

ORACLE®



# 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

Original edition copyright © 2018 by McGraw-Hill Education.  
All rights reserved.

Polish edition copyright © 2018 by HELION SA  
All rights reserved.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz HELION SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

HELION SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto: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>

Mozesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>O autorze</b> .....	13
<b>O redaktorze technicznym</b> .....	14
<b>Wstęp</b> .....	15
<b>1. Podstawy Javy</b> .....	21
Pochodzenie Javy .....	22
Java a języki C i C++ .....	22
Wpływ Javy na Internet .....	23
Java uprościła programowanie aplikacji internetowych .....	23
Aplety Javy .....	24
Bezpieczeństwo .....	24
Przenośność .....	25
Magiczny kod bajtowy .....	25
Coś więcej niż aplety .....	26
Terminologia Javy .....	27
Programowanie obiektowe .....	27
Hermetyzacja .....	28
Polimorfizm .....	29
Dziedziczenie .....	29
Java Development Kit .....	29
Pierwszy prosty program .....	30
Wprowadzenie tekstu programu .....	31
Kompilowanie programu .....	31
Pierwszy program wiersz po wierszu .....	32
Obsługa błędów składni .....	34
Drugi prosty program .....	34
Inne typy danych .....	36
Dwie instrukcje sterujące .....	38
Instrukcja if .....	38
Pętla for .....	40
Bloki kodu .....	41
Średnik i pozycja kodu w wierszu .....	42
Wcięcia .....	42
Słowa kluczowe języka Java .....	44
Identyfikatory .....	45
Biblioteki klas .....	45
Test sprawdzający .....	45

<b>2. Typy danych i operatory</b> .....	<b>47</b>
Dlaczego typy danych są tak ważne .....	47
Typy proste .....	48
Typy całkowite .....	48
Typy zmiennoprzecinkowe .....	49
Znaki .....	50
Typ logiczny .....	51
Literały .....	53
Literały szesnastkowe, ósemkowe i binarne .....	53
Specjalne sekwencje znaków .....	53
Literały łańcuchowe .....	54
Zmienne .....	55
Inicjalizacja zmiennej .....	55
Dynamiczna inicjalizacja .....	55
Zasięg deklaracji i czas istnienia zmiennych .....	56
Operatory .....	58
Operatory arytmetyczne .....	58
Inkrementacja i dekrementacja .....	59
Operatory relacyjne i logiczne .....	60
Warunkowe operatory logiczne .....	62
Operator przypisania .....	63
Skrótowe operatory przypisania .....	63
Konwersje typów w instrukcjach przypisania .....	64
Rzutowanie typów niezgodnych .....	65
Priorytet operatorów .....	67
Wyrażenia .....	68
Konwersja typów w wyrażeniach .....	68
Odstępy i nawiasy .....	70
Test sprawdzający .....	70
<b>3. Instrukcje sterujące</b> .....	<b>71</b>
Wprowadzanie znaków z klawiatury .....	71
Instrukcja if .....	72
Zagnieżdżanie instrukcji if .....	73
Drabinka if-else-if .....	74
Instrukcja switch .....	75
Zagnieżdżanie instrukcji switch .....	78
Pętla for .....	80
Wariacje na temat pętli for .....	81
Brakujące elementy .....	82
Pętla nieskończona .....	83
Pętle bez ciała .....	83
Deklaracja zmiennych sterujących wewnątrz pętli .....	84
Rozszerzona pętla for .....	85
Pętla while .....	85
Pętla do-while .....	86
Przerywanie pętli instrukcją break .....	90
Zastosowanie break jako formy goto .....	91
Zastosowanie instrukcji continue .....	95
Pętle zagnieżdżone .....	99
Test sprawdzający .....	99

<b>4. Wprowadzenie do klas, obiektów i metod .....</b>	<b>101</b>
Podstawy klas .....	101
Ogólna postać klasy .....	102
Definiowanie klasy .....	102
Jak powstają obiekty .....	105
Referencje obiektów i operacje przypisania .....	105
Metody .....	106
Dodajemy metodę do klasy Vehicle .....	106
Powrót z metody .....	108
Zwracanie wartości .....	109
Stosowanie parametrów .....	110
Dodajemy sparametryzowaną metodę do klasy Vehicle .....	112
Konstruktory .....	117
Konstruktory z parametrami .....	118
Dodajemy konstruktor do klasy Vehicle .....	119
Operator new .....	120
Odzyskiwanie pamięci .....	120
Słowo kluczowe this .....	121
Test sprawdzający .....	122
<b>5. Więcej typów danych i operatorów .....</b>	<b>123</b>
Tablice .....	123
Tablice jednowymiarowe .....	124
Tablice wielowymiarowe .....	128
Tablice dwuwymiarowe .....	128
Tablice nieregularne .....	129
Tablice o trzech i więcej wymiarach .....	130
Inicjalizacja tablic wielowymiarowych .....	130
Alternatywna składnia deklaracji tablic .....	131
Przypisywanie referencji tablic .....	131
Wykorzystanie składowej length .....	132
Styl for-each pętli for .....	137
Iteracje w tablicach wielowymiarowych .....	139
Zastosowania rozszerzonej pętli for .....	140
Łańcuchy znaków .....	141
Tworzenie łańcuchów .....	141
Operacje na łańcuchach .....	142
Tablice łańcuchów .....	144
Łańcuchy są niezmiennie .....	144
Stosowanie łańcuchów do sterowania instrukcją switch .....	145
Wykorzystanie argumentów wywołania programu .....	146
Operatory bitowe .....	147
Operatory bitowe AND, OR, XOR i NOT .....	147
Operatory przesunięcia .....	151
Skrótowe bitowe operatory przypisania .....	153
Operator ? .....	155
Test sprawdzający .....	157

<b>6. Więcej o metodach i klasach</b> .....	<b>159</b>
Kontrola dostępu do składowych klasy .....	159
Modyfikatory dostępu w Javie .....	160
Przekazywanie obiektów do metod .....	164
Sposób przekazywania argumentów .....	165
Zwracanie obiektów .....	167
Przeciążanie metod .....	169
Przeciążanie konstruktorów .....	173
Rekurencja .....	177
Słowo kluczowe static .....	178
Bloki static .....	181
Klasy zagnieżdżone i klasy wewnętrzne .....	184
Zmienne liczby argumentów .....	186
Metody o zmiennej liczbie argumentów .....	187
Przeciążanie metod o zmiennej liczbie argumentów .....	189
Zmienna liczba argumentów i niejednoznaczność .....	190
Test sprawdzający .....	191
<b>7. Dziedziczenie</b> .....	<b>193</b>
Podstawy dziedziczenia .....	193
Dostęp do składowych a dziedziczenie .....	196
Konstruktory i dziedziczenie .....	198
Użycie słowa kluczowego super do wywołania konstruktora klasy bazowej .....	199
Użycie słowa kluczowego super do dostępu do składowych klasy bazowej .....	203
Wielopoziomowe hierarchie klas .....	206
Kiedy wywoływane są konstruktory? .....	208
Referencje klasy bazowej i obiekty klasy pochodnej .....	209
Przesłaniec metod .....	213
Przesłaniec metod i polimorfizm .....	215
Po co przesłaniać metody? .....	216
Zastosowanie przesłaniania metod w klasie TwoDShape .....	217
Klasy abstrakcyjne .....	220
Słowo kluczowe final .....	223
final zapobiega przesłanianiu .....	223
final zapobiega dziedziczeniu .....	223
Użycie final dla zmiennych składowych .....	224
Klasa Object .....	225
Test sprawdzający .....	226
<b>8. Pakiety i interfejsy</b> .....	<b>227</b>
Pakiety .....	227
Definiowanie pakietu .....	228
Wyszukiwanie pakietów i zmienna CLASSPATH .....	229
Prosty przykład pakietu .....	229
Pakiety i dostęp do składowych .....	230
Przykład dostępu do pakietu .....	231
Składowe chronione .....	232
Import pakietów .....	234
Biblioteka klas Java używa pakietów .....	235
Interfejsy .....	235
Implementacje interfejsów .....	237

Referencje interfejsu .....	240
Zmienne w interfejsach .....	245
Interfejsy mogą dziedziczyć .....	246
Domyślne metody interfejsów .....	247
Podstawowe informacje o metodach domyślnych .....	248
Praktyczny przykład metody domyślnej .....	249
Problemy wielokrotnego dziedziczenia .....	250
Stosowanie metod statycznych w interfejsach .....	251
Stosowanie metod prywatnych w interfejsach .....	252
Ostatnie uwagi o pakietach i interfejsach .....	253
Test sprawdzający .....	253
<b>9. Obsługa wyjątków .....</b>	<b>255</b>
Hierarchia wyjątków .....	256
Podstawy obsługi wyjątków .....	256
Słowa kluczowe try i catch .....	256
Prosty przykład wyjątku .....	257
Konsekwencje nieprzechwycenia wyjątku .....	259
Wyjątki umożliwiają obsługę błędów .....	260
Użycie wielu klauzul catch .....	261
Przechwytywanie wyjątków klas pochodnych .....	261
Zagnieżdżanie bloków try .....	263
Generowanie wyjątku .....	264
Powtórne generowanie wyjątku .....	264
Klasa Throwable .....	265
Klauzula finally .....	267
Użycie klauzuli throws .....	268
Trzy dodatkowe możliwości wyjątków .....	269
Wyjątki wbudowane w Javę .....	270
Tworzenie klas pochodnych wyjątków .....	272
Test sprawdzający .....	276
<b>10. Obsługa wejścia i wyjścia .....</b>	<b>279</b>
Strumienie wejścia i wyjścia .....	280
Strumienie bajtowe i strumienie znakowe .....	280
Klasy strumieni bajtowych .....	280
Klasy strumieni znakowych .....	280
Strumienie predefiniowane .....	281
Używanie strumieni bajtowych .....	282
Odczyt wejścia konsoli .....	282
Zapis do wyjścia konsoli .....	284
Odczyt i zapis plików za pomocą strumieni bajtowych .....	285
Odczyt z pliku .....	285
Zapis w pliku .....	288
Automatyczne zamykanie pliku .....	290
Odczyt i zapis danych binarnych .....	292
Pliki o dostępie swobodnym .....	296
Strumienie znakowe .....	298
Odczyt konsoli za pomocą strumieni znakowych .....	298
Obsługa wyjścia konsoli za pomocą strumieni znakowych .....	301

Obsługa plików za pomocą strumieni znakowych .....	302
Klasa FileWriter .....	302
Klasa FileReader .....	303
Zastosowanie klas opakowujących do konwersji łańcuchów numerycznych .....	304
Test sprawdzający .....	311
<b>11. Programowanie wielowątkowe .....</b>	<b>313</b>
Podstawy wielowątkowości .....	313
Klasa Thread i interfejs Runnable .....	314
Tworzenie wątku .....	315
Usprawnienie i dwie modyfikacje .....	318
Tworzenie wielu wątków .....	323
Jak ustalić, kiedy wątek zakończył działanie? .....	326
Priorytety wątków .....	328
Synchronizacja .....	331
Synchronizacja metod .....	331
Synchronizacja instrukcji .....	334
Komunikacja międzywątkowa .....	336
Przykład użycia metod wait() i notify() .....	337
Wstrzymywanie, wznawianie i kończenie działania wątków .....	341
Test sprawdzający .....	346
<b>12. Typy wyliczeniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje .....</b>	<b>347</b>
Wyliczenia .....	348
Podstawy wyliczeń .....	348
Wyliczenia są klasami .....	350
Metody values() i valueOf() .....	350
Konstruktory, metody, zmienne instancji a wyliczenia .....	351
Dwa ważne ograniczenia .....	353
Typy wyliczeniowe dziedziczą po klasie Enum .....	353
Automatyczne opakowywanie .....	358
Typy opakowujące .....	359
Podstawy automatycznego opakowywania .....	360
Automatyczne opakowywanie i metody .....	361
Automatyczne opakowywanie i wyrażenia .....	362
Przestroga .....	364
Import składowych statycznych .....	364
Adnotacje (metadane) .....	366
Test sprawdzający .....	369
<b>13. Typy sparametryzowane .....</b>	<b>371</b>
Podstawy typów sparametryzowanych .....	372
Prosty przykład typów sparametryzowanych .....	372
Parametryzacja dotyczy tylko typów obiektowych .....	375
Typy sparametryzowane różnią się dla różnych argumentów .....	375
Klasa sparametryzowana o dwóch parametrach .....	376
Ogólna postać klasy sparametryzowanej .....	377
Ograniczanie typów .....	377
Stosowanie argumentów wieloznacznych .....	380
Ograniczanie argumentów wieloznacznych .....	382



Metody sparametryzowane .....	384
Konstruktory sparametryzowane .....	386
Interfejsy sparametryzowane .....	386
Typy surowe i tradycyjny kod .....	392
Wnioskowanie typów i operator diamentowy .....	395
Wymazywanie .....	396
Błędy niejednoznaczności .....	396
Ograniczenia związane z typami sparametryzowanymi .....	397
Zakaz tworzenia instancji parametru typu .....	397
Ograniczenia dla składowych statycznych .....	397
Ograniczenia tablic sparametryzowanych .....	398
Ograniczenia związane z wyjątkami .....	399
Dalsze studiowanie typów sparametryzowanych .....	399
Test sprawdzający .....	399
<b>14. Wyrażenia lambda i referencje metod .....</b>	<b>401</b>
Przedstawienie wyrażen lambda .....	402
Podstawowe informacje o wyrażeniach lambda .....	402
Interfejsy funkcyjne .....	403
Wyrażenia lambda w działaniu .....	405
Blokowe wyrażenia lambda .....	408
Sparametryzowane interfejsy funkcyjne .....	409
Wyrażenia lambda i przechwytywanie zmiennych .....	415
Zgłaszanie wyjątków w wyrażeniach lambda .....	416
Referencje metod .....	417
Referencje metod statycznych .....	417
Referencje metod instancyjnych .....	419
Referencje konstruktorów .....	423
Predefiniowane interfejsy funkcyjne .....	425
Test sprawdzający .....	426
<b>15. Moduły .....</b>	<b>429</b>
Podstawowe informacje o modułach .....	430
Przykład prostego modułu .....	431
Kompilowanie i uruchamianie przykładowej aplikacji .....	434
Dokładniejsze informacje o instrukcjach requires i exports .....	435
java.base i moduły platformy .....	436
Stary kod i moduł nienazwany .....	437
Eksportowanie do konkretnego modułu .....	438
Wymagania przechodnie .....	439
Stosowanie usług .....	443
Podstawowe informacje o usługach i dostawcach usług .....	444
Słowa kluczowe związane z usługami .....	444
Przykład stosowania usług i modułów .....	445
Dodatkowe cechy modułów .....	451
Moduły otwarte .....	451
Instrukcja opens .....	451
requires static .....	451
Dalsze samodzielne poznawanie modułów .....	452
Test sprawdzający .....	453

<b>16. Wprowadzenie do Swing</b> .....	<b>455</b>
Pochodzenie i filozofia Swing .....	456
Komponenty i kontenery .....	457
Komponenty .....	458
Kontenery .....	458
Panele kontenerów szczytowych .....	459
Menedżery układu .....	459
Pierwszy program wykorzystujący Swing .....	460
Pierwszy program Swing wiersz po wierszu .....	461
Obsługa zdarzeń w Swing .....	464
Zdarzenia .....	464
Obiekty nasłuchujące .....	465
Klasy zdarzeń i interfejsy obiektów nasłuchujących .....	465
Komponent JButton .....	465
Komponent JTextField .....	468
Komponent JCheckBox .....	471
Komponent JList .....	474
Wykorzystanie anonimowych klas wewnętrznych lub wyrażeń lambda do obsługi zdarzeń .....	481
Test sprawdzający .....	482
<b>17. Wprowadzenie do JavaFX</b> .....	<b>485</b>
Podstawowe pojęcia JavaFX .....	486
Pakiety JavaFX .....	486
Klasy Stage oraz Scene .....	486
Węzły i graf sceny .....	487
Układy .....	487
Klasa Application oraz metody cyklu życia .....	487
Uruchamianie aplikacji JavaFX .....	488
Szkielet aplikacji JavaFX .....	488
Kompilacja i uruchamianie programów JavaFX .....	491
Wątek aplikacji .....	491
Prosta kontrolka JavaFX: Label .....	491
Stosowanie przycisków i zdarzeń .....	493
Podstawy obsługi zdarzeń .....	493
Przedstawienie kontrolki Button .....	494
Przedstawienie obsługi zdarzeń i stosowania przycisków .....	494
Trzy kolejne kontrolki JavaFX .....	497
Pola wyboru .....	497
Listy .....	501
Pola tekstowe .....	505
Przedstawienie efektów i transformacji .....	508
Efekty .....	508
Transformacje .....	509
Prezentacja zastosowania efektów i transformacji .....	511
Co dalej? .....	513
Test sprawdzający .....	514

<b>A</b>	<b>Rozwiązania testów sprawdzających .....</b>	<b>515</b>
	Rozdział 1. Podstawy Javy .....	515
	Rozdział 2. Typy danych i operatory .....	517
	Rozdział 3. Instrukcje sterujące .....	518
	Rozdział 4. Wprowadzenie do klas, obiektów i metod .....	520
	Rozdział 5. Więcej typów danych i operatorów .....	521
	Rozdział 6. Więcej o metodach i klasach .....	524
	Rozdział 7. Dziedziczenie .....	528
	Rozdział 8. Pakiety i interfejsy .....	529
	Rozdział 9. Obsługa wyjątków .....	531
	Rozdział 10. Obsługa wejścia i wyjścia .....	533
	Rozdział 11. Programowanie wielowątkowe .....	536
	Rozdział 12. Typy wycieniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje .....	538
	Rozdział 13. Typy sparametryzowane .....	541
	Rozdział 14. Wyrażenia lambda i referencje metod .....	544
	Rozdział 15. Moduły .....	547
	Rozdział 16. Wprowadzenie do Swing .....	548
	Rozdział 17. Wprowadzenie do JavaFX .....	552
<b>B</b>	<b>Komentarze dokumentacyjne .....</b>	<b>557</b>
	Znaczniki javadoc .....	557
	@author .....	558
	{@code} .....	558
	@deprecated .....	558
	{@docRoot} .....	559
	@exception .....	559
	@hidden .....	559
	{@index} .....	559
	{@inheritDoc} .....	559
	{@link} .....	559
	{@linkplain} .....	560
	{@literal} .....	560
	@param .....	560
	@provides .....	560
	@return .....	560
	@see .....	560
	@since .....	560
	@throws .....	561
	@uses .....	561
	{@value} .....	561
	@version .....	561
	Ogólna postać komentarza dokumentacyjnego .....	561
	Wynik działania programu javadoc .....	561
	Przykład użycia komentarzy dokumentacyjnych .....	562

<b>C</b>	<b>Przegląd technologii Java Web Start .....</b>	<b>563</b>
	Czym jest Java Web Start? .....	563
	Cztery kluczowe aspekty Java Web Start .....	564
	Aplikacje Java Web Start wymagają pliku JAR .....	564
	Aplikacje Java Web Start są podpisywane cyfrowo .....	564
	Java Web Start bazuje na JNLP .....	565
	Tworzenie odnośnika do pliku JNLP .....	566
	Eksperymenty z Java Web Start z wykorzystaniem lokalnego systemu plików .....	567
	Utworzenie pliku JAR aplikacji ButtonDemo .....	567
	Utworzenie magazynu kluczy i podpisanie pliku ButtonDemo.jar .....	568
	Utworzenie pliku JNLP dla aplikacji ButtonDemo .....	569
	Utworzenie pliku HTML o nazwie StartBD.html .....	570
	Dodanie pliku ButtonDemo.jnlp do listy wyjątków w aplikacji Java Control Panel .....	570
	Wykonanie aplikacji ButtonDemo z poziomu przeglądarki .....	570
	Uruchamianie aplikacji Java Web Start przy użyciu programu javaws .....	571
	Stosowanie technologii Java Web Start z apletami .....	571
<b>D</b>	<b>Wprowadzenie do JShell .....</b>	<b>573</b>
	Podstawy JShell .....	573
	Wyświetlanie, edycja i ponowne wykonywanie kodu .....	575
	Dodanie metody .....	576
	Utworzenie klasy .....	577
	Stosowanie interfejsu .....	577
	Przetwarzanie wyrażeń i wbudowane zmienne .....	578
	Importowanie pakietów .....	579
	Wyjątki .....	579
	Inne polecenia JShell .....	580
	Dalsze poznawanie możliwości JShell .....	581
<b>E</b>	<b>Więcej słów kluczowych języka Java .....</b>	<b>583</b>
	Modyfikatory transient i volatile .....	583
	instanceof .....	584
	strictfp .....	584
	assert .....	584
	Metody rodzime .....	585
	Inna postać this .....	585
	<b>Skorowidz .....</b>	<b>587</b>

# 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.

Koniecznym 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 zacniemy 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 zaczną 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 stworzyć 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 przesłać. 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 scenaGłówna)
```

```
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 przesłać, 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 przesłać.

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, konieczne 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 jakiegokolwiek 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(); ←————— Tworzenie korzenia.

// Tworzy scenę.
Scene myScene = new Scene(rootNode, 300, 200); ←————— Tworzenie 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 przesłanie 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 przesłanie 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 wywołania 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("Szkielec 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 tym 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 przesyłanie 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 kontrolki. 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 kontrolki.

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 jakiegokolwiek 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 `ObservableList` 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.
        Label myLabel = new Label("JavaFX ma duże możliwości");

        // Dodaje etykietę do grafu sceny.
        rootNode.getChildren().add(myLabel);

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

↑ Tworzenie etykiety.

← Dodanie etykiety do grafu sceny.

## Ekspert odpowiada

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

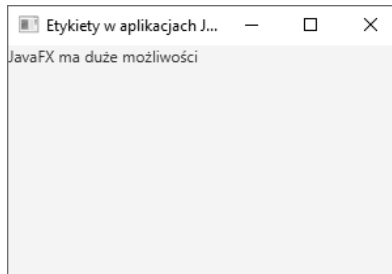
**Odpowiedź:** Owszem, można. Aby usunąć węzeł z grafu sceny, należy wywołać metodę `remove()` obiektu `ObservableList`. Na przykład poniższa instrukcja:

```
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(myLabel)
```

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 kontrolki JavaFX i tworzenie grafu sceny, to jednak nie pokazuje, w jaki sposób można obsługiwać zdarzenia. A jest to bardzo istotne zagadnienie, gdyż większość kontrolki używanych do tworzenia graficznego interfejsu użytkownika generuje zdarzenia, które są następnie obsługiwane przez aplikację. Na przykład kiedy użytkownik korzysta z takich kontrolki 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 kontrolki 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 EventHandler<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 etykietce 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");
        // ← Utworzenie dwóch przycisków.

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

        // Obsługa zdarzenia ActionEvent przycisku Dół.
        btnDown.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                response.setText("Nacisnąłeś przycisk Dół.");
            }
        });
        // ← Zdefiniowanie obiektów nasłuchujących
        // zdarzeń ActionEvent przycisków.
    }
}

```

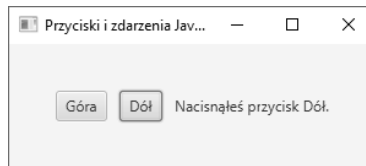
```
// Dodaje etykietę i przyciski do grafu sceny.
rootNode.getChildren().addAll(btnUp, btnDown, response);

// 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 klikniętemu 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 kontrolki 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 odstępu.



Interesujący jest także poniższy wiersz kodu, określający wyrównanie kontrolki umieszczonej 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 kontrolki, 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 kontrolki JavaFX. Pozwalają także przedstawić kilka powszechnie stosowanych technik. Kiedy opanujesz te techniki, pozostałe kontrolki będziesz mógł poznać samodzielnie.

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

```

← Tworzenie 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);
}
}

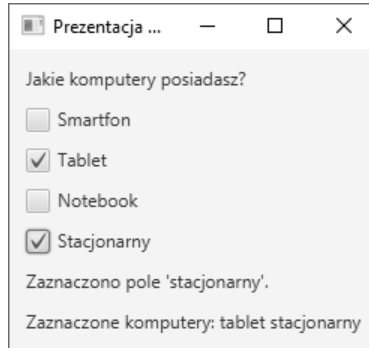
```

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 — niezany — 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 nieznanym 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 isIndetermined();
```

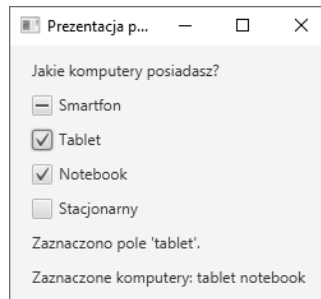
Metoda ta zwraca wartość *true*, jeśli pole wyboru znajduje się w stanie nieznanym, a wartość *false* w przeciwnym razie. Obiekt nasłuchujący, który obsługuje zdarzenia pola wyboru, będzie musiał sprawdzać, czy pole znajduje się w tym trzecim stanie. W tym celu zmodyfikuj obiekt nasłuchujący zdarzeń 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 nieznanym");
        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 nieznanym, 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 kontrolki *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 kontrolki 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 *elements*.

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 `SelectionMode`. 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);
    }
}
```

↓  
Tworzenie kontrolki `ListView`, prezentującej  
elementy umieszczone na liście `computerTypes`.

```

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

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

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

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

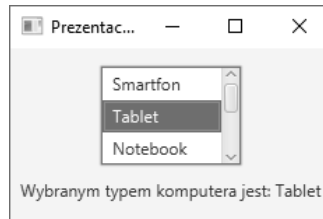
// Wyświetla scenę i obszar roboczy.
myStage.show();
}
}

```

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:

```

MultipleSelectionMode<String> lvSelModel =
    lvComputers.getSelectionModel();

```

Jak już wspominałem, kontrolka ListView używa modelu zaznaczania MultipleSelectionMode 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:



```

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

            // Wyświetla nazwę wybranego elementu listy.
            response.setText("Wybrany 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 kontrolki `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 kontrolki 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 kontrolki `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(); ←————— Tworzenie pola tekstowego.

// Określa komunikat informacyjny.
tf.setPromptText("Wpisz imię."); ←————— Określenie treści komunikatu informacyjnego.

// Podaje preferowaną liczbę kolumn — wielkość pola.
tf.setPrefColumnCount(15); ←————— Określenie szerokości pola w kolumnach.

// 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());
————— Obsługa zdarzeń generowanych przez pole tekstowe.

// 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);

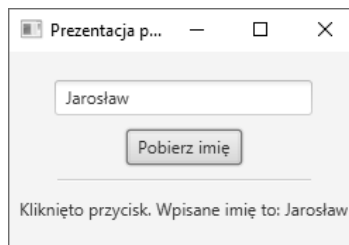
// Wyświetla scenę i obszar roboczy.
myStage.show();
}
}

```

Postać okna tego programu przedstawia rysunek 17.6.

W powyższym programie warto zwrócić uwagę na zastosowanie wyrażen lambda jako obiektów nasłuchujących zdarzeń. Obsługa każdego ze zdarzeń sprowadza się do jednego wywołania metody, dzięki czemu doskonale nadaje się do zaimplementowania przy użyciu wyrażen 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 wyszukanim, 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.effect`. Korzystając z tych efektów, można modyfikować wygląd węzłów umieszczonych w grafie sceny. Dostępnych jest kilka predefiniowanych efektów, a wybrane z nich przedstawiłem poniżej:

<code>Bloom</code>	Zwiększa jasność jasnej części węzła.
<code>BoxBlur</code>	Zamazuje węzeł.
<code>DropShadow</code>	Wyświetla cień umieszczony poniżej węzła.
<code>Glow</code>	Tworzy efekt rozświetlenia.
<code>InnerShadow</code>	Wyświetla cień wewnątrz węzła.
<code>Lighting</code>	Tworzy efekt cienia wywołwanego przez źródło światła.
<code>Reflection</code>	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.

Effekt `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 wężła, 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, double 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 wyjaśnił 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 `getTransforms()` 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);
```

Utworzenie efektów i transformacji.

```
    // 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.
```

## 512 Java. Przewodnik dla początkujących

```
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);
    }
});
```



```

    }
  });

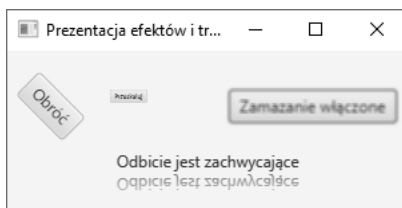
  // Dodaje etykietę i kontrolkę listy do grafu sceny.
  rootNode.getChildren().addAll(btnRotate, btnScale, btnBlur, reflect);

  // 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 stworzyć 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 twardej. 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 `Grow` 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

## A

adnotacje, 366, 368, 538  
    znacznikowe, 367  
algorytm Quicksort, 182  
anonimowe klasy wewnętrzne, 481  
API, Application Programming Interface, 235  
aplet, 24, 26, 571  
aplikacja ButtonDemo, 567, 569, 570  
aplikacje  
    Java Web Start, 564  
    JavaFX, 485, 488  
architektura model-widok-kontroler, 457  
argumenty, 110, 146  
    wieloznaczne, 380, 382  
    zmiennie liczby, 186, 190  
ASCII, 292  
automatyczna konwersja typu, 64  
automatyczne  
    opakowywanie, 358–362, 538  
    zamykanie pliku, 290  
    zarządzanie zasobami, 269  
AWT, Abstract Window Toolkit, 456

## B

bezpieczeństwo, 24, 27  
biblioteka Swing, 455  
biblioteki klas, 45, 235  
blok kodu, 41, 42  
    finally, 267  
    static, 181  
    try, 263  
blokowe wyrażenia lambda, 408  
błędy, 260  
    niejednoznaczności, 396  
    składni, 34  
bufor, 72

## C

certyfikat, 565  
ciało wyrażeniowe, 408  
czas istnienia zmiennych, 56

## D

dane binarne, 292  
definiowanie  
    klasy, 102  
    pakietu, 228  
deklaracja tablicy, 131  
dekrementacja, 59  
deskryptory modułów, 430  
domknięcia, 402  
domyślne metody interfejsów, 247  
dostawcy usług, 444  
dostęp  
    do pakietu, 231  
    do składowych, 159, 196, 230  
    do składowych klasy bazowej, 203  
    sekwencyjny, 296  
drabinka if-else-if, 74  
drzewo katalogów, 432  
dynamiczność, 27  
działanie programu javadoc, 561  
dziedziczenie, 29, 193, 196, 198, 223, 528  
    wielokrotne, 250

## E

efekty, 508  
efektywność, 27  
eksport kwalifikowany, 438  
eksportowanie do modułu, 438

**F**

falszywe przebudzenie, 337

**G**

generowanie

powtórne wyjątku, 264

wyjątku, 264

główny obszar roboczy, 486

graf sceny, 487

graficzny interfejs użytkownika, GUI, 485

**H**

hermetyzacja, 28

hierarchia

klas, 203

wyjątków, 256

hierarchie wielopoziomowe, 206

**I**

identyfikatory, 45

implementacje interfejsów, 237

import

pakietów, 234, 579

składowych statycznych, 364

informacje

o metodach domyślnych, 248

o modułach, 430

o usługach, 444

o wyrażeniach lambda, 402

inicjalizacja

dynamiczna zmiennej, 55

tablic wielowymiarowych, 130

zmiennej, 55

inkrementacja, 59

instrukcja

break, 90

continue, 95

exports, 435

goto, 91

if, 38, 41, 72

import, 234

opens, 451

requires, 435, 451

requires transitive, 440

return, 108, 109

switch, 75, 91, 145

instrukcje

przypisania, 64

skoku, 71

sterujące, 38, 71, 518

wyboru, 71

zagnieżdżone if, 73

zagnieżdżone switch, 78

interfejs, 29, 227, 235, 529

AutoCloseable, 290

BinaryFunc, 445

BinFuncProvider, 446

DataOutput, 292

ICharQ, 244

Queue, 241

Runnable, 314–318, 355

Throwable, 292

interfejsy

domyślne metody, 247

dziedziczenie, 246

funkcyjne, 402, 425

funkcyjne sparametryzowane, 409

metody prywatne, 252

metody statyczne, 251

obiektów nasłuchujących, 465

referencje, 240

sparametryzowane, 386

usługi, 445

Internet, 23

interpreter kodu bajtowego, 25

interpretowalność, 27

iteracje, 139

**J**

Java, 22

Control Panel, 570

SE 9, 17

Web Start, 563

podpis cyfrowy, 564

JavaFX, 485, 552

efekty, 508

klasa Application, 487

klasa Scene, 486

klasa Stage, 486

kompilacja aplikacji, 491

kontrolka Button, 494

kontrolka Label, 491

listy, 501

pola tekstowe, 505

pola wyboru, 497

przyciski, 493

- szkielet aplikacji, 488
- transformacje, 509
- uruchamianie aplikacji, 488
- uruchamianie programów, 491
- wątek aplikacji, 491
- zdarzenia, 493

JDK, Java Development Kit, 29

jednostka kompilacji, 31

język

- C, 22

- C++, 22

- Java, 22

JShell, 573

- dodanie metody, 576

- edycja kodu, 575

- importowanie pakietów, 579

- polecenia, 580

- ponowne wykonywanie kodu, 575

- przetwarzanie wyrażeń, 578

- stosowanie interfejsu, 577

- utworzenie klasy, 577

- wbudowane zmienne, 578

- wyjątki, 579

JVM, Java Virtual Machine, 25

## K

kierownica, 29

klasa, 28, 101, 520, 524

- AbsMinusProvider, 447

- AbsPlus, 446

- AbsPlusProvider, 447

- Application, 487

- ArithmeticException, 259

- Console, 297

- DataInputStream, 292

- DataOutputStream, 292

- Enuma, 353

- ExtBook, 233

- FailSoftArray, 163

- FileInputStream, 290

- FileOutputStream, 290

- FileReader, 303

- FileWriter, 302

- Help, 114

- InputStream, 283

- InputStreamReader, 298

- MyModAppDemo, 433, 448

- MyThread, 316, 344

- NonIntResultException, 272

- Object, 225

- OutputStream, 283

- PrintWriter, 301

- Priority, 329

- Queue, 134, 163, 174, 241

- sparametryzowana, 389

- wyjątki, 274

- Reader, 298

- Scale, 510

- Scene, 486

- ShowBits, 153

- SimpleMathFuncs, 433, 441

- Stage, 486

- String, 141

- Thread, 314, 321, 326

- Throwable, 265

- Triangle, 195

- TwoDShape, 217

- Vehicle, 119, 203

- Writer, 299

klasy

- abstrakcyjne, 220

- bazowe, 199, 209

- definiowanie, 102

- implementacji, 446

- komponentów Swing, 458

- opakowujące, 304–306, 310

- pochodne, 209, 321

- pochodne wyjątków, 272

- sparametryzowane, 376, 377

- strumieni bajtowych, 280

- strumieni znakowych, 280, 282

- wewnętrzne, 184

- zagnieżdżone, 184

- zdarzeń, 465

klauzula

- catch, 261

- else, 74

- finally, 267

- implements, 237

- throws, 268

kod bajtowy, 25

kolejka, 134, 163, 174, 241

komentarze

- dokumentacyjne, 557, 561

- jednowierszowe, 32

- wielowierszowe, 32

kompilacja, 491

kompilator

- ahead-of-time, 26

- javac, 31, 434

kompilowanie, 434  
 komponent, 457  
     JButton, 465  
     JCheckBox, 471  
     JList, 474  
     JTextField, 468  
 komunikacja międzywątkowa, 336  
 konstrukcja if-else-if, 74  
 konstruktory, 117, 198  
     domyślne, 211  
     przeciążanie, 173, 174  
     klasy Queue, 174  
     referencje, 423  
     sparametryzowane, 386  
     wywoływanie, 208  
     z parametrami, 118, 211  
 kontenery, 457, 458  
 kontroler, 457  
 kontrolka  
     Button, 494  
     Label, 491  
 konwersja  
     łańcuchów numerycznych, 304  
     typów, 64  
     typów w wyrażeniach, 68

## L

listy, 501  
     argumentów, 186  
 literały, 53  
     klasowe, 444  
     łańcuchowe, 54  
     szesnastkowe, 53

## Ł

łańcuchy  
     numeryczne, 304  
     wczytywanie, 300  
     znaków, 141

## M

magazyn  
     certyfikatów, 565  
     kluczy, 568  
 maszyna wirtualna Javy, JVM, 25  
 menedżery układu, 459  
 metadane, 366

metoda, 28, 33, 106, 108, 520, 524  
     area(), 196  
     changeColor(), 356  
     close(), 288  
     createAndStart(), 320  
     doubleValue(), 359  
     f(), 172  
     getMessage(), 367  
     getName(), 318  
     helpOn(), 113, 308  
     indexOK(), 163  
     init(), 487  
     isAlive(), 326  
     isValid(), 114  
     join(), 326  
     main(), 33  
     Math.sqrt(), 364  
     notify(), 336, 337  
     notifyAll(), 336  
     println(), 34, 37  
     readLine(), 301  
     show(), 215  
     showType(), 374  
     sqrt(), 109  
     start(), 320  
     substring(), 145  
     sumArray(), 331  
     valueOf(), 350  
     values(), 350  
     vaTest(), 187, 189  
     wait(), 336, 337, 356  
     write(), 284  
     waitForChange(), 356  
 metody  
     abstrakcyjne, 220  
     cyklu życia, 487  
     domyślne, 247–249  
     dostępowe, 211  
     instancyjne, 419  
 klasy  
     DataInputStream, 293  
     DataOutputStream, 293  
     InputStream, 283  
     Object, 225  
     OutputStream, 283  
     Reader, 298  
     Thread, 315  
     Throwable, 266  
     Writer, 299

- konwersji łańcuchów, 305
  - parsujące, 305
  - prywatne, 252
  - przeciążanie, 169, 189
  - przekazywanie argumentów, 165
  - przekazywanie obiektów, 164
  - przesłanie, 213, 215, 217
  - referencje, 417
  - rekurencyjne, 177
  - rodzime, 585
  - rozszerzeń, 247
  - sparametryzowane, 112, 384
  - statyczne, 180, 251
  - wytwórcze, 318
  - zwracanie obiektów, 167
- model, 457
- model-widok-kontroler, 457
- modularny plik JAR, 452
- moduł, 429, 547
  - appstart, 434
  - appsupport, 441
  - java.base, 436
  - java.desktop, 436
- moduły
  - otwarte, 451
  - nienazwane, 437
  - platformy, 436
- modyfikator
  - abstract, 220
  - private, 160, 163, 196, 230
  - protected, 160, 232
  - public, 160, 230
  - transient, 583
  - volatile, 583
- modyfikatory dostępu, 33, 160
- monitor, 331

## N

- nawiasy, 70
- niejawna
  - dostępność, 440
  - zależność, 440
- niejednoznaczność, 190, 396
- niezależność, 27
- niezawodność, 27
- NIO, 304

## O

- obiektowość, 27
- obiekty, 28, 102, 105, 520
  - klasy pochodnej, 209
  - nasłuchujące, 465
  - referencje, 105
- obsługa
  - błędów, 260
  - błędów składni, 34
  - plików, 302
  - wejścia i wyjścia, 279, 533
  - wyjatków, 255, 531
  - wyjścia konsoli, 301
  - zdarzeń, 464, 481, 493, 494
    - anonimowe klasy wewnętrzne, 481
    - wyrażenia lambda, 481
- obszar roboczy, 486
- odczyt
  - danych binarnych, 292
  - konsoli, 298
  - plików, 285
  - znaków, 299
- odnośnik do pliku JNLP, 566
- odstęp, 70
- odzyskiwanie pamięci, 120
- ogólna klasa akcji, 29
- ograniczenia
  - argumentów wieloznacznych, 382
  - dla składowych statycznych, 397
  - tablic sparametryzowanych, 398
  - typów, 377, 378
  - związane z wyjątkami, 399
- opakowywanie, 360
- operacje
  - na łańcuchach, 142
  - przypisania, 105
- operator, 47, 517, 521
  - ?, 155
  - AND, 149
  - diamentowy, 395
  - new, 120, 348
  - NOT, 151
  - OR, 149
  - XOR, 150
- operatory
  - arytmetyczne, 58
  - bitowe, 147
  - logiczne, 60
  - logiczne warunkowe, 62

operatory

- priorytet, 67
- przesunięcia, 151
- przypisania skrótowe, 63, 153
- relacyjne, 38, 60

**P**

pakiet, 160, 227, 529

- AWT, 456
- java.lang, 235
  - wyjątki, 271
- java.util.function, 425
- mypack, 229
- Swing, 455

pakiety

- definiowanie, 228
- import, 234
- Java API, 235
- JavaFX, 486
- wyszukiwanie, 229

panel

- szklany, 459
- warstw, 459
- zawartości, 459

panele kontenerów szczytowych, 459

parametr, 33, 110

parametryzacja, 375

pętla

- do-while, 86
- for, 40, 80, 81, 137
- for-each, 137
- rozszerzona for, 85, 140
- while, 85

pętle

- bez ciała, 83
- nieskończone, 83
- zagnieżdżone, 99
- zmiennie sterujące, 84

piaskownica, sandbox, 25

pierwszy program, 30, 32

- Swing, 461

plik

- AbsShape.java, 220
- AccessDemo.java, 160
- ACopy.java, 133
- AddMeth.java, 107
- ArrayDemo.java, 124
- ArrayErr.java, 126
- AssignARef.java, 131

- AutoBox2.java, 361
- AutoBox3.java, 362
- AvgNums.java, 305
- BlockLambdaDemo.java, 409
- Book.java, 231, 232
- BookDemo.java, 229
- BoolDemo.java, 51
- BoundsDemo.java, 378
- Break2.java, 90
- Break4.java, 92
- Break5.java, 93
- BreakDemo.java, 90
- BreakErr.java, 94
- Bubble.java, 126, 127
- ButtonDemo.jar, 568
- ButtonDemo.java, 466, 567
- ButtonDemo.jnlp, 570
- ByThrees.java, 239
- ByTwos.java, 237, 238
- CallByValue.java, 166
- CastDemo.java, 66
- CBDemo.java, 472
- CharArithDemo.java, 50
- CheckboxDemo.java, 498
- CLDemo.java, 146
- Comma.java, 81
- CompFiles.java, 295
- CompFuel.java, 112
- ConsDemo.java, 118
- ConstructorRefDemo.java, 423
- ContDemo.java, 95
- ContToLabel.java, 95
- CopyFile.java, 289
- CopyFile2.java, 291
- CustomExceptDemo.java, 272
- DecrFor.java, 81
- DefaultMethodDemo.java, 248
- DemoPwr.java, 121
- DtoL.java, 65
- DtoS.java, 303
- DWDemo.java, 86
- DynDispDemo.java, 215
- DynInit.java, 55
- DynShapes.java, 217
- EffectsAndTransformsDemo.java, 511
- Empty.java, 82
- Empty2.java, 83
- Encode.java, 150
- EnumDemo.java, 349
- EnumDemo2.java, 350



EnumDemo4.java, 353  
 ErrInfo.java, 168  
 Errmsg.java, 167  
 ExcDemo1.java, 257  
 ExcDemo2.java, 258  
 ExcDemo3.java, 260  
 ExcDemo4.java, 261  
 ExcDemo5.java, 262  
 ExcTypeMismatch.java, 259  
 ExtendThread.java, 322  
 FileHelp.java, 309  
 FinalD.java, 224  
 FindFac.java, 99  
 ForEach.java, 138  
 ForEach2.java, 139  
 ForTest.java, 82  
 ForVar.java, 84  
 PSDemo.java, 161  
 GalToLit.java, 37  
 GalToLitTable.java, 43  
 GenArrays.java, 398  
 GenDemo.java, 372  
 GenericFunctionalInterfaceDemo.java, 410  
 GenericMethodDemo.java, 384  
 GenIfDemo.java, 387  
 GenQDemo.java, 391  
 GenQueue.java, 390  
 Guess.java, 73  
 Guess2.java, 73  
 Guess3.java, 74  
 Guess4.java, 87  
 Help.java, 79  
 Help2.java, 89  
 Help3.java, 97  
 HelpClassDemo.java, 116  
 Hypot.java, 50  
 ICharQ.java, 241  
 IConstD.java, 245  
 IFExtend.java, 246  
 IGenQ.java, 389  
 Inches.java, 49  
 IncompatibleRef.java, 209  
 IQDemo.java, 244  
 JavaFXEventDemo.java, 495  
 JavaFXLabelDemo.java, 492  
 JavaFXSkel.java, 488  
 JoinThreads.java, 326  
 KbIn.java, 72  
 KtoD.java, 303  
 Ladder.java, 75  
 LambdaArgumentDemo.java, 413  
 LambdaDemo.java, 405  
 LambdaDemo2.java, 406  
 LambdaDemo3.java, 407  
 LambdaExceptionDemo.java, 416  
 LengthDemo.java, 132  
 ListDemo.java, 475  
 ListViewDemo.java, 503  
 LocalClassDemo.java, 185  
 LogicalOpTable.java, 67  
 LowCase.java, 149  
 LtoD.java, 65  
 MethodRefDemo.java, 418  
 MethodRefDemo2.java, 419  
 MethodRefDemo3.java, 421  
 MinMax.java, 125  
 MinMax2.java, 126  
 ModDemo.java, 59  
 MoreThreads.java, 324  
 MoreThreads2.java, 326  
 MultiCatch.java, 270  
 MyIF.java, 248  
 MyIF2.java, 251  
 MyIFImp.java, 248  
 MyIFImp2.java, 249  
 MyModAppDemo.java, 442  
 NestedClassDemo.java, 184  
 NestTrys.java, 263  
 NestVar.java, 58  
 NoBreak.java, 77  
 NoChange.java, 139  
 NotDemo.java, 151  
 NotHandled.java, 259  
 NoZeroDiv.java, 156  
 NoZeroDiv2.java, 156  
 OrderOfConstruction.java, 208  
 OverloadConsDemo.java, 173  
 OverloadDemo.java, 169  
 ParmDemo.java, 111  
 PassOb.java, 164  
 PassObjRef.java, 166  
 Phone.java, 147  
 Power.java, 85  
 PrintWriterDemo.java, 302  
 PriorityDemo.java, 329  
 PromDemo.java, 69  
 ProtectDemo.java, 233  
 QDemo.java, 134  
 QDemo2.java, 175  
 QExc.java, 389  
 QExcDemo.java, 275  
 QSDemo.java, 182, 183

## plik

Quadratic.java, 364  
 Quadratic2.java, 365  
 Queue.java, 163  
 Ragged.java, 129  
 RandomAccessDemo.java, 296  
 RawDemo.java, 393  
 ReadBytes.java, 284  
 ReadChars.java, 300  
 ReadLines.java, 301  
 Recursion.java, 177  
 RelLogOps.java, 61  
 RethrowDemo.java, 264  
 RetMeth.java, 109  
 RWData.java, 293  
 ScopeDemo.java, 56  
 SCops.java, 62  
 SDemo.java, 179  
 SDemo2.java, 180  
 SDemo3.java, 181  
 Series.java, 237  
 Series2.java, 250  
 Series3.java, 252  
 SeriesDemo.java, 238  
 SeriesDemo2.java, 240  
 Shapes.java, 194  
 Shapes2.java, 197  
 Shapes3.java, 198  
 Shapes4.java, 199  
 Shapes5.java, 201  
 Shapes6.java, 206  
 Shapes7.java, 211  
 ShiftDemo.java, 152  
 ShowBits.java, 149  
 ShowBitsDemo.java, 154  
 ShowFile.java, 286  
 ShowFile2.java, 287  
 ShowFile3.java, 290  
 SideEffects.java, 64  
 SimpleMathFuncs.java, 432  
 Sound.java, 52  
 SqrRoot.java, 80  
 Squares.java, 130  
 StaticError.java, 181  
 StrDemo.java, 54  
 StringArrays.java, 144  
 StringSwitch.java, 145  
 StrOps.java, 142  
 SubStr.java, 145  
 SumDemo.java, 174  
 SupportFuncs.java, 441

SupSubRef.java, 210  
 Suspend.java, 342  
 SwingDemo.java, 460  
 SwingFC.java, 479  
 SwitchDemo.java, 76  
 Sync.java, 331  
 Sync2.java, 334  
 TextFieldDemo.java, 506  
 TFDemo.java, 469  
 ThreadCom.java, 337  
 ThreadVariations.java, 319  
 ThrowDemo.java, 264  
 ThrowsDemo.java, 269  
 TrafficLigthDemo.java, 354, 356  
 Triangle.java, 196  
 TruckDemo.java, 204  
 TwoD.java, 128  
 TwoVehicles.java, 104  
 TypeConv.java, 170  
 TypeConv2.java, 171  
 UpCase.java, 148  
 Usebook.java, 232  
 UseBook2.java, 235  
 UseCast.java, 69  
 UseMain.java, 345  
 UsePredicateInterface.java, 425  
 UseSuper.java, 203  
 UseThreads.java, 315  
 UseThrowableMethods.java, 266  
 VarArgs.java, 187  
 VarArgs2.java, 188  
 VarArgs3.java, 189  
 VarArgs4.java, 190  
 VarCapture.java, 415  
 VarInitDemo.java, 57  
 VehConsDemo.java, 119  
 VehicleDemo.java, 103  
 WhileDemo.java, 85  
 WildcardDemo.java, 381  
 Wrap.java, 360

## pliki

.class, 228  
 automatyczne zamykanie, 290  
 definicji modułów, 448  
 HTML, 570  
 JAR, 564  
 JNLP, 566, 569  
 o dostępie swobodnym, 296  
 odczyt, 285  
 porównywanie, 295  
 system pomocy, 306  
 zapis, 285, 288

podpis cyfrowy, 564  
 pola
 

- tekstowe, 505
- wyboru, 497

 polecenia JShell, 580  
 polimorfizm, 29, 215  
 porównywanie plików, 295, 477  
 powtórne generowanie wyjątku, 269  
 predefiniowane interfejsy funkcyjne, 425  
 priorytety
 

- operatorów, 67
- wątków, 328

 programowanie
 

- aplikacji internetowych, 23
- obiektove, 27
- wielowątkowe, 313, 536

 programy wielowątkowe, 318  
 przechwytywanie zmiennych, 415  
 przeciążanie
 

- konstruktorów, 173
- metod, 169, 189

 przekazywanie
 

- argumentów, 165
- obiektów, 164

 przenośność, 25, 27  
 przesłanie metod, 213, 215, 217, 223  
 przestroga, 364  
 przestrzeń nazw, 228  
 przycisk, 493, 494  
 pusta instrukcja, 83

## R

referencje
 

- interfejsu, 240
- klasy bazowej, 209
- konstruktorów, 423
- metod, 401, 417, 544
- metod instancyjnych, 419
- obiektów, 105
- tablicy, 131

 rekurencja, 177  
 REPL, 573  
 rozproszoność, 27  
 rzutowanie typów, 65

## S

scena, 486  
 sekwencje specjalne, 53  
 selektor, 304

serwlet, 26  
 silnia, 177  
 składowa length, 132  
 składowe klasy, 28, 102, 159, 231
 

- chronione, 232
- statyczne, 179, 364, 538

 skrótove operatory przypisania, 63  
 słowo kluczowe, 44, 430
 

- catch, 256, 261
- class, 102
- extends, 246
- final, 223
- finally, 256, 267
- instanceof, 584
- provides, 444
- public, 33
- static, 33, 178
- strictfp, 584
- super, 199, 203
- this, 121, 585
- throw, 256
- throws, 256, 268
- transient, 583
- try, 256, 263
- uses, 444
- void, 33, 109
- volatile, 583
- with, 444

 sortowanie
 

- pęcherzykowe, 182
- tablicy, 126

 sparametryzowana klasa Queue, 389  
 sparametryzowane interfejsy funkcyjne, 409  
 specjalne sekwencje znaków, 53  
 stała, 53  
 stałe wylczeniowe, 348  
 sterowanie instrukcją switch, 145  
 strumienie
 

- bajtowe, 280, 282, 285
- predefiniowane, 281
- wejścia i wyjścia, 280
- znakowe, 280, 298–302
  - obsługa plików, 302
  - obsługa wyjścia konsoli, 301
  - odczyt konsoli, 298

 Swing, 455, 548
 

- komponent, 457
  - JButton, 465
  - JList, 474
  - JTextField, 468

Swing

- kontenery, 457
- obsługa zdarzeń, 464
- pierwszy program, 460

synchronizacja, 331

- instrukcji, 334
- metod, 331

system

- ósemkowy, 53
- plików, 567
- pomocy, 88, 96, 306
- pomocy jako klasa, 113

**Ś**

średnik, 42

**T**

tabela prawdy, 67

tablice, 33, 123

- deklaracja, 131
- dwuwymiarowe, 128
- jednowymiarowe, 124
- łańcuchów, 144
- nieregularne, 129
- referencje, 131
- sortowanie, 126
- sparametryzowane, 398
- wielowymiarowe, 130
  - inicjalizacja, 130
  - iteracje, 139

technologia Java Web Start, 563

terminologia, 27

transformacje, 509

tryb wielomodułowy, 439

tworzenie

- łańcuchów, 141
- wątku, 315
- wielu wątków, 323

typ danych, 35, 47, 517, 521

- boolean, 48, 51
- byte, 48
- char, 48
- double, 36, 48
- float, 36, 48
- int, 36, 48
- long, 48
- short, 48

typy

- całkowite, 48
- obiektywne, 375
- opakowujące, 359
- proste, 48
- sparametryzowane, 371, 397, 541
- surowe, 392
- wyliczeniowe, 353, 538
- zmiennoprzecinkowe, 49

**U**

układ, 487

- BorderLayout, 459
- BoxLayout, 459
- FlowLayout, 459
- GridBagLayout, 459
- GridLayout, 459
- SpringLayout, 459

uruchamianie aplikacji, 434

- JavaFX, 488, 491

usługi, 443

**W**

warunkowe operatory logiczne, 62

wątek, 313

- aplikacji, 491
- główny, 314, 344
- rozdziatu zdarzeń, 462

wątki

- kończenie działania, 326, 341
- priorytety, 328
- wstrzymywanie działania, 341
- wznawianie działania, 341

wcięcia, 42

wczytywanie łańcuchów, 300

węzły, 487

widok, 457

wielowątkowość, 27, 313

własne

- komponenty, 456
- typy, 348

wnioskowanie typów, 395

wprowadzanie znaków, 71

wskaźnik, 296

wyjątek, 255, 531, 579

- ArithmeticException, 260, 270
- ArrayIndexOutOfBoundsException, 257
- InterruptedException, 316

wyjątki  
 generowanie, 264  
 hierarchia, 256  
 klas pochodnych, 261  
 powtórne generowanie, 264  
 w klasie Queue, 274  
 wbudowane, 270  
 wylczenia, 348–351  
 wymagania przechodnie, 439  
 wymazywanie, 374, 396  
 wypakowywanie, 360  
 wyrażenia, 68, 362  
 konwersja typów, 68  
 lambda, 401, 405, 415, 544  
 blokowe, 408  
 jako argument, 411  
 zgłaszanie wyjątków, 416  
 wyszukiwanie pakietów, 229  
 wywołanie  
 konstruktora klasy bazowej, 199  
 konstruktorów, 208  
 programu, 146

## Z

zagnieżdżanie  
 bloków try, 263  
 instrukcji if, 73  
 instrukcji switch, 78  
 zapis  
 danych binarnych, 292  
 do wyjścia konsoli, 284  
 plików, 285  
 w pliku, 288  
 zarządzanie zasobami, 269  
 zasady promocji typów, 68  
 zasięg deklaracji, 56  
 zbiór znaków, 304  
 zdarzenia, 464, 493  
 zgłaszanie wyjątków, 416

znacznik  
 @author, 558  
 @deprecated, 558  
 @exception, 559  
 @hidden, 559  
 @param, 560  
 @provides, 560  
 @return, 560  
 @see, 560  
 @since, 560  
 @throws, 561  
 @uses, 561  
 @version, 561  
 {@code}, 558  
 {@docRoot}, 559  
 {@index}, 559  
 {@inheritDoc}, 559  
 {@link}, 559  
 {@linkplain}, 560  
 {@literal}, 560  
 {@value}, 561  
 zmienna środowiskowa CLASSPATH, 229  
 zmienne, 34, 55  
 instancji, 351  
 iteracyjne, 137  
 liczby argumentów, 186, 190  
 static, 179  
 sterujące, 84  
 w interfejsach, 245  
 znaczniki javadoc, 557  
 znaki, 50  
 zwracanie  
 obiektów, 167  
 wartości, 109

## Ź

źródło zdarzenia, 464

## Notatki

# PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion** 

## 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 przejrzysto, 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 stały 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.

 <b>hellon.pl</b>	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	<b>KOD KORZYŚCI</b> Słęgnij po więcej! ▶  ISBN 978-83-283-4611-6  9 788328 346116
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 89,00 zł

