Dostępnych jest także kilka innych klas, które bezpośrednio lub pośrednio są podklasami klasy Node. Do tej grupy zaliczają się między innymi klasy: Parent, Group, Region oraz Control.

Układy

JavaFX udostępnia kilka paneli układów, które zarządzają procesem rozmieszczania elementów na scenie. Na przykład panel FlowPane tworzy układ rozmieszczający elementy jeden za drugim, a panel GridPane tworzy układ pozwalający rozmieszczać elementy w wierszach i kolumnach. Dostępnych jest także kilka innych układów, na przykład BorderPane (przypominający menedżera układu BorderLayout znanego z AWT). Wszystkie te panele układów są umieszczone w pakiecie javafx.scene.layout.

Klasa Application i metody cyklu życia

Każda aplikacja JavaFX musi być klasą dziedziczącą po klasie Application, umieszczoną w pakiecie javafx.application. Oznacza to, że klasy tworzonych przez nas aplikacji JavaFX będą rozszerzać klasę Application. Klasa ta definiuje trzy metody cyklu życia, które aplikacje mogą przesłaniać. Są nimi przedstawione poniżej metody: init(), start() oraz stop(). Kolejność ich wywoływania odpowiada kolejności, w jakiej zostały podane na poniższej liście:
void init()
abstract void start(Stage *primaryStage*)
void stop()
Metoda init() jest wywoływana podczas uruchamiania aplikacji. Służy ona do wykonywania różnych czynności związanych z inicjalizacją aplikacji. Klasy tej ***nie można*** używać ani do utworzenia obiektu obszaru roboczego, ani do skonstruowania sceny. Jeśli wykonywanie czynności inicjalizacyjnych nie jest konieczne, to metody tej nie trzeba przesłaniać, gdyż jest dostępna jej domyślna, pusta wersja.
Po metodzie init() wywoływana jest metoda start(). To od niej rozpoczyna się działanie aplikacji i to właśnie *jej* można używać do konstruowania sceny. Warto zauważyć, że do tej metody jest przekazywana referencja do obiektu Stage. Obiekt ten reprezentuje obszar roboczy dostarczony przez środowisko wykonawcze, czyli główny obszar roboczy. Należy zauważyć, że start() jest metodą abstrakcyjną, co oznacza, że aplikacja musi ją przesłonić.
Podczas kończenia działania aplikacji wywoływana jest metoda stop(). Jest ona miejscem, w którym można wykonywać wszelkie czynności porządkowe oraz inne, związane z

zamykaniem aplikacji. Jeśli wykonywanie takich operacji nie jest potrzebne, metody stop() nie trzeba przesłaniać, gdyż jest dostępna jej domyślna, pusta wersja.

Uruchamianie aplikacji JavaFX

Aby uruchomić niezależną aplikację JavaFX, należy wywołać metodę launch() zdefiniowaną w klasie Application. Metoda ta ma dwie wersje; poniżej przedstawiona została ta, która będzie używana w tym rozdziale:
public static void launch(String ... *args*)
Parametr *args* reprezentuje listę łańcuchów, które zazwyczaj są argumentami podanymi w wierszu poleceń. Warto zwrócić uwagę, że lista ta może być pusta. W momencie wywołania metoda launch() powoduje utworzenie aplikacji oraz wywołanie najpierw metody init(), a następnie metody start(). Metoda launch() nie zwraca sterowania aż do momentu zakończenia aplikacji. Przedstawiona wcześniej wersja metody launch() uruchamia aplikację, czyli podklasę klasy Application, wewnątrz której metoda ta została wywołana. Druga wersja metody launch() pozwala określić inną klasę aplikacji, którą należy uruchomić.
Aplikacje JavaFX, które zostały spakowane przy użyciu programu narzędziowego *javafxpackager* (bądź analogicznych narzędzi udostępnianych przez zintegrowane środowiska programistyczne), nie muszą zawierać wywołania metody launch(). Niemniej jednak umieszczenie jawnego wywołania tej metody w kodzie aplikacji niejednokrotnie ułatwia proces jej testowania i uruchamiania, a oprócz tego zapewnia możliwość korzystania z programu bez umieszczania go w pliku JAR.

Szkielet aplikacji JavaFX

Wszystkie aplikacje JavaFX mają podobną podstawową strukturę. Dlatego zanim zajmiemy się innymi możliwościami platformy JavaFX, warto przedstawić bardzo prosty szkielet takiej aplikacji.
Oprócz przedstawienia podstawowej struktury aplikacji zamieszczony poniżej kod pokazuje także, w jaki sposób można ją uruchomić oraz jak wywoływać metody cyklu życia aplikacji. Komunikaty informujące o wywołaniach metod cyklu życia aplikacji są wyświetlane w oknie konsoli.
Przykładowy szkielet aplikacji JavaFX:

```
package application;

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


public class Okno_JavaFX extends Application {

	//Przesłonięta metoda init().
	public void init() {
	System.out.println("W metodzie init().");
	}
	//Przesłonięta metoda start().
	public void start(Stage myStage) {
	System.out.println("W metodzie start().");
	//Określa nazwę obszaru roboczego.
	myStage.setTitle("Szkielet aplikacji JavaFX");
	//Tworzy korzeń grafu sceny. W tym przypadku zostaje
	//użyty panel FlowPane, lecz dostępnych jest także
	//kilka alternatywnych paneli.
	FlowPane rootNode = new FlowPane();
	//Tworzy obiekt sceny.
	Scene myScene = new Scene(rootNode, 300, 200);
```

```
        //Zapisuje scenę w obszarze roboczym.
        myStage.setScene(myScene);
        //Wyświetla obszar roboczy i scenę.
        myStage.show();
        }
        //Przesłonięta wersja metody stop().
        public void stop() {
        System.out.println("W metodzie stop().");
        }

        public static void main(String[] args) {
              System.out.println("Uruchamianie aplikacji JavaFX.");
              //Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
              launch(args);
        }

}
```

Wątek aplikacji

Metoda init() nie może być używana do tworzenia obszaru roboczego ani sceny. Nie można ich także tworzyć w konstruktorze aplikacji. Wynika to z faktu, że obiekty te muszą być utworzone w **wątku aplikacji**. Okazuje się jednak, że zarówno konstruktor aplikacji, jak i metoda init() są wykonywane w **wątku uruchomieniowym**. Oznacza to, że nie można ich używać do tworzenia obiektów obszaru roboczego ani sceny. Zamiast tego, jak pokazano w przedstawionym wcześniej szkielecie aplikacji JavaFX, cały graficzny interfejs użytkownika aplikacji musi zostać skonstruowany i zainicjowany w metodzie start(), gdyż ona jest wywoływana w **wątku aplikacji**.
Wprowadzanie zmiana do interfejsu użytkownika musi być wykonywane w wątku aplikacji. Zdarzenia realizowane w JavaFX są przekazywane do aplikacji w wątku aplikacji. Dzięki temu obiekty nasłuchujące mogą bez przeszkód manipulować graficznym interfejsem użytkownika aplikacji. Także metoda stop() jest wywoływana w wątku aplikacji.

Kontrolki JavaFX:

Etykieta Label

Podstawowymi elementami wszystkich graficznych interfejsów użytkownika są kontrolki, gdyż to właśnie one zapewniają użytkownikom możliwość interakcji z aplikacjami. JavaFX udostępnia bogaty zbiór kontrolek. Jedną z najprostszych z nich jest etykieta.
Na przykładzie kontrolki etykiety przedstawiony zostanie proces tworzenia układu sceny W JavaFX kontrolka etykiety jest obiektem klasy Label, która należy do pakietu javafx.scene.control. Klasa ta dziedziczy po klasie Labeled, która z kolei dziedziczy między innymi po klasie Control. Klasa Labeled definiuje kilka cech wspólnych dla wszystkich elementów, które mogą zawierać tekst; z kolei klasa Control definiuje cechy wspólne dla wszystkich kontrolek.
Klasa Label definiuje trzy konstruktory; użyty zostanie następujący konstruktor:
Label(String *str*)
Parametr *str* określa łańcuch, który zostanie wyświetlony w etykiecie. Po utworzeniu etykiety (jak również dowolnej innej kontrolki) należy ją dodać do zawartości sceny, co jednocześnie oznacza dodanie jej do grafu sceny. W tym celu należy skorzystać z węzła korzenia używanego przez obiekt sceny i wywołać jego metodę getChildren(). Metoda ta zwraca listę węzłów potomnych, reprezentowaną w formie obiektu typu ObservableList<Node>. Interfejs ObservableList należy do pakietu javafx.collections i dziedziczy po interfejsie java.util.List, co oznacza, że udostępnia wszystkie możliwości list znanych z frameworku Collections. Używając listy węzłów potomnych zwróconej przez metodę getChildren(), można dodać do niej etykietę — w tym celu wystarczy wywołać metodę add() i przekazać do niej referencję do etykiety.

Aby usunąć kontrolkę z grafu sceny, należy wywołać metodę remove() interfejsu ObservableList. Oto przykład takiego wywołania:
rootNode.getChildren().remove(myLabel);
Powyższa instrukcja usuwa ze sceny komponent myLabel.
Poniższy program przedstawia przykład praktycznego zastosowania tych informacji, tworząc prostą aplikację JavaFX, która wyświetla etykietę:

```
package application;

// Prezentacja komponentu JavaFX - Label.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
public class Kontrolka_FX extends Application {

      public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja etykiet JavaFX");
// Tworzy panel FlowPane, który stanie się węzłem korzenia.
FlowPane rootNode = new FlowPane();
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 300, 200);
// Zapisuje scenę w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę.
Label myLabel = new Label("To jest etykieta JavaFX.");
// Dodaje etykietę do grafu sceny.
rootNode.getChildren().add(myLabel);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Kontrolki przycisków i obsługa zdarzeń

W poprzednim programie nie pokazano obsługi zdarzeń. Większość kontrolek graficznego interfejsu użytkownika generuje zdarzenia, które są obsługiwane przez kod aplikacji. Na przykład przyciski, pola wyboru oraz listy to jedne z wielu kontrolek, które w przypadku użycia generują zdarzenia. Obsługa zdarzeń w aplikacjach JavaFX przypomina obsługę zdarzeń w aplikacjach korzystających z pakietów Swing lub AWT, a nawet jest nieco łatwiejsza.

Jedną z najczęściej używanych kontrolek są przyciski. Wynika z tego, że zdarzenia generowane przez przyciski także są jednymi z najczęściej obsługiwanych.

Właśnie dlatego prezentacja przycisków oraz obsługa zdarzeń zostały połączone i opisane wspólnie.

Podstawowe informacje o zdarzeniach

W JavaFX klasą bazową wszystkich zdarzeń jest klasa Event, umieszczona w pakiecie javafx.event.

Klasa ta dziedziczy po java.util.EventObject, co oznacza, że wszystkie zdarzenia JavaFX dysponują tymi samymi podstawowymi możliwościami co pozostałe zdarzenia w języku Java. Dostępnych jest kilka klas potomnych klasy Event. Jedną z nich, a jednocześnie tą, która zostanie przedstawiona w kolejnym przykładzie, jest klasa ActionEvent. Obsługuje ona zdarzenia generowane przez przyciski.

Ogólnie rzecz ujmując, platforma JavaFX obsługuje zdarzenia, korzystając z modelu delegacji.

A zatem aby obsłużyć zdarzenie, w pierwszej kolejności należy zarejestrować obiekt nasłuchujący, który będzie na nie oczekiwał. W momencie wystąpienia zdarzenia zostanie wywołany odpowiedni obiekt nasłuchujący. Musi on obsłużyć zdarzenie, a następnie oddać sterowanie. Jak zatem widać, zdarzenia JavaFX są obsługiwane bardzo podobnie do zdarzeń stosowanych w pakiecie Swing.

Zdarzenia są obsługiwane poprzez zaimplementowanie interfejsu EventHandler, zdefiniowanego w pakiecie javafx.event. Jest to interfejs sparametryzowany mający następującą deklarację:

interface EventHandler<T extends Event>

Parametr typu T określa typ zdarzeń, które będą obsługiwane. Interfejs ten definiuje tylko jedną metodę, handle(), której parametrem jest obiekt zdarzenia. Postać tej metody została przedstawiona poniżej:

void handle(T *eventObj*)

Parametr *eventObj* reprezentuje wygenerowane zdarzenie. Obiekty nasłuchujące są zazwyczaj implementowane przy użyciu anonimowych klas wewnętrznych, choć można w tym celu używać także niezależnych klas, o ile tylko takie rozwiązanie będzie uzasadnione (na przykład jeśli ten sam obiekt nasłuchujący będzie obsługiwał zdarzenia generowane przez kilka źródeł).

Istnieje możliwość określenia źródła, które wygenerowało zdarzenie. W szczególności dotyczy to tych sytuacji, gdy ten sam obiekt nasłuchujący jest używany do obsługi zdarzeń pochodzących z kilku różnych źródeł. Źródło zdarzenia można określić, wywołując przedstawioną poniżej metodę getSource(), dziedziczoną po klasie java.util.EventObject:

Object getSource()

Inne metody klasy Event powalają określić typ zdarzenia, sprawdzić, czy zdarzenie zostało przetworzone, przetworzyć zdarzenie, wygenerować zdarzenie oraz określić jego element docelowy.

Kiedy zdarzenie zostanie przetworzone, nie jest już ono przekazywane do nadrzędnego obiektu nasłuchującego.

W platformie JavaFX zdarzenia są przetwarzane przy wykorzystaniu **łańcucha przydzielania zdarzeń**. Po wygenerowaniu zdarzenia jest ono przekazywane do węzła korzenia, stanowiącego początek tego łańcucha. Następnie zdarzenie jest przekazywane w

dół łańcucha, aż do elementu docelowego. Kiedy węzeł docelowy obsłuży zdarzenie, jest ono przekazywane z powrotem aż na początek łańcucha, dzięki czemu, jeśli pojawi się taka konieczność, będzie ono mogło zostać obsłużone przez węzły przodków elementu docelowego. Rozwiązanie to nosi nazwę **propagacji zdarzeń**. Jakiś węzeł należący do łańcucha może przetworzyć zdarzenie, co sprawi, że nie będzie już ono dalej przekazywane.

Kontrolka - Button

W platformie JavaFX kontrolka przycisku jest reprezentowana przez klasę Button, zdefiniowaną w pakiecie javafx.scene.control. Klasa Button dziedziczy po stosunkowo wielu klasach bazowych, do których należą między innymi ButtonBase, Labeled, Region, Control oraz Node.
Przyciski JavaFX mogą zawierać tekst, grafikę bądź oba te elementy.
Klasa Button definiuje trzy konstruktory. Podstawowy, ma następującą postać:
Button(String *str*)
W tym przypadku *str* jest łańcuchem określającym komunikat wyświetlany na przycisku. Kliknięcie przycisku powoduje wygenerowanie zdarzenia ActionEvent. ActionEvent jest klasą zdefiniowaną w pakiecie javafx.event. Obiekt nasłuchujący obsługujący zdarzenia tego typu można zarejestrować, wywołując metodę setOnAction(), której ogólna postać została przedstawiona poniżej:
final void setOnAction(EventHandler<ActionEvent> *handler*)
Parametr *handler* jest rejestrowanym obiektem nasłuchującym. Jak już wspomniano, obiektami nasłuchującymi są przeważnie anonimowe klasy wewnętrzne lub wyrażenia lambda. Metoda setOnAction() ustawia wartość właściwości onAction, która przechowuje referencję do obiektu nasłuchującego.
Podobnie jak w przypadku wcześniejszych technologii związanych z obsługą zdarzeń w języku Java, także i teraz obiekty nasłuchujące powinny obsługiwać zdarzenia tak szybko, jak to tylko możliwe, a następnie oddawać sterowanie. Jeśli obsługa zdarzeń przez obiekt nasłuchujący zajmuje zbyt wiele czasu, może to doprowadzić do zauważalnego spadku wydajności działania aplikacji. A zatem w przypadku długotrwałych operacji należy je realizować w odrębnych wątkach.
Przykład zastosowania przycisku i obsługi zdarzeń:

```
package application;
// Prezentacja zdarzeń i przycisków 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 Przycisk_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja przycisków i zdarzeń JavaFX");
// Tworzy panel FlowPane, który stanie się węzłem korzenia.
// W tym przypadku pionowe i poziome odstępy pomiędzy
// komponentami będą miały wielkość 10.
FlowPane rootNode = new FlowPane(20,20);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 300, 100);
```

```
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę.
response = new Label("Kliknij przycisk");
// Tworzy dwa przyciski.
Button btnAlpha = new Button("Alfa");
Button btnBeta = new Button("Beta");
// Obsługa zdarzeń ActionEvent przycisku Alfa.
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Kliknięto przycisk Alfa.");
}
});
// Obsługa zdarzeń ActionEvent przycisku Beta.
btnBeta.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Kliknięto przycisk Beta.");
}
});
//Dodaje etykietę i przyciski do grafu sceny.
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Kontrolki Image i ImageView

Kilka kontrolek JavaFX pozwala na wyświetlanie wewnątrz nich obrazów. Przykładami takich kontrolek są etykiety i przyciski, które pozwalają na wyświetlanie obrazów oraz tekstu. Oprócz tego obrazy można także dodawać bezpośrednio na scenie. Podstawowe możliwości obsługi obrazów w JavaFX zapewniają dwie klasy: Image oraz ImageView. Pierwsza z nich hermetyzuje sam obrazek jako taki, z kolei klasa ImageView udostępnia możliwości niezbędne do jego wyświetlania. Obie te klasy należą do pakietu javafx.scene.image.

Klasa Image wczytuje obraz ze strumienia InputStream, wskazanego adresu URL bądź ścieżki do pliku graficznego. Definiuje ona kilka konstruktorów, w tym poniższy:

Image(String *url*)

Parametr *url* określa adres URL lub ścieżkę dostępu do pliku graficznego. Jeśli przekazany argument nie będzie prawidłowo zapisanym adresem URL, to klasa przyjmie, że stanowi on ścieżkę do pliku. Jeśli okaże się, że argument jest prawidłowym adresem URL, to obraz zostanie pobrany.

Pozostałe konstruktory klasy Image pozwalają na określanie różnego rodzaju opcji, takich jak szerokość i wysokość obrazu. I jeszcze jedna uwaga: klasa Image nie dziedziczy po klasie Node, co oznacza, że jej instancji nie można dodawać do grafu sceny.

W celu wyświetlania obrazu po utworzeniu obiektu Image należy utworzyć obiekt ImageView. Klasa ImageView dziedziczy po Node, co oznacza, że jej obiekty mogą być dodawane do grafu sceny. Klasa ta definiuje trzy konstruktory. Poniżej został przedstawiony pierwszy z nich:

ImageView(Image *image*)

Tworzy on obiekt ImageView, który będzie prezentował obraz określony przez przekazany obiekt Image.

Przykładowy kod programu:

```
package application;


// Program wczytuje i wyświetla obraz.
import javafx.application.*;
```

```
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.image.*;
public class Obraz_FX extends Application {
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa tytuł obszaru roboczego.
myStage.setTitle("Wyświetlanie obrazu");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia.
FlowPane rootNode = new FlowPane();
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 300, 200);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy obraz.
Image zima = new Image("zima.png");
//Image zima = new Image("http://img-
fotki.yandex.ru/get/4613/41885099.223/0_69ee5_3061cc7c_L.jpg");
// Tworzy obiekt ImageView korzystający z utworzonego
// wcześniej obiektu Image.
ImageView zima_View = new ImageView(zima);
//Dodaje obraz do grafu sceny.
rootNode.getChildren().add(zima_View);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Umieszczanie obrazów w innych kontrolkach
Dodawanie obrazów do etykiet

Zgodnie z informacjami podanymi w poprzednim rozdziale klasa Label reprezentuje etykiety. Taka etykieta może prezentować łańcuch, grafikę lub jedno i drugie.
Należy skorzystać z poniższego konstruktora klasy Label:
Label(String *str*, Node *image*)
Parametr *str* określa komunikat tekstowy, natomiast parametr *image* — obraz, który zostanie wyświetlony w etykiecie. Warto zwrócić uwagę, że parametr *image* jest typu Node. Takie rozwiązanie zapewnia bardzo dużą elastyczność, jeśli chodzi o obrazy dodawane do etykiet, jednak do naszych celów wystarczy użycie obiektu ImageView.

Przykładowy kod programu:

```
package application;
//Stosowanie obrazów w etykietach.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.scene.image.*;
public class Etykieta_FX extends Application {
```

```
public static void main(String[] args) {
//Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
Launch(args);
}
//Przesłonięta metoda start().
public void start(Stage myStage) {
//Określa nazwę obszaru roboczego.
myStage.setTitle("Stosowanie obrazów w etykietach");
//Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia.
FlowPane rootNode = new FlowPane();
//Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
//Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 600, 500);
//Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
//Tworzy obiekt ImageView zawierający wskazany obraz.
ImageView zima_View = new ImageView("zima.png");

//Tworzy etykietę zawierającą tekst i obraz.
Label zima_Label = new Label("Zima", zima_View);
// ustawienie sposobu wyświetlania obrazu i tekstu
zima_Label.setContentDisplay(ContentDisplay.TOP);
//Dodaje etykietę do grafu sceny.
rootNode.getChildren().add(zima_Label);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Wyświetlony został zarówno tekst, jak i obraz. Warto zwrócić uwagę, że tekst został umieszczony z prawej strony obrazu. To domyślny sposób działania etykiet; można go zmienić, wywołując metodę setContentDisplay(). Poniżej przedstawiona została deklaracja tej metody:
final void setContentDisplay(ContentDisplay *position*)
Wartość parametru *position* określa, w jaki sposób zostaną wyświetlone tekst i obraz. Musi to być jedna ze stałych zdefiniowanych w typie wyliczeniowym ContentDisplay:

- BOTTOM RIGHT
- CENTER TEXT_ONLY
- GRAPHICS_ONLY TOP
- LEFT

Dodawanie obrazów w przyciskach

W JavaFX przyciski są reprezentowane przez klasę Button.
Przyciski oprócz tekstu mogą zawierać grafikę lub tekst i grafikę w zależności od zastosowanego konstruktora.
W pierwszej kolejności należy utworzyć obiekt ImageView prezentujący wybrany obraz, a następnie dodać go do przycisku.
Jednym ze sposobów, by to zrobić, jest skorzystanie z poniższego konstruktora:
Button(String *str*, Node *image*)
Parametr *str* określa tekst prezentowany na przycisku, a parametr *image* — obraz, który zostanie na nim wyświetlony. Położenie obrazu w stosunku do tekstu na przycisku można określić w taki sam sposób jak w przypadku etykiet — przy użyciu metody setContentDisplay().

Przykład aplikacji:

```
package application;

//Umieszczanie obrazów na przyciskach.
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.image.*;
public class Przycisk_FX extends Application {
Label response;
public static void main(String[] args) {
//Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
Launch(args);
}
//Przesłonięta metoda start().
public void start(Stage myStage) {
//Określa nazwę obszaru roboczego.
myStage.setTitle("Umieszczanie obrazów na przyciskach");
//Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
//przypadku poziome i pionowe odstępy pomiędzy kontrolkami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
//Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
//Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 250, 450);
//Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
//Tworzy etykietę.
response = new Label("Kliknij przycisk");
//Tworzy dwa przyciski z tekstem i obrazkiem.
Button btnzima = new Button("Zima",
new ImageView("zima.png"));
Button btnlato = new Button("Lato", new ImageView("http://www.na-
pulpit.com/tapety/lato-zagajnik-brzozy-rzeka-kladka.jpeg"));
//Button btnlato = new Button("Lato", new ImageView("lato.png"));
//Button btnlato = new Button("Lato",
new ImageView("lato.png"));
//Rozmieszcza tekst poniżej obrazu.
btnzima.setContentDisplay(ContentDisplay.TOP);
btnlato.setContentDisplay(ContentDisplay.TOP);
//Obsługa zdarzeń przycisku zima.
btnzima.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Kliknięto zima.");
}
});
//Obsługa zdarzeń przycisku lato.
btnlato.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Kliknięto lato.");
}
});
//Dodaje etykietę i przyciski do grafu sceny.
rootNode.getChildren().addAll(btnzima, btnlato, response);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Kontrolka RadioButton

Kolejnym rodzajem przycisków udostępnianych przez JavaFX są **przyciski opcji.** Przyciski opcji to grupy wzajemnie wykluczających się przycisków, z których w danej chwili tylko jeden może być zaznaczony. Przyciski opcji są reprezentowane przez klasę RadioButton, która dziedziczy zarówno po klasie ButtonBase, jak i ToggleButton. Oprócz tego klasa ta implementuje także interfejs Toggle.
Do tworzenia kontrolek RadioButton będzie używany następujący konstruktor:
RadioButton(String *str*)
Parametr *str* określa etykietę przycisku. Podobnie jak w przypadku wszystkich innych przycisków kliknięcie przycisku opcji powoduje wygenerowanie zdarzenia.
Ze względu na swój charakter przyciski opcji muszą być organizowane w grupy. W dowolnej chwili tylko jeden przycisk w danej grupie może być zaznaczony. Jeśli użytkownik kliknie jeden z przycisków opcji należących do grupy, to wybrany dotychczas przycisk przestanie być zaznaczony. Grupy przycisków opcji są tworzone przy użyciu klasy ToggleGroup, należącej do pakietu javafx.scene.control.
Klasa ta udostępnia jedynie domyślny konstruktor.
Przyciski opcji są dodawane do grupy poprzez wywołanie na rzecz kontrolki przycisku metody setToggleGroup() zdefiniowanej w klasie ToggleButton. Jej deklaracja została przedstawiona poniżej:
final void setToggleGroup(ToggleGroup *tg*)
Parametr *tg* jest referencją do obiektu ToggleGroup reprezentującego grupę, do której dany przycisk opcji ma należeć. Po dodaniu do grupy wszystkich przycisków opcji zyskają one charakterystyczne dla siebie działanie. Ogólnie rzecz biorąc, jeśli przyciski opcji są stosowane w grupach, to podczas początkowego wyświetlania interfejsu użytkownika aplikacji jeden z tych przycisków będzie zaznaczony. Poniżej opisane zostały dwa sposoby, jak określić ten początkowo zaznaczony przycisk. Pierwszym rozwiązaniem jest wywołanie metody setSelected() na rzecz przycisku, który należy zaznaczyć. Metoda ta, przedstawiona poniżej, jest zdefiniowana w klasie ToggleButton (jednej z klas bazowych klasy RadioButton):
final void setSelected(boolean *state*)
Jeśli parametr *state* przyjmie wartość true, to przycisk opcji zostanie zaznaczony; w przeciwnym razie zaznaczenie zostanie z niego usunięte. W przypadku użycia tej metody nie będzie generowane żadne zdarzenie, nawet jeśli przycisk opcji zostanie zaznaczony.
Drugim sposobem początkowego zaznaczenia przycisku opcji jest wywołanie na jego rzecz metody fire(). Poniżej przedstawiona została jej deklaracja:
void fire()
Wywołanie tej metody powoduje wygenerowanie zdarzenia skierowanego do przycisku, o ile nie był on wcześniej zaznaczony. Najprostszym z nich jest odpowiadanie na zdarzenia ActionEvent, generowane w momencie zaznaczania przycisków.

```
package application;
// Prosty przykład stosowania przycisków opcji.
//
// Ten program odpowiada na zdarzenia ActionEvent
// generowane przez wybierane przyciski opcji. Program
// pokazuje także, jak można programowo generować
// zdarzenia.
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 Przycisk_Opcji_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
Launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Demonstracja działania Radio Buttons");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 290, 120);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę prezentującą aktualnie wybraną opcję.
response = new Label("");
// Tworzy przyciski opcji.
RadioButton rbTrain = new RadioButton("Pociąg");
RadioButton rbCar = new RadioButton("Samochód");
RadioButton rbPlane = new RadioButton("Samolot");
// Tworzy grupę przycisków opcji.
ToggleGroup tg = new ToggleGroup();
// Dodaje poszczególne przyciski do grupy.
rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
// Obiekt nasłuchujący obsługujący zdarzenia generowane przez przyciski opcji.
rbTrain.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Wybranym środkiem transportu jest pociąg.");
}
});
rbCar.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Wybranym środkiem transportu jest samochód.");
}
});
rbPlane.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
            response.setText("Wybranym środkiem transportu jest samolot.");
            }
            });
// Generuje zdarzenie dla pierwszego przycisku opcji.
// Powoduje to zaznaczenie przycisku i wygenerowanie
// zdarzenia ActionEvent.
```

```
rbTrain.fire();
// Dodaje etykietę i przyciski do grafu sceny.
rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Obsługa zdarzeń w grupie

Jedną z metod obsługi zdarzeń jest ActionEvent, to jednak czasami lepszym rozwiązaniem może być obsługa zdarzeń informujących o zmianach stanu grupy. Kiedy takie zdarzenie zostanie zgłoszone, obsługujący je obiekt nasłuchujący bez trudu będzie mógł określić, który przycisk opcji został zaznaczony, i odpowiednio zareagować. Aby skorzystać z tego rozwiązania, należy zarejestrować obiekt nasłuchujący obsługujący zdarzenia typu ChangeListener w obiekcie ToggleGroup.
Przykład kodu:

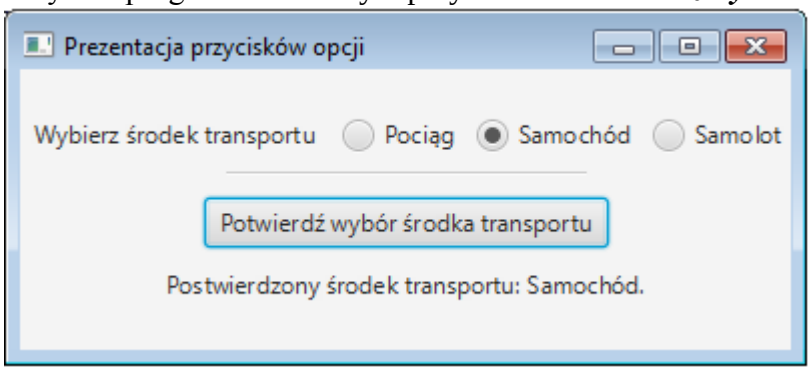
```
package application;
// Prosty przykład stosowania przycisków opcji.
//
// Ten program odpowiada na zdarzenia ActionEvent
// generowane przez wybierane przyciski opcji. Program
// pokazuje także, jak można programowo generować
// zdarzenia.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.beans.value.*;

public class Przycisk_Opcji_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Demonstracja działania Radio Buttons");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 290, 120);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę prezentującą aktualnie wybraną opcję.
response = new Label("");
// Tworzy przyciski opcji.
RadioButton rbTrain = new RadioButton("Pociąg");
RadioButton rbCar = new RadioButton("Samochód");
RadioButton rbPlane = new RadioButton("Samolot");
// Tworzy grupę przycisków opcji.
ToggleGroup tg = new ToggleGroup();
// Dodaje poszczególne przyciski do grupy.
rbTrain.setToggleGroup(tg);
```

```
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
//Używa obiektu nasłuchującego, by odpowiadać na zdarzenia
//zmiany stanu grupy przycisków opcji.
tg.selectedTogglePropery().addListener(new ChangeListener<Toggle>() {
public void changed(ObservableValue<? extends Toggle> changed,
Toggle oldVal, Toggle newVal) {
//Rzutuje nową zaznaczoną kontrolkę do typu RadioButton.
RadioButton rb = (RadioButton) newVal;
//Wyświetla komunikat o zaznaczonym przycisku.
response.setText("Wybrany środek transportu to: " + rb.getText());
}
});
//Zaznacza pierwszy przycisk opcji. Spowoduje to wygenerowanie
//zdarzenia informującego o zmianie stanu grupy.
rbTrain.setSelected(true);
// Dodaje etykietę i przyciski do grafu sceny.
rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);
            // Wyświetla obszar roboczy i scenę.
            myStage.show();
}
}
```

Obsługa zdarzeń metody alternatywne

Nie zawsze obsługa zdarzeń jest generowanych przez przyciski opcji jest potrzebna, znacznie częściej chcemy znać aktualny status tych obiektów

Przykład programu z dodanym przyciskiem ***Potwierdź wybór środka transportu***

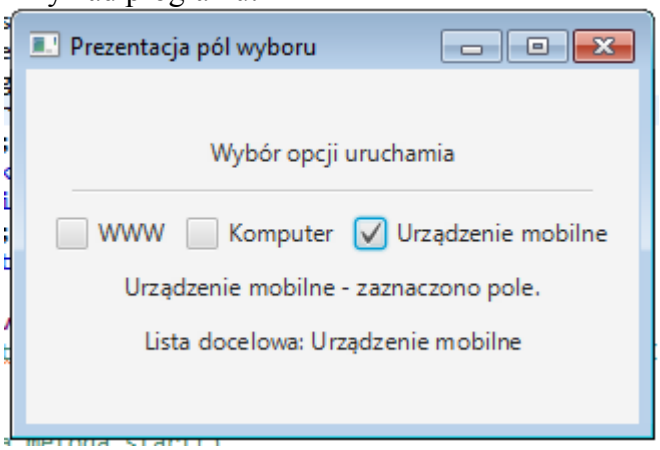



```
package application;
// Ten przykład demonstruje, w jaki sposób można programowo
// w wybranym momencie odczytać, który z przycisków opcji
// w grupie jest zaznaczony; dzięki czemu nie trzeba
// obsługiwać zdarzeń generowanych przez przyciski opcji
// lub ich grupę.
//
// W tym programie nie są obsługiwane żadne zdarzenia
// generowane przez przyciski opcji lub ich grupę.
// Zamiast tego aktualnie zaznaczone pole wyboru zostaje
// określone podczas obsługi kliknięcia przycisku
// Potwierdź wybór środka transportu.
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 Przyciski_opcji_FX extends Application {
```

```
Label response;
ToggleGroup tg;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja przycisków opcji");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 250, 140);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy dwie etykiety.
Label choose = new Label(" Wybierz środek transportu ");
response = new Label("Nie potwierdzono środka transportu.");
// Tworzy przycisk służący do potwierdzenia wybranego środka transportu.
Button btnConfirm = new Button("Potwierdź wybór środka transportu");
// Tworzy przyciski opcji.
RadioButton rbTrain = new RadioButton("Pociąg");
RadioButton rbCar = new RadioButton("Samochód");
RadioButton rbPlane = new RadioButton("Samolot");
// Tworzy grupę przycisków opcji.
tg = new ToggleGroup();
//Dodaje poszczególne przyciski do grupy.
rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
//Określa początkowo zaznaczony przycisk opcji.
rbTrain.setSelected(true);
//Obsługa zdarzeń przycisku potwierdzającego wybór.
btnConfirm.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
//Pobiera aktualnie zaznaczony przycisk opcji.
RadioButton rb = (RadioButton) tg.getSelectedToggle();
//Wyświetla informacje o zaznaczonym przycisku.
response.setText("Postwierdzony środek transportu: " + rb.getText() + ".");
}
});
//Tworzy separator, by poprawić wygląd układu.
Separator separator = new Separator();
separator.setPrefWidth(180);
//Dodaje do grafu sceny etykiety i przyciski.
rootNode.getChildren().addAll(choose, rbTrain, rbCar, rbPlane,
separator, btnConfirm, response);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Kontrolka CheckBox

Pola wyboru są powszechnie znanym obiektem Klasa CheckBox dziedziczy po klasie ButtonBase. W JavaFX pola wyboru obsługują trzy stany. Pierwsze dwa są oczywiste pole może być zaznaczone lub niezaznaczone. Jednak dostępny jest także trzeci stan **nieokreślony** (nazywany także czasami niezdefiniowanym). Ten trzeci stan jest zazwyczaj używany w celu zasygnalizowania, że pole wyboru nie zostało jeszcze ustawione bądź stosowanie go w bieżącej sytuacji nie jest właściwe. Aby móc korzystać z tego trzeciego stanu pól wyboru, trzeba jawnie zażądać jego udostępnienia.
Klasa CheckBox definiuje dwa konstruktory. Pierwszy z nich jest konstruktorem domyślnym, a drugi, przedstawiony poniżej, pozwala na przekazanie łańcucha identyfikującego tworzone pole:
CheckBox(String *str*)
Konstruktor ten tworzy pole wyboru prezentujące tekst przekazany przy użyciu parametru *str*.
Podobnie jak wszystkie inne przyciski, także pola wyboru w momencie wybrania generują zdarzenia ActionEvent.
Przykład programu:




```
package application;
// Prezentacja 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 Przyciski_Wyboru_FX extends Application {
CheckBox cbWeb;
CheckBox cbDesktop;
CheckBox cbMobile;
Label response;
Label allTargets;
String targets = "";
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
      // Określa nazwę obszaru roboczego.
      myStage.setTitle("Prezentacja pól wyboru");
      // Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W
tym
      // przypadku odstępy pomiędzy komponentami wynoszą 10.
```

```
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 300, 140);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
Label heading = new Label("Wybór opcji uruchamia");
// Tworzy etykietę, która będzie używana do wyświetlania
// stanu pola wyboru.
response = new Label("Nie zaznaczono żadnej opcji");
// Tworzy etykietę informującą o wszystkich wybranych opcjach.
allTargets = new Label("Lista docelowa: <brak>");
// Tworzy pola wyboru.
cbWeb = new CheckBox("WWW");
cbDesktop = new CheckBox("Komputer");
cbMobile = new CheckBox("Urządzenie mobilne");
// Obsługa zdarzeń pól wyboru.
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
if(cbWeb.isSelected())
response.setText("WWW - zaznaczono pole.");
else
response.setText("WWW - usunięto zaznaczenie pola.");
showAll();
}
});
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
if(cbDesktop.isSelected())
response.setText("Komputer - zaznaczono pole.");
else
response.setText("Komputer - usunięto zaznaczenie pola.");
showAll();
}
});
cbMobile.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
if(cbMobile.isSelected())
response.setText("Urządzenie mobilne - zaznaczono pole.");
else
response.setText("Urządzenie mobilne - usunięto zaznaczenie pola.");
showAll();
}
});
// Tworzy separator, by poprawić wygląd układu.
Separator separator = new Separator();
separator.setPrefWidth(260);
// Dodaje wszystkie kontrolki do grafu sceny.
rootNode.getChildren().addAll(heading, separator, cbWeb, cbDesktop,
cbMobile, response, allTargets);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
// Aktualizacja i wyświetlenie listy docelowej.
void showAll() {
targets = "";
if(cbWeb.isSelected()) targets = "WWW ";
if(cbDesktop.isSelected()) targets += "Komputer ";
if(cbMobile.isSelected()) targets += "Urządzenie mobilne";
if(targets.equals("")) targets = "<brak>";
allTargets.setText("Lista docelowa: " + targets);
}
}
```

Trzeci stan dla przycisków wyboru

Kontrolki CheckBox standardowo udostępniają domyślnie dwa stany odpowiadające sytuacjom, gdy pole jest zaznaczone lub niezaznaczone. Aby dodać do niech możliwość obsługi trzeciego stanu, nieokreślonego, trzeba to jawnie zdefiniować. Do tego celu służy przedstawiona poniżej metoda setAllowIndeterminate():

final void setAllowIndeterminate(boolean *enable*)

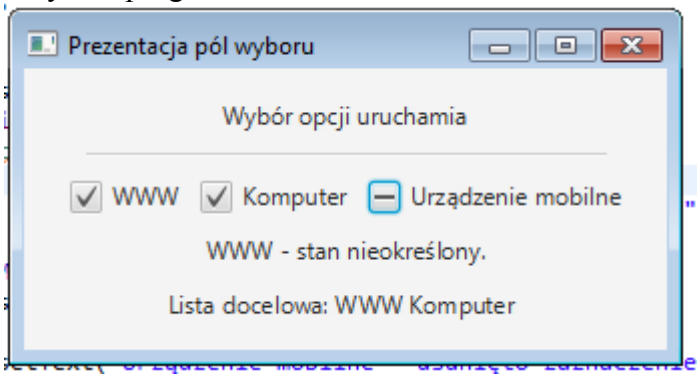
Przekazanie w jej wywołaniu wartości true włącza możliwość korzystania ze stanu nieokreślonego, jeśli natomiast zostanie przekazana wartość false, to korzystanie z tego stanu nie będzie możliwe. Po włączeniu obsługi stanu niezdefiniowanego użytkownik może wybierać wszystkie trzy stany pola wyboru — zaznaczony, niezaznaczony i nieokreślony — klikając na nie.

Aby sprawdzić, czy pole wyboru znajduje się w stanie nieokreślonym, należy skorzystać z przedstawionej poniżej metody isIndeterminate():

final boolean isIndeterminate()

Zwraca ona wartość true, jeśli pole znajduje się w stanie nieokreślonym, a wartość false w przeciwnym razie.

Przykład programu :




```
package application;
// Prezentacja 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 Przyciski_Wyboru_FX extends Application {
CheckBox cbWeb;
CheckBox cbDesktop;
CheckBox cbMobile;
Label response;
Label allTargets;
String targets = "";
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
      // Określa nazwę obszaru roboczego.
      myStage.setTitle("Prezentacja pól wyboru");
      // Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W
tym
      // przypadku odstępy pomiędzy komponentami wynoszą 10.
      FlowPane rootNode = new FlowPane(10, 10);
      // Wyrównuje kontrolki na scenie do środka.
```

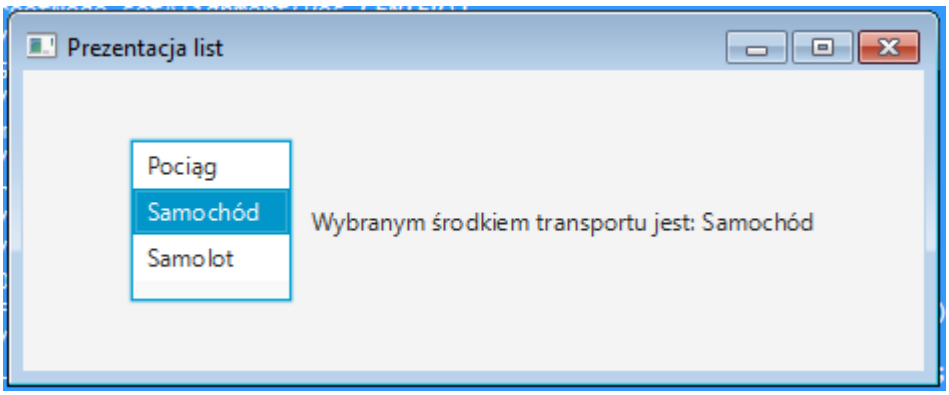
```
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 300, 140);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
Label heading = new Label("Wybór opcji uruchamia");
// Tworzy etykietę, która będzie używana do wyświetlania
// stanu pola wyboru.
response = new Label("Nie zaznaczono żadnej opcji");
// Tworzy etykietę informującą o wszystkich wybranych opcjach.
allTargets = new Label("Lista docelowa: <brak>");
// Tworzy pola wyboru.
cbWeb = new CheckBox("WWW");
cbDesktop = new CheckBox("Komputer");
cbMobile = new CheckBox("Urządzenie mobilne");
//ustawienie możliwości dodania trzeciego stanu
cbWeb.setAllowIndeterminate(true);
cbDesktop.setAllowIndeterminate(true);
cbMobile.setAllowIndeterminate(true);
// Obsługa zdarzeń pól wyboru.
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
      //testowanie trzeciego stanu
      if(cbWeb.isIndeterminate())
            response.setText("WWW - stan nieokreślony.");
            else if(cbWeb.isSelected())
            response.setText("WWWW - zaznaczono pole.");
            else
            response.setText("WWW - usunięto zaznaczenie pola.");
showAll();
}
});
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
      //testowanie trzeciego stanu
                  if(cbDesktop.isIndeterminate())
                        response.setText("WWW - stan nieokreślony.");
                        else if(cbDesktop.isSelected())
response.setText("Komputer - zaznaczono pole.");
else
response.setText("Komputer - usunięto zaznaczenie pola.");
showAll();
}
});
cbMobile.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
      //testowanie trzeciego stanu
                  if(cbMobile.isIndeterminate())
                        response.setText("WWW - stan nieokreślony.");
                        else
      if(cbMobile.isSelected())
response.setText("Urządzenie mobilne - zaznaczono pole.");
else
response.setText("Urządzenie mobilne - usunięto zaznaczenie pola.");
showAll();
}
});
// Tworzy separator, by poprawić wygląd układu.
Separator separator = new Separator();
separator.setPrefWidth(260);
// Dodaje wszystkie kontrolki do grafu sceny.
rootNode.getChildren().addAll(heading, separator, cbWeb, cbDesktop,
cbMobile, response, allTargets);
// Wyświetla obszar roboczy i scenę.
```

```
myStage.show();
}
// Aktualizacja i wyświetlenie listy docelowej.
void showAll() {
targets = "";
if(cbWeb.isSelected()) targets = "WWW ";
if(cbDesktop.isSelected()) targets += "Komputer ";
if(cbMobile.isSelected()) targets += "Urządzenie mobilne";
if(targets.equals("")) targets = "<brak>";
allTargets.setText("Lista docelowa: " + targets);
}
}
```

Kontrolki list
ListView

Często stosowanymi kontrolkami są listy, które na platformie JavaFX są reprezentowane przez klasę ListView. Listy są kontrolkami wyświetlającymi listy elementów, spośród których użytkownik może wybrać jeden lub więcej elementów.
ListView jest klasą sparametryzowaną i ma następującą deklarację:
class ListView<T>
Parametr typu T określa typ elementów przechowywanych na liście. Bardzo często są to elementy typu String, choć możliwe jest stosowanie także innych typów.
Klasa ListView definiuje dwa konstruktory. Pierwszy z nich jest konstruktorem domyślnym, który tworzy pustą kontrolkę ListView. Z kolei drugi konstruktor pozwala na przekazanie listy elementów kontrolki. Oto deklaracja tego konstruktora:
ListView(ObservableList<T> *list*)
Parametr *list* określa listę elementów, które będą wyświetlane w kontrolce. Jak widać, jest to obiekt typu ObservableList, definiujący listę obiektów, którą można obserwować. Klasa ta dziedziczy po typie java.util.List, a zatem udostępnia wszystkie standardowe metody kolekcji. Interfejs ObservableList należy do pakietu javafx.collection.
Najprostszym sposobem utworzenia obiektu ObservableList, którego będzie można użyć do utworzenia kontrolki ListView, jest skorzystanie z metody fabrycznej observableArrayList(), zdefiniowanej jako statyczna metoda klasy FXCollections
Deklaracja tej wersji metody observableArrayList(), która będzie używana w kolejnych przykładach:
static <E> ObservableList<E> observableArrayList(E ... *elements*) W tym przypadku parametr typu E określa typ elementów przekazywanych jako *elements*. Domyślnie kontrolki ListView pozwalają, by w danej chwili był zaznaczony tylko jeden element listy. Niemniej jednak aby umożliwić zaznaczanie wielu elementów, wystarczy zmienić używany tryb wyboru.
Określenie wysokość lub szerokość z dostosowaniem do bieżących potrzeb jest możliwe przy zastosowaniu podanych metod
setPrefHeight() oraz setPrefWidth():
final void setPrefHeight(double *height*)
final void setPrefWidth(double *height*)
Ewentualnie można także określić oba wymiary w jednym wywołaniu, używając w tym celu metody setPrefSize():
void setPrefSize(double *width*, double *height*)
Kontrolki ListView można używać na dwa podstawowe sposoby. Pierwszy z nich polega na ignorowaniu generowanych przez nią zdarzeń i pobieraniu aktualnie zaznaczonego elementu listy wtedy, gdy będzie to konieczne. Drugim sposobem jest monitorowanie zmiany stanu zaznaczenia elementów listy poprzez zarejestrowanie odpowiedniego obiektu nasłuchującego. W ten sposób można wykonać jakieś czynności za każdym razem, gdy zmieni się aktualnie zaznaczony element listy.

Obsługa zdarzeń listy, wymaga: użycia metody getSelectionModel():
final MultipleSelectionModel<T> getSelectionModel()
Zwraca ona referencję do obiektu modelu. MultipleSelectionModel to klasa definiująca model pozwalający na zaznaczanie wielu elementów listy, dziedziczy ona po klasie SelectionModel. Możliwość zaznaczania wielu elementów listy będzie dostępna wyłącznie w przypadku, gdy ten model zaznaczania zostanie włączony.
Używając obiektu modelu zwróconego przez wywołanie metody getSelectionModel(), można pobrać referencję do właściwości aktualnie zaznaczonego elementu listy, która definiuje, co ma się stać, kiedy element zostanie zaznaczony. Referencję tę można pobrać, wywołując metodę selectedItemProperty(). Poniżej przedstawiona została jej deklaracja:
final ReadOnlyObjectProperty<T> selectedItemProperty()
Przykład programu z obsługą list i zdarzeń




```
package application;
// Prezentacja list.
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 Lista_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
Launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja list");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 450, 150);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę.
response = new Label("Wybierz środek transportu");
// Tworzy listę ObservableList elementów, które zostaną wyświetlone
// w kontrolce ListView.
ObservableList<String> transportTypes =
FXCollections.observableArrayList( "Pociąg", "Samochód", "Samolot" );
```

```
// Tworzy listę - kontrolkę ListView.
ListView<String> lvTransport = new ListView<String>(transportTypes);
// Określa preferowaną szerokość i wysokość listy.
lvTransport.setPrefSize(80, 80);
// Pobiera aktualnie używany model wyboru.
MultipleSelectionModel<String> lvSelModel =
lvTransport.getSelectionModel();
// Określa obiekt nasłuchujący, który będzie obsługiwał zmianę
// zaznaczonego elementu listy.
lvSelModel.selectedItemProperty().addListener(
new ChangeListener<String>() {
public void changed(ObservableValue<? extends String> changed,
String oldVal, String newVal) {
// Wyświetla zaznaczony element listy.
response.setText("Wybranym środkiem transportu jest: " + newVal);
}
});
// Dodaje etykietę i listę do grafu sceny.
rootNode.getChildren().addAll(lvTransport, response);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Paski przewijania w kontrolkach ListView

Jedną z bardzo użytecznych cech kontrolek ListView jest to, że gdy zawierają więcej elementów, niż można wyświetlić w ich obszarze, automatycznie są do nich dodawane paski przewijania.

```
package application;
// Prezentacja list.
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 Lista_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja list");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 450, 150);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę.
response = new Label("Wybierz środek transportu");
// Tworzy listę ObservableList elementów, które zostaną wyświetlone
// w kontrolce ListView.
```

```
ObservableList<String> transportTypes =
FXCollections.observableArrayList( "Pociąg", "Samochód", "Samolot",
"Rower", "Na piechotę" );
// Tworzy listę - kontrolkę ListView.
ListView<String> lvTransport = new ListView<String>(transportTypes);
// Określa preferowaną szerokość i wysokość listy.
lvTransport.setPrefSize(80, 80);
// Pobiera aktualnie używany model wyboru.
MultipleSelectionModel<String> lvSelModel =
lvTransport.getSelectionModel();
// Określa obiekt nasłuchujący, który będzie obsługiwał zmianę
// zaznaczonego elementu listy.
lvSelModel.selectedItemProperty().addListener(
new ChangeListener<String>() {
public void changed(ObservableValue<? extends String> changed,
String oldVal, String newVal) {
// Wyświetla zaznaczony element listy.
response.setText("Wybranym środkiem transportu jest: " + newVal);
}
});
// Dodaje etykietę i listę do grafu sceny.
rootNode.getChildren().addAll(lvTransport, response);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Włączanie możliwości wielokrotnego wyboru

Włączenie możliwości wybierania więcej niż jednego elementu listy, wymaga wywołania metody setSelectionMode() kontrolki ListView i przekazania w jej wywołaniu wartość SelectionMode.MULTIPLE. Poniżej przedstawiona została deklaracja tej metody:
final void setSelectionMode(SelectionMode *mode*)
Parametr *mode* musi mieć jedną z dwóch wartości:

- SelectionMode.MULTIPLE
- SelectionMode.SINGLE.

Kiedy zostanie włączony tryb wyboru wielu elementów, listę aktualnie zaznaczonych elementów można pobierać na dwa sposoby: jako listę zaznaczonych indeksów oraz jako listę zaznaczonych elementów.
Aby pobrać listę zaznaczonych elementów, należy wywołać metodę getSelectedItems() obiektu modelu. Poniżej przedstawiona została jej deklaracja:
ObservableList<T> getSelectedItems()

```
package application;
// Prezentacja list.
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 Lista_FX extends Application {
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
```

```
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja list");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 450, 250);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę.
response = new Label("Wybierz środek transportu");
// Tworzy listę ObservableList elementów, które zostaną wyświetlone
// w kontrolce ListView.
ObservableList<String> transportTypes =
FXCollections.observableArrayList( "Pociąg", "Samochód", "Samolot",
"Rower", "Na piechotę" );
// Tworzy listę - kontrolkę ListView.
ListView<String> lvTransport = new ListView<String>(transportTypes);
// Określa preferowaną szerokość i wysokość listy.
lvTransport.setPrefSize(180, 180);
// Pobiera aktualnie używany model wyboru.
MultipleSelectionModel<String> lvSelModel =
lvTransport.getSelectionModel();
//Jawne ustawienie wielokrotnego zaznaczania elementów listy
lvTransport.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
// Określa obiekt nasłuchujący, który będzie obsługiwał zmianę
// zaznaczonego elementu listy.
lvSelModel.selectedItemProperty().addListener(
new ChangeListener<String>() {
public void changed(ObservableValue<? extends String> changed,
String oldVal, String newVal) {
String selItems = "";
ObservableList<String> selected =
lvTransport.getSelectionModel().getSelectedItems();
// Wyświetla zaznaczone elementy.
for(int i=0; i < selected.size(); i++)
selItems += "\n " + selected.get(i);
response.setText("Wybrane środki transportu: " + selItems);
}
});
// Dodaje etykietę i listę do grafu sceny.
rootNode.getChildren().addAll(lvTransport, response);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Kontrolka ComboBox

Klasa ComboBox - lista kombinowana prezentuje jeden, wybrany element, lecz ma także przycisk pozwalający rozwinąć listę, z której użytkownik może wybrać inny element. Można także pozwolić użytkownikom na edycję wybranego elementu listy kombinowanej. Klasa ComboBox dziedziczy po klasie ComboBoxBase, która implementuje większość jej możliwości funkcjonalnych. W odróżnieniu od list ListView listy kombinowane zostały zaprojektowane w **celu wybierania tylko jednego elementu**.
ComboBox jest klasą sparametryzowaną, zadeklarowaną w następujący sposób:
class ComboBox<T>
Parametr typu T określa typ elementów listy kombinowanej. Bardzo często elementy te są typu String, choć możliwe jest także stosowanie innych typów.
Klasa ComboBox definiuje dwa konstruktory. Pierwszy z nich jest konstruktorem

domyślnym, który tworzy pustą kontrolkę. Drugi konstruktor, pozwala na przekazanie listy elementów, które zostaną wyświetlone w kontrolce:
ComboBox(ObservableList<T> *list*)
Parametr *list* określa listę elementów, które zostaną wyświetlone w kontrolce. Parametr ten jest typu ObservableList — definiuje on listę elementów, której zawartość może być obserwowana.
W momencie zaznaczenia innego elementu listy kontrolki ComboBox generują zdarzenie ActionEvent. Oprócz tego generują one także zdarzenia informujące o zmianie aktualnie wybranej wartości.
Można także całkowicie pominąć obsługę zdarzeń i po prostu pobierać aktualnie wybraną wartość w momencie, gdy będzie to potrzebne. Aktualnie wybraną wartość można pobrać przy użyciu przedstawionej poniżej metody getValue():
final T getValue()
Jeśli wartość listy nie została jeszcze określona (niezależnie od tego, czy zrobiłby to użytkownik, czy też program), to wywołanie metody getValue() zwróci wartość null. Aby programowo określić wartość kontrolki ComboBox, należy skorzystać z przedstawionej poniżej metody setValue():
final void setValue(T *newVal*)
Przykład użycia kontrolki:

```
package application;

// Prezentacja listy kombinowanej.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.collections.*;
import javafx.event.*;
public class ComboBox_FX extends Application {
ComboBox<String> cbTransport;
Label response;
public static void main(String[] args) {
	// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
	launch(args);
	}
	// Przesłonięta metoda start().
	public void start(Stage myStage) {
	// Określa nazwę obszaru roboczego.
	myStage.setTitle("Prezentacja listy kombinowanej");
	// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W
tym
	// przypadku odstępy pomiędzy komponentami wynoszą 10.
	FlowPane rootNode = new FlowPane(10, 10);
	// Wyrównuje kontrolki na scenie do środka.
	rootNode.setAlignment(Pos.CENTER);
	// Tworzy obiekt sceny.
	Scene myScene = new Scene(rootNode, 380, 190);
	// Zapisuje obiekt sceny w obszarze roboczym.
	myStage.setScene(myScene);
	// Tworzy etykietę.
	response = new Label();
	// Tworzy listę ObservableList elementów, które będą prezentowane
	// na liście kombinowanej.
	ObservableList<String> transportTypes =
	FXCollections.observableArrayList( "Pociąg", "Samochód", "Samolot" );
	// Tworzy listę kombinowaną - kontrolkę ComboBox.
	cbTransport = new ComboBox<String>(transportTypes);
```

```
        // Ustawia wartość domyślną.
        cbTransport.setValue("Pociąg");
        // Ustawia zawartość etykiety response, tak by odpowiadała ona domyślnej
        // wartości listy cbTransport.
        response.setText("Wybranym środkiem transportu jest: " +
cbTranspport.getValue());
        // Obsługa zdarzeń ActionEvent generowanych przez kontrolkę ComboBox.
        cbTransport.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
        response.setText("Wybranym środkiem transportu jest: " +
cbTranspport.getValue());
        }
        });
        // Dodaje etykietę i listę kombinowaną do grafu sceny.
        rootNode.getChildren().addAll(cbTransport, response);
        // Wyświetla obszar roboczy i scenę.
        myStage.show();
        }
        }
```

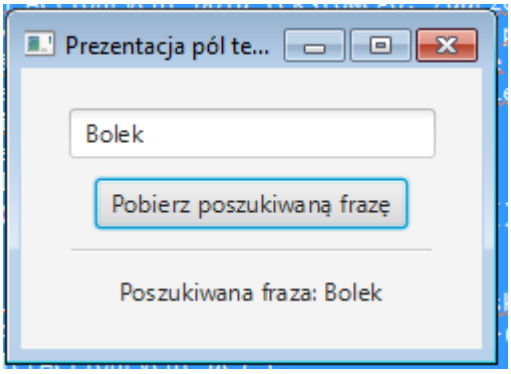
Edycja elementu listy w kontrolce ComboBox

Przekazanie w wywołaniu tej metody wartości true spowoduje włączenie możliwości edycji; przekazanie wartości false sprawi, że edycja nie będzie możliwa. Aby wypróbować możliwość edycji wartości kontrolki ComboBox, należy dodać do ostatniego programu następujący wiersz kodu:
cbTransport.setEditable(true);

Kontrolka TextField

JavaFX udostępnia kilka kontrolek tekstowych. W tym kontrolka TextField. Pozwala ona na wpisanie jednego wiersza tekstu, a zatem może być przydatna do podawania imion, identyfikatorów, adresów itp. Jak wszystkie kontrolki tekstowe, także i klasa TextField dziedziczy po klasie TextInputControl definiującej znaczną część jej możliwości funkcjonalnych. Klasa TextField definiuje dwa konstruktory. Pierwszym z nich jest konstruktor domyślny, który tworzy puste pole tekstowe o domyślnej wielkości. Drugi konstruktor pozwala na określenie początkowej zawartości pola. Określenie wielkości pola tekstowego. Do tego celu służy przedstawiona poniżej metoda setPrefColumnCount():
final void setPrefColumnCount(int *columns*)
Wartość parametru *columns* jest używana przez kontrolkę TextField do określenia jej wielkości. Tekst umieszczony w kontrolce TextField można ustawić, wywołując metodę setText(). Z kolei do pobierania tego tekstu służy metoda getText(). Oprócz tych podstawowych możliwości kontrolki TextField udostępniają także kilka innych, które warto poznać; między innymi pozwalają one na wycinanie, wklejanie i dodawanie tekstu. Istnieje także możliwość programowego zaznaczenia fragmentu tekstu umieszczonego w polu. Jedną ze szczególnie użytecznych możliwości kontrolki TextField jest określanie komunikatu początkowego, który będzie w niej wyświetlany, gdy użytkownik niczego jeszcze w niej nie wpisał.
Jego treść można określić przy użyciu pokazanej poniżej metody setPromptText():
final void setPromptText(String *str*)

Przykład programu:




```
package application;
// Prezentacja pól tekstowych - kontrolek TextField.
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 TextBox_FX extends Application {
TextField tf;
Label response;
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
// Przesłonięta metoda start().
public void start(Stage myStage) {
// Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja pól tekstowych");
// Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
// przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
// Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
// Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 230, 140);
// Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
// Tworzy etykietę, w której będzie wyświetlana
// poszukiwana fraza odczytana z pola tekstowego.
response = new Label("Poszukiwana fraza: ");
// Tworzy przycisk używany do pobrania frazy.
Button btnGetText = new Button("Pobierz poszukiwaną frazę");
// Tworzy pole tekstowe.
tf = new TextField();
// Określa komunikat początkowy.
tf.setPromptText("Wpisz poszukiwaną frazę");
//Ustawia preferowaną szerokość pola.
tf.setPrefColumnCount(15);
//Obsługa zdarzeń ActionEvent pola tekstowego. Zdarzenia te
//są generowane w wyniku naciśnięcia klawisza Enter podczas
//edycji zawartości pola tekstowego. W tym programie po
//kliknięciu Enter zawartość pola tekstowego zostanie
//pobrana i wyświetlona.
tf.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Poszukiwana fraza: " + tf.getText());
}
});
//Pobranie tekstu z pola w wyniku kliknięcia przycisku.
```

```
btnGetText.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("Poszukiwana fraza: " + tf.getText());
}
});
//Tworzy separator, by poprawić wygląd układu.
Separator separator = new Separator();
separator.setPrefWidth(180);
//Dodaje wszystkie kontrolki do grafu sceny.
rootNode.getChildren().addAll(tf, btnGetText, separator, response);
//Wyświetla obszar roboczy i scenę.
myStage.show();
}
}
```

Interfejs użytkownika -GUI
zastosowanie efektów i transformacji

Największą zaletą platformy JavaFX jest możliwość modyfikowania wyglądu każdej kontrolki w scenie poprzez zastosowanie **efektów** oraz **transformacji**. Dzięki nim graficzny interfejsu użytkownika tworzonych aplikacji może zyskać wyszukany, nowoczesny wygląd, którego oczekują użytkownicy.

Efekty

Efekty zostały zaimplementowane jako abstrakcyjna klasa Effect oraz jej konkretne klasy pochodne. Wszystkie one należą do pakietu javafx.scene.effect. Korzystając z tych efektów, można modyfikować wygląd węzła dodanego do grafu sceny.

Efekt Opis

- Bloom Zwiększa jasność najjaśniejszej części węzła.
- BoxBlur Rozmazuje węzeł.
- DropShadow Wyświetla cień umieszczony poniżej węzła.
- Glow Generuje efekt rozświetlenia.
- InnerShadow Wyświetla cień wewnątrz węzła.
- Lighting Tworzy efekt cienia generowanego przez źródło światła.
- Reflection Wyświetla odbicie.

Aby zastosować efekt w wybranym węźle, należy wywołać metodę setEffect() zdefiniowaną w klasie Node. Poniżej przedstawiona została jej deklaracja:
final void setEffect(Effect *effect*)
Parametr *effect* określa efekt, który należy zastosować. Aby nie używać żadnego efektu, w wywołaniu tej metody należy przekazać wartość null. A zatem aby dodać efekt do węzła, w pierwszej kolejności należy utworzyć instancję odpowiedniej klasy, a następnie przekazać ją w wywołaniu metody setEffect(). Po wykonaniu tej operacji efekt będzie używany za każdym razem, gdy kontrolka będzie wyświetlana (oczywiście o ile tylko dany efekt jest obsługiwany w używanym środowisku).
W celu przedstawienia możliwości, jakie dają efekty, poniżej został zamieszczony program używający dwóch z nich: Glow oraz InnerShadow.
Efekt Glow sprawia, że węzeł wydaje się być rozświetlony. Istnieje przy tym możliwość kontroli intensywności tego rozświetlenia. Aby zastosować ten efekt, w pierwszej kolejności należy utworzyć obiekt Glow. W przedstawionym przykładzie jest on tworzony przy użyciu poniższego konstruktora:
Glow(double *glowLevel*)
Parametr *glowLevel* określa poziom rozświetlenia i może przyjmować wartości z zakresu od 0.0 do 1.0.
Intensywność rozświetlenia można także zmieniać już po utworzeniu obiektu Glow; służy

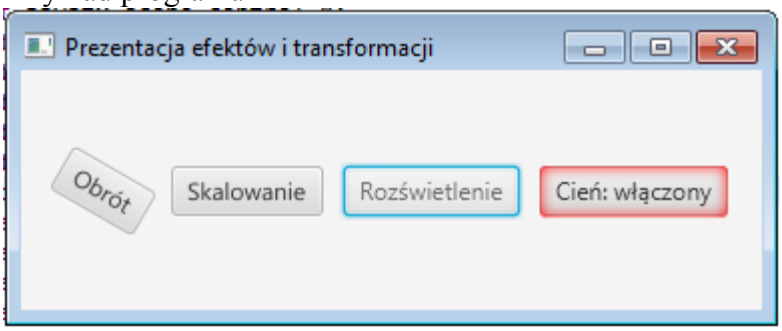
do tego przedstawiona poniżej metoda setLevel():
final void setLevel(double *glowLevel*)
Podobnie jak w konstruktorze, także i tutaj parametr *glowLevel* musi być wartością z zakresu od 0.0 do 1.0.
Klasa InnerShadow generuje efekt symulujący cień umieszczony wewnątrz węzła. Udostępnia ona kilka różnych konstruktorów:
InnerShadow(double *radius*, Color *shadowColor*)
Parametr *radius* określa promień cienia wewnątrz węzła. W gruncie rzeczy parametr ten określa wielkość cienia. Drugi parametr, *shadowColor*, określa kolor cienia. Musi to być wartość typu Color, a dokładniej rzecz biorąc: javafx.scene.paint.Color. Klasa ta definiuje wiele stałych reprezentujących różne kolory, na przykład Color.GREEN, Color.RED lub Color.BLUE, dzięki czemu stosowanie jest bardzo łatwe i wygodne.

Transformacje

Transformacje są zaimplementowane przez abstrakcyjną klasę Transform zdefiniowaną w pakiecie javafx.scene.transform. Jej czterema najczęściej używanymi klasami pochodnymi są: Rotate, Scale, Shear oraz Translate. Pozwalają one odpowiednio na: obrót, przeskalowanie, zniekształcenie oraz przesunięcie węzła. W danym węźle można zastosować więcej niż jedną transformację; na przykład można go obrócić i przeskalować. Transformacje są stosowane przy użyciu metod klasy Node, Jednym ze sposobów zastosowania transformacji jest dodanie jej do listy transformacji węzła. Listę tę można pobrać, wywołując przedstawioną poniżej metodę getTransforms()klasy Node:
final ObservableList<Transform> getTransforms()
Metoda ta zwraca referencję do listy transformacji używanych w węźle. Aby zastosować transformację, wystarczy ją dodać do tej listy, wywołując jej metodę add(). Listę transformacji można wyczyścić, używając metody clear(). Istnieje także możliwość usunięcia konkretnego elementu tej listy; do tego celu służy metoda remove().
W niektórych przypadkach istnieje także możliwość bezpośredniego zastosowania transformacji w węźle poprzez określenie wartości odpowiedniej właściwości klasy Node. Na przykład wywołując metodę setRotate() i przekazując do niej wartość reprezentującą kąt, można określić kąt obrotu węzła, przy czym oś obrotu będzie się znajdować w jego środku. W podobny sposób za pomocą metod setScaleX() oraz setScaleY() można określić przeskalowanie węzła, a za pomocą metod setTranslateX()
oraz setTranslateY() — jego przesunięcie.
W celu przedstawienia transformacji w kolejnym przykładzie zostaną użyte dwie z nich: Rotate oraz Scale. Ogólny sposób stosowania pozostałych jest taki sam. Transformacja Rotate obraca węzeł względem określonego punktu. Klasa ta definiuje kilka konstruktorów:
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 użyć konstruktora domyślnego, a parametry transformacji określić później, już po utworzeniu obiektu Rotate. Takie rozwiązanie zostało zaprezentowane w programie przykładowym. W takim przypadku parametry transformacji są określane przy użyciu przedstawionych poniżej metod setAngle(),
setPivotX() oraz setPivotY():
final void setAngle(double *angle*)
final void setPivotX(double *x*)
final void setPivotY(double *y*)
Także w przypadku tych metod parametr *angle* określa kąt obrotu wyrażony w stopniach, a parametry *x* i *y* określają współrzędne punktu obrotu.
Transformacja Scale skaluje węzeł o podany współczynnik. Klasa Scale definiuje kilka konstruktorów.
Scale(double *widthFactor*, double *heightFactor*)

Parametr *widthFactor* określa współczynnik skalowania szerokości węzła, natomiast parametr *heightFactor* — współczynnik skalowania jego wysokości. Oba te współczynniki można także zmieniać za pomocą metod setX() oraz setY() po utworzeniu obiektu Transform:
final void setX(double *widthFactor*)
final void setY(double *heightFactor*)
Podobnie jak w przypadku przedstawionego wcześniej konstruktora, także tutaj parametr *widthFactor* określa współczynnik skalowania szerokości węzła, a parametr *heightFactor* — współczynnik skalowania jego wysokości.

Przykład programu




```
package application;
// Prezentacja transformacji Rotate i Scaling oraz
// efektów Glow oraz InnerShadow.
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 Efekty_FX extends Application {
double angle = 0.0;
double glowVal = 0.0;
boolean shadow = false;
double scaleFactor = 1.0;
// Utworzenie początkowych efektów i transformacji.
Glow glow = new Glow(0.0);
InnerShadow innerShadow = new InnerShadow(10.0, Color.RED);
Rotate rotate = new Rotate();
Scale scale = new Scale(scaleFactor, scaleFactor);
// Tworzy przyciski.
Button btnRotate = new Button("Obrót");
Button btnGlow = new Button("Rozświetlenie");
Button btnShadow = new Button("Cień: wyłączony");
Button btnScale = new Button("Skalowanie");
public static void main(String[] args) {
// Wywołuje metodę launch(), która uruchamia aplikację JavaFX.
launch(args);
}
//Przesłonięta metoda start().
public void start(Stage myStage) {
//Określa nazwę obszaru roboczego.
myStage.setTitle("Prezentacja efektów i transformacji");
//Tworzy panel FlowPane, który zostanie użyty jako węzeł korzenia. W tym
//przypadku odstępy pomiędzy komponentami wynoszą 10.
FlowPane rootNode = new FlowPane(10, 10);
```

```
//Wyrównuje kontrolki na scenie do środka.
rootNode.setAlignment(Pos.CENTER);
//Tworzy obiekt sceny.
Scene myScene = new Scene(rootNode, 370, 120);
//Zapisuje obiekt sceny w obszarze roboczym.
myStage.setScene(myScene);
//Stosuje początkowy efekt rozświetlenia.
btnGlow.setEffect(glow);
//Dodaje obrót do listy transformacji przycisku Obrót.
btnRotate.getTransforms().add(rotate);
//Dodaje skalowanie do listy transformacji przycisku Skalowanie.
btnScale.getTransforms().add(scale);
//Obsługa zdarzeń ActionEvent przycisku Obrót.
btnRotate.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
//Każde kliknięcie przycisku powoduje obrócenie przycisku
//o 30 stopni względem jego środka.
angle += 30.0;
rotate.setAngle(angle);
rotate.setPivotX(btnRotate.getWidth()/2);
rotate.setPivotY(btnRotate.getHeight()/2);
}
});
//Obsługa zdarzeń ActionEvent przycisku Skalowanie.
btnScale.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
//Każde kliknięcie przycisku powoduje zmianę jego współczynników
//skalowania.
scaleFactor += 0.1;
if(scaleFactor > 1.0) scaleFactor = 0.4;
scale.setX(scaleFactor);
scale.setY(scaleFactor);
}
});
//Obsługa zdarzeń ActionEvent przycisku Rozświetlenie.
btnGlow.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
//Każde kliknięcie przycisku powoduje zmianę wartości efektu.
glowVal += 0.1;
if(glowVal > 1.0) glowVal = 0.0;
//Ustawia nową wartość efektu.
glow.setLevel(glowVal);
}
});
// Obsługa zdarzeń ActionEvent przycisku Cień.
btnShadow.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
// Każde kliknięcie przycisku powoduje włączenie lub
// wyłączenie cienia.
shadow = !shadow;
if(shadow) {
btnShadow.setEffect(innerShadow);
btnShadow.setText("Cień: włączony");
} else {
btnShadow.setEffect(null);
btnShadow.setText("Cień: wyłączony");
}
}
});
// Dodaje kontrolki do grafu sceny.
rootNode.getChildren().addAll(btnRotate, btnScale, btnGlow, btnShadow);
// Wyświetla obszar roboczy i scenę.
myStage.show();
}
```

```
}
```

Opracowano na podstawie:
HERBERT SCHILDT - „Java. Kompendium programisty. Wydanie IX” Wydawnictwo Helion.