



Kotlin

w akcji

Dmitry Jemerov, Svetlana Isakova

Helion 

Tytuł oryginału: Kotlin in Action

Tłumaczenie: Andrzej Watrak

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-4720-5

Original edition copyright © 2017 by Manning Publications Co.
All rights reserved.

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

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 Helion SA 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
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

Printed in Poland.

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

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

Spis treści

Przedmowa	11
Wstęp	13
Podziękowania	15
O książce	17
O autorach	21
CZĘŚĆ I WPROWADZENIE DO JĘZYKA KOTLIN.....	23
Rozdział 1. Kotlin: co to jest i po co?	25
1.1. Przedsmak Kotlin	25
1.2. Najważniejsze cechy języka Kotlin	26
1.2.1. Docelowe platformy: serwery, Android i wszystko, gdzie jest Java	26
1.2.2. Statyczne typowanie danych	27
1.2.3. Programowanie funkcyjne i obiektowe	28
1.2.4. Bezpłatne i otwarte oprogramowanie	29
1.3. Zastosowania języka Kotlin	30
1.3.1. Kotlin na serwerach	30
1.3.2. Kotlin w Androidzie	31
1.4. Filozofia języka Kotlin	32
1.4.1. Pragmatyzm	32
1.4.2. Zwięzłość	33
1.4.3. Bezpieczeństwo	34
1.4.4. Kompatybilność	35
1.5. Narzędzia języka Kotlin	35
1.5.1. Kompilator kodu	36
1.5.2. Wtyczki dla IntelliJ IDAE i Android Studio	36
1.5.3. Interaktywna powłoka	37
1.5.4. Wtyczka dla Eclipse	37
1.5.5. Internetowy „plac zabaw”	37
1.5.6. Konwerter Java-Kotlin	37
1.6. Podsumowanie	38
Rozdział 2. Podstawy języka Kotlin	39
2.1. Podstawowe elementy: funkcje i zmienne	39
2.1.1. Witaj, świecie!	40
2.1.2. Funkcje	40
2.1.3. Zmienne	42
2.1.4. Proste formatowanie ciągów znaków: szablony	43

2.2.	Klasy i właściwości	44
2.2.1.	Właściwości	45
2.2.2.	Własne metody dostępowe	47
2.2.3.	Układ kodu źródłowego: katalogi i pakiety	48
2.3.	Kodowanie i dokonywanie wyborów: klasa wyliczeniowa i wyrażenie when	49
2.3.1.	Deklarowanie klasy wyliczeniowej	49
2.3.2.	Wyrażenie when i klasy wyliczeniowe	50
2.3.3.	Wyrażenie when i dowolne obiekty	51
2.3.4.	Wyrażenie when bez argumentów	52
2.3.5.	Inteligentne rzutowanie: połączenie sprawdzania i rzutowania typów	53
2.3.6.	Refaktoryzacja kodu: zamiana if na when	55
2.3.7.	Bloki kodu w odgąszeniach wyrażen if i when	56
2.4.	Iteracje: pętle while i for	57
2.4.1.	Pętla while	57
2.4.2.	Iteracje liczb: zakresy i postępy	57
2.4.3.	Iterowanie elementów map	59
2.4.4.	Sprawdzanie przynależności do zakresu i kolekcji za pomocą słowa in	60
2.5.	Wyjątki w Kotlinie	61
2.5.1.	Instrukcje try, catch i finally	62
2.5.2.	Słowo kluczowe try jako wyrażenie	63
2.6.	Podsumowanie	64
Rozdział 3. Definiowanie i wywoływanie funkcji		65
3.1.	Tworzenie kolekcji	66
3.2.	Łatwiejsze wywoływanie funkcji	67
3.2.1.	Nazwane argumenty	68
3.2.2.	Domyślne wartości argumentów	68
3.2.3.	Koniec ze statycznymi klasami pomocniczymi, czyli funkcje i właściwości najwyższego poziomu	70
3.3.	Dodawanie elementów do zewnętrznych klas: funkcje i właściwości rozszerzające	72
3.3.1.	Importowanie klas a funkcje rozszerzające	73
3.3.2.	Wywoływanie funkcji rozszerzających w kodzie Java	74
3.3.3.	Funkcje pomocnicze jako rozszerzenia	74
3.3.4.	Nienadpiszalność funkcji rozszerzających	75
3.3.5.	Właściwości rozszerzające	76
3.4.	Przetwarzanie kolekcji: funkcjonalność varargs, wywołania infix i obsługa bibliotek	77
3.4.1.	Rozbudowa interfejsu API kolekcji Java	78
3.4.2.	Deklarowanie funkcji o dowolnej liczbie argumentów	78
3.4.3.	Działania w parach: wywołania infix i deklaracje destrukuryzujące	79
3.5.	Operacje na ciągach znaków i wyrażeniach regularnych	80
3.5.1.	Dzielenie ciągów znaków	81
3.5.2.	Wyrażenia regularne i potrójne cudzysłowy	81
3.5.3.	Potrójne cudzysłowy i wielowierszowe ciągi znaków	83
3.6.	Wyglądanie kodu: lokalne funkcje i rozszerzenia	84
3.7.	Podsumowanie	86

Rozdział 4. Klasy, obiekty i interfejsy	89
4.1. Definiowanie hierarchii klas	90
4.1.1. <i>Interfejsy w Kotlinie</i>	90
4.1.2. <i>Modyfikatory open, final (domyślny) i abstract</i>	92
4.1.3. <i>Modyfikatory widoczności, domyślny public</i>	94
4.1.4. <i>Klasy wewnętrzne i zagnieżdżone (domyślnie)</i>	96
4.1.5. <i>Klasy zapieczętowane: definiowanie ograniczonych hierarchii klas</i>	98
4.2. Deklarowanie klas z nietrywialnymi konstruktorami i właściwościami	100
4.2.1. <i>Inicjowanie klas: konstruktor główny i bloki inicjatora</i>	100
4.2.2. <i>Konstruktory dodatkowe i różne sposoby inicjowania klas nadrzędnych</i>	102
4.2.3. <i>Implementowanie właściwości zadeklarowanych w interfejsie</i>	104
4.2.4. <i>Dostęp do pól za pomocą getterów i setterów</i>	105
4.2.5. <i>Zmianianie widoczności metody dostępowej</i>	106
4.3. Metody generowane przez kompilator, klasy danych i delegowanie klas	108
4.3.1. <i>Metody uniwersalnych obiektów</i>	108
4.3.2. <i>Klasy danych i automatyczne generowanie uniwersalnych metod</i>	111
4.3.3. <i>Delegowanie klas i słowo kluczowe by</i>	112
4.4. Słowo kluczowe object łączące deklarację klasy z utworzeniem jej instancji	114
4.4.1. <i>Łatwe tworzenie singletonów poprzez deklarowanie obiektów</i>	114
4.4.2. <i>Obiekty towarzyszące: miejsce dla metod wytwórczych i elementów statycznych</i>	116
4.4.3. <i>Obiekty towarzyszące jako zwykłe obiekty</i>	118
4.4.4. <i>Wyrażenia obiektowe, czyli anonimowe klasy wewnętrzne</i>	121
4.5. Podsumowanie	122
Rozdział 5. Wyrażenia lambda	123
5.1. Wyrażenia lambda i odwołania do elementów obiektów	123
5.1.1. <i>Wprowadzenie do wyrażen lambda: bloki kodu jako argumenty funkcji</i>	124
5.1.2. <i>Lambdy i kolekcje</i>	125
5.1.3. <i>Składnia wyrażenia lambda</i>	126
5.1.4. <i>Odwołania do zmiennych w bieżącym kontekście</i>	129
5.1.5. <i>Odwołania do elementów klas</i>	131
5.2. Interfejsy funkcyjne do przetwarzania kolekcji	133
5.2.1. <i>Podstawy: filtry i mapy</i>	133
5.2.2. <i>Warunki i funkcje all(), any(), count() oraz find() w kolekcjach</i>	135
5.2.3. <i>Funkcja groupBy() i konwersja listy na mapę grup</i>	136
5.2.4. <i>Funkcja flatMap(), spłaszczanie struktury danych i przetwarzanie zagnieżdżonych kolekcji</i>	137
5.3. Leniwe operacje na kolekcjach oraz sekwencje	138
5.3.1. <i>Pośrednie i końcowe operacje na sekwencjach</i>	139
5.3.2. <i>Tworzenie sekwencji</i>	142
5.4. Interfejsy funkcyjne Java	143
5.4.1. <i>Umieszczanie wyrażen lambda w argumentach metod Java</i>	144
5.4.2. <i>Konstruktory SAM i jawna konwersja wyrażen lambda na interfejsy funkcyjne</i>	146

5.5.	Wyrażenia lambda, odbiorniki oraz funkcje with() i apply()	147
5.5.1.	Funkcja with()	147
5.5.2.	Funkcja apply()	149
5.6.	Podsumowanie	151
Rozdział 6. System typów danych		153
6.1.	Zerowalność typów danych	153
6.1.1.	Zerowalne typy danych	154
6.1.2.	Znaczenie typów danych	156
6.1.3.	Bezpieczny operator wywołania „?”	157
6.1.4.	Operator Elvisa „?:”	158
6.1.5.	Bezpieczne rzutowanie typów: operator „as?”	160
6.1.6.	Asercja niezeraowa „!!”	161
6.1.7.	Funkcja let()	163
6.1.8.	Właściwości inicjowane z opóźnieniem	164
6.1.9.	Rozszerzenia typów zerowalnych	166
6.1.10.	Zerowalność argumentów typowanych	167
6.1.11.	Zerowalność typów i Java	168
6.2.	Typy proste oraz inne typy podstawowe	172
6.2.1.	Typy proste Int, Boolean i inne	172
6.2.2.	Zerowalne typy proste Int?, Boolean? i inne	173
6.2.3.	Przekształcanie liczb	174
6.2.4.	Typy główne „Any” i „Any?”	176
6.2.5.	Typ Unit, odpowiednik „void”	177
6.2.6.	Typ Nothing, czyli „funkcja nigdy nie kończy działania”	178
6.3.	Kolekcje i tablice	178
6.3.1.	Zerowalność typów danych i kolekcje	178
6.3.2.	Kolekcje tylko do odczytu i kolekcje mutowalne	181
6.3.3.	Kolekcje w Kotlinie i w Javie	182
6.3.4.	Kolekcje jako typy platformowe	184
6.3.5.	Tablice obiektów i typów prostych	186
6.4.	Podsumowanie	189
CZĘŚĆ II WZBOGACANIE KOTLINA.....		191
Rozdział 7. Przeciążanie operatorów oraz inne konwencje		193
7.1.	Przeciążanie operatorów arytmetycznych	194
7.1.1.	Przeciążanie dwuargumentowych operatorów arytmetycznych	194
7.1.2.	Przeciążanie złożonych operatorów przypisania	196
7.1.3.	Przeciążanie operatorów jednoargumentowych	198
7.2.	Przeciążanie operatorów porównania	199
7.2.1.	Operatory równości	199
7.2.2.	Przeciążanie operatorów nierówności: metoda compareTo()	200
7.3.	Konwencje stosowane w kolekcjach i zakresach	202
7.3.1.	Dostęp do elementu za pomocą indeksu, metod get() i set()	202
7.3.2.	Konwencja operatora in	203
7.3.3.	Metoda rangeTo()	204
7.3.4.	Konwencja „iterator” w pętli loop	205

7.4.	Deklaracje destrukuryzujące i metody komponentowe	206
7.4.1.	<i>Deklaracje destrukuryzujące i pętle</i>	207
7.5.	Współdzielenie metod dostępowych i delegowanie właściwości	208
7.5.1.	<i>Podstawy delegowania właściwości</i>	209
7.5.2.	<i>Korzystanie z delegowanych właściwości: inicjalizacja z opóźnieniem i funkcja lazy</i>	209
7.5.3.	<i>Implementacja delegowanych właściwości</i>	211
7.5.4.	<i>Zasady translacji delegowanych właściwości</i>	215
7.5.5.	<i>Przechowywanie wartości właściwości w mapie</i>	215
7.5.6.	<i>Delegowane właściwości w bibliotekach</i>	216
7.6.	Podsumowanie	218
Rozdział 8. Funkcje wysokopoziomowe: wyrażenia lambda jako argumenty oraz wyniki		219
8.1.	Deklarowanie funkcji wysokopoziomowych	220
8.1.1.	<i>Typy funkcyjne</i>	220
8.1.2.	<i>Wywoływanie funkcji podanych w argumentach</i>	221
8.1.3.	<i>Stosowanie typów funkcyjnych w kodzie Java</i>	222
8.1.4.	<i>Wartość domyślna i wartość null w argumentach typów funkcyjnych</i>	223
8.1.5.	<i>Funkcje zawierające w wynikach inne funkcje</i>	226
8.1.6.	<i>Usuwanie duplikatów kodu za pomocą wyrażeń lambda</i>	227
8.2.	Funkcje śródwierszowe i wydajność wyrażeń lambda	229
8.2.1.	<i>Wstawianie kodu funkcji</i>	230
8.2.2.	<i>Ograniczenia funkcji śródwierszowych</i>	232
8.2.3.	<i>Wstawianie operacji na kolekcjach</i>	233
8.2.4.	<i>Kiedy należy stosować funkcje śródwierszowe</i>	234
8.2.5.	<i>Zarządzanie zasobami za pomocą śródwierszowych wyrażeń lambda</i>	234
8.3.	Sterowanie realizacją kodu w funkcjach wysokopoziomowych	236
8.3.1.	<i>Instrukcja return w wyrażeniach lambda: wyjście z nadrzędnej funkcji</i>	236
8.3.2.	<i>Wyjście z wyrażenia lambda: instrukcja return z etykietą</i>	237
8.3.3.	<i>Funkcje anonimowe i domyślne wyjścia lokalne</i>	239
8.4.	Podsumowanie	240
Rozdział 9. Typy generyczne		241
9.1.	Generyczne argumenty typowane	242
9.1.1.	<i>Generyczne funkcje i właściwości</i>	243
9.1.2.	<i>Deklarowanie klas generycznych</i>	244
9.1.3.	<i>Ograniczenia argumentów typowanych</i>	245
9.1.4.	<i>Deklarowanie niezzerowalnego argumentu typowanego</i>	247
9.2.	Typy generyczne w działającym kodzie, wymazane i urzeczowione argumenty typowane	248
9.2.1.	<i>Typy generyczne w działającym kodzie: sprawdzanie i rzutowanie typów</i>	248
9.2.2.	<i>Deklarowanie funkcji z urzeczowionymi argumentami typowanymi</i>	250
9.2.3.	<i>Zastępowanie odwołań do klas urzeczowionymi argumentami typowanymi</i>	252
9.2.4.	<i>Ograniczenia urzeczowionych argumentów typowanych</i>	253

9.3.	Wariancje, typy generyczne i podtypy	254
9.3.1.	<i>Idea wariancji i umieszczanie wartości w argumentach funkcji</i>	254
9.3.2.	<i>Klasy, typy i podtypy</i>	255
9.3.3.	<i>Kowariancja: zachowanie zależności między podtypami</i>	257
9.3.4.	<i>Kontrawariancja: odwrotna zależność podtypów</i>	261
9.3.5.	<i>Wariancja typu w miejscu deklaracji</i>	263
9.3.6.	<i>Projekcja z gwiazdką: symbol * zamiast argumentu typowanego</i>	266
9.4.	Podsumowanie	270
Rozdział 10. Adnotacje i refleksja		271
10.1.	Deklarowanie i stosowanie adnotacji	272
10.1.1.	<i>Stosowanie adnotacji</i>	272
10.1.2.	<i>Adres adnotacji</i>	273
10.1.3.	<i>Dostosowywanie procesu serializacji JSON za pomocą adnotacji</i>	275
10.1.4.	<i>Deklarowanie adnotacji</i>	277
10.1.5.	<i>Metaadnotacje: kontrolowanie procesu przetwarzania adnotacji</i>	277
10.1.6.	<i>Klasy jako argumenty adnotacji</i>	278
10.1.7.	<i>Klasy generyczne jako argumenty adnotacji</i>	279
10.2.	Refleksja: badanie obiektów w trakcie działania kodu	280
10.2.1.	<i>Interfejs API refleksji w Kotlinie: interfejsy KClass, KCallable, KFunction i KProperty</i>	281
10.2.2.	<i>Serializacja obiektów z wykorzystaniem refleksji</i>	285
10.2.3.	<i>Dostosowywanie serializacji za pomocą adnotacji</i>	286
10.2.4.	<i>Analiza danych JSON i deserializacja obiektów</i>	289
10.2.5.	<i>Ostatni etap deserializacji: wywołanie metody callBy() i utworzenie obiektu za pomocą refleksji</i>	293
10.3.	Podsumowanie	297
Rozdział 11. Definiowanie języka DSL		299
11.1.	Od interfejsu API do języka DSL	300
11.1.1.	<i>Idea języków domenowych</i>	301
11.1.2.	<i>Wewnętrzny język DSL</i>	302
11.1.3.	<i>Struktura języka DSL</i>	303
11.1.4.	<i>Generowanie kodu HTML za pomocą wewnętrznego języka DSL</i>	304
11.2.	Tworzenie strukturalnego interfejsu API: wyrażenia lambda z odbiornikami w języku DSL	305
11.2.1.	<i>Wyrażenie lambda z odbiornikiem i typ funkcyjny rozszerzający</i>	305
11.2.2.	<i>Wyrażenia lambda z odbiornikami w generatorze HTML</i>	309
11.2.3.	<i>Generatory w Kotlinie, abstrakcje i powtarzalny kod</i>	313
11.3.	Bardziej elastyczne zagnieżdżanie bloków kodu dzięki konwencji invoke	316
11.3.1.	<i>Konwencja invoke, czyli obiekty wywoływane tak jak funkcje</i>	316
11.3.2.	<i>Konwencja invoke i typy funkcyjne</i>	317
11.3.3.	<i>Konwencja invoke w języku DSL: deklarowanie zależności w narzędziu Gradle</i>	318
11.4.	Język DSL w praktyce	319
11.4.1.	<i>Łączenie wywołań infix i asercja should w platformach testowych</i>	319
11.4.2.	<i>Rozszerzenia typów prostych i przetwarzanie dat</i>	321

11.4.3. <i>Funkcje rozszerzające i wewnętrzny język DSL do obsługi zapytań SQL</i>	322
11.4.4. <i>Biblioteka Anko i dynamiczne tworzenie interfejsu użytkownika w systemie Android</i>	325
11.5. Podsumowanie	327
DODATKI	329
<i>Dodatek A Kompilowanie projektów Kotlin</i>	331
A.1. Kompilowanie kodu Kotlin za pomocą narzędzia Gradle	331
A.1.1. <i>Kompilowanie za pomocą narzędzia Gradle aplikacji dla systemu Android</i>	332
A.1.2. <i>Kompilowanie projektów wykorzystujących adnotacje</i>	332
A.2. Kompilowanie kodu Kotlin za pomocą narzędzia Maven	333
A.3. Kompilowanie kodu Kotlin za pomocą narzędzia Ant	333
<i>Dodatek B Dokumentowanie kodu Kotlin</i>	335
B.1. Umieszczanie komentarzy dokumentacyjnych	335
B.2. Generowanie dokumentacji interfejsu API	336
<i>Dodatek C Ekosystem Kotlin</i>	339
C.1. Testowanie kodu	339
C.2. Wstrzykiwanie zależności	340
C.3. Serializacja JSON	340
C.4. Klienci HTTP	340
C.5. Aplikacje WWW	340
C.6. Operacje na bazach danych	341
C.7. Narzędzia i struktury danych	341
C.8. Aplikacje stacjonarne	341
<i>Skorowidz</i>	343

Kotlin: co to jest i po co?

1

Ten rozdział opisuje:

- podstawy języka Kotlin,
- najważniejsze cechy języka,
- tworzenie aplikacji serwerowych i aplikacji dla systemu Android,
- różnice w porównaniu z innymi językami,
- tworzenie i uruchamianie kodu.

Czym jest Kotlin? To nowy język programowania przeznaczony dla platformy JVM. Jest zwięzły, pragmatyczny i ukierunkowany na współpracę z kodem Java. Można go stosować niemal wszędzie tam, gdzie dzisiaj jest używana Java, tj. do tworzenia aplikacji serwerowych, aplikacji dla systemu Android i w wielu innych miejscach. Kod w języku Kotlin doskonale współpracuje ze wszystkimi bibliotekami i platformami języka Java i jest równie wydajny. W tym rozdziale szczegółowo opisujemy najważniejsze cechy Kotlin.

1.1. Przedsmak Kotlin

Zacznijmy od prostego kodu pokazującego, czym jest Kotlin. Kod ten definiuje klasę `Person` (osoba), kolekcję klas, wyszukuje najstarszą osobę i wyświetla wynik. Nawet w tym niewielkim przykładzie odkryjesz wiele ciekawych cech języka. Opiszemy dokładniej kilka z nich, aby były zrozumiałe w dalszej części książki. Kod jest opisany bardzo ogólnie, nie przejmuj się więc, jeżeli nie wszystko będzie od razu zrozumiałe. W dalszej części rozdziału szczegółowo opiszemy wszystkie zawiłości.

Aby wypróbować poniższy przykład, najłatwiej uruchomić go w internetowym „placu zabaw” na stronie <http://try.kotl.in>. Wpisz na niej kod z listingu 1.1 i uruchom, klikając przycisk *Run*.

Listing 1.1. Przedsmak Kotlina

```
data class Person(val name: String, ← Klasa „danych”.
                  val age: Int? = null) ← Typ zerowalny (Int?) i domyślna
fun main(args: Array<String>) { ← Główna funkcja.      wartość argumentu.
    val persons = listOf(Person("Alicja"),
                          Person("Bartek", age = 29)) ← Nazwany argument.
    val oldest = persons.maxBy { it.age ?: 0 } ← Wyrażenie lambda
    println("Najstarsza osoba: $oldest") ← Znakowy szablon.   i operator Elvisa.
}
// Najstarsza osoba: Person(name=Bartek, age=29) ← Automatyczne użycie metody toString().
```

Zadeklarowałeś prostą klasę z dwiema właściwościami: `name` (imię) i `age` (wiek). Jeżeli wartość właściwości `age` nie zostanie określona, wtedy przyjmie ona domyślną wartość `null`. Tworząc listę klas, nie określiłeś wieku Alicji, więc została użyta domyślna wartość. Później użyłeś metody `maxBy()` do wyszukania najstarszej osoby. Wyrażenie lambda umieszczone w argumencie tej metody ma jeden parametr o domyślnej nazwie `it` (to). Operator Elvisa (`?:`) zwraca wartość `0`, jeżeli właściwość `age` ma wartość `null`. Ponieważ nie określiłeś wieku Alicji, operator użył wartości `0`, zatem za najstarszą osobę został uznany Bartek.

Podoba Ci się to? Czytaj więc dalej i zostań ekspertem od Kotlina. Mamy nadzieję, że wkrótce taki kod będziesz stosował w swoich projektach, nie tylko podczas lektury tej książki.

1.2. Najważniejsze cechy języka Kotlin

Prawdopodobnie masz już wyobrażenie, czym jest język Kotlin. Przyjrzyjmy się teraz dokładniej jego najważniejszym cechom. Najpierw sprawdźmy, jakiego rodzaju aplikacje można pisać w tym języku.

1.2.1. Docelowe platformy: serwery, Android i wszystko, gdzie jest Java

Kotlin ma być przede wszystkim bardziej zwięzłą, skuteczniejszą i bezpieczniejszą alternatywą dla Javy. Powinien dać się stosować wszędzie tam, gdzie jest Java, która jest niezwykle popularnym językiem, wykorzystywanym w szerokim spektrum środowisk, od inteligentnych kart (Java Card) do największych centrów danych działających w serwisach Google, Twitter, LinkedIn oraz w dużych firmach internetowych. W większości powyższych przypadków programiści mogą za pomocą języka Kotlin osiągać swoje cele, pisząc krótszy kod i unikając dzięki temu kłopotów.

Największe obszary zastosowań języka Kotlin to:

- aplikacje serwerowe (zazwyczaj WWW),
- aplikacje przenośne dla systemu Android.

Język Kotlin można stosować również w innych sytuacjach. Na przykład można uruchamiać kod w systemie macOS, wykorzystując silnik Multi-OS Engine (<https://software.intel.com/en-us/multi-os-engine>), a za pomocą platform TornadoFX (<https://github.com/edvin/tornadofx>) i JavaFX¹ można tworzyć aplikacje dla komputerów stacjonarnych.

Ponadto, w odróżnieniu od Javy, kod Kotlin można przekształcać w kod JavaScript i uruchamiać w przeglądarce. Jednak w chwili powstawania tej książki wciąż trwało w JetBrains rozpoznawanie obsługi języka JavaScript oraz tworzenie prototypu konwertera, dlatego temat ten nie jest opisany w tej książce. Rozważamy też przystosowanie przyszłych wersji języka do nowych platform.

Jak widzisz, obszar zastosowania Kotlin jest bardzo szeroki. Język ten nie skupia się na jednej domenie ani nie próbuje rozwiązać określonego problemu, z którym mierzą się dziś programiści. Wręcz przeciwnie, posiada wiele udoskonaleń usprawniających realizację zadań, które pojawiają się w procesie tworzenia oprogramowania. Jest doskonale zintegrowany z istniejącymi bibliotekami do różnych zastosowań i zagadnień programistycznych. Przyjrzyjmy się teraz kluczowym cechom wyróżniającym ten język programowania.

1.2.2. Statyczne typowanie danych

Kotlin, podobnie jak Java, jest językiem wykorzystującym **statyczne typowanie** danych. Oznacza to, że typ każdego wyrażenia jest znany na etapie kompilacji kodu, a kompilator sprawdza, czy istnieją metody i pola w obiektach, do których odwołuje się kod. Tym Kotlin różni się od języków z **dynamicznym typowaniem**, takich jak Groovy i JRuby w środowisku JVM. W tych językach można definiować zmienne oraz funkcje przechowujące i zwracające dane dowolnych typów, a także identyfikować metody i pola w trakcie działania kodu. Dzięki temu kod jest krótszy i bardziej elastyczny pod względem tworzenia struktur danych. Problem polega jednak na tym, że podczas kompilacji kodu nie są wykrywane pomyłki w nazwach identyfikatorów, co skutkuje błędnym niekiedy działaniem kodu.

W Kotlinie, inaczej niż w Javie, nie trzeba jawnie określać typu każdej zmiennej. W wielu przypadkach typ jest określany automatycznie na podstawie kontekstu, dzięki czemu można pomijać deklarację typu. Ilustruje to poniższy najprostszy przykład:

```
val x = 1
```

Deklarowana jest tu zmienna. Ponieważ jest ona inicjowana liczbą całkowitą, Kotlin automatycznie przyjmuje typ `Int`. Określanie przez kompilator typu danych na podstawie kontekstu nosi nazwę **domniemania typów**.

Statycznie typowany kod ma m.in. następujące zalety:

- *Wydajność* — metody wywoływane są szybciej, ponieważ w trakcie działania kodu nie trzeba wyszukiwać wywoływanej metody.
- *Niezawodność* — kompilator sprawdza poprawność kodu, dzięki czemu zmniejsza się prawdopodobieństwo jego awarii po uruchomieniu.

¹ Patrz: *JavaFX: Getting Started with JavaFX*, Oracle, <http://mng.bz/500y>.

- *Łatwość utrzymania* — praca z nieznanym kodem jest łatwiejsza, ponieważ wiadomo, jakiego rodzaju obiekty są w nim stosowane.
- *Dostępność narzędzi* — można korzystać z niezawodnej refaktoryzacji, precyzyjnego uzupełniania kodu oraz innych funkcjonalności środowiska IDE.

Dzięki domniemaniu typów zmniejsza się rozwlekłość kodu, charakterystyczna dla statycznego typowania, ponieważ nie trzeba jawnie deklorować wszystkich typów.

Jeżeli przyjrzyysz się specyfikacji typów w języku Kotlin, znajdziesz w niej znane pojęcia. Klasy, interfejsy i typy generyczne są bardzo podobne do stosowanych w Javie. Zatem swoją wiedzę o tym języku możesz w dużej mierze przenieść do Kotlinia.

Są też nowe idee. Najważniejsze z nich to **typy zerowalne**, dzięki którym można tworzyć stabilniejsze programy, w których już na etapie kompilacji można wykrywać wskaźniki o wartościach null. Do tych typów powrócimy w dalszej części rozdziału, a dokładniej opiszemy je w rozdziale 6.

Kolejną nowością w systemie typów są **typy funkcyjne**. Aby dowiedzieć się, czym one są i jak są obsługiwane w języku Kotlin, przypomnijmy sobie najważniejsze zasady programowania funkcyjnego.

1.2.3. Programowanie funkcyjne i obiektowe

Jako programista Java z pewnością znasz kluczowe pojęcia z dziedziny programowania obiektowego, natomiast programowanie funkcyjne może być dla Ciebie nowością. Poniżej wymienione są najważniejsze terminy stosowane w programowaniu funkcyjnym:

- *Funkcje pierwszej klasy* — funkcje (ciągi operacji) traktuje się jak wartości. Można je zapisywać w zmiennych, umieszczać w argumentach i zwracać w wynikach innych funkcji.
- *Niezmiennosc* — obiekty są niezmiennie, tzn. po utworzeniu nie można zmieniać ich stanu.
- *Brak efektów ubocznych* — wykorzystywane są czyste funkcje zwracające te same wyniki dla tych samych wartości argumentów, nie można również zmieniać stanu innych obiektów ani komunikować się z kodem zewnętrznym.

Jakie korzyści płyną z programowania funkcyjnego? Po pierwsze, kod jest zwięzły. Jest krótszy i bardziej elegancki od kodu imperatywnego, ponieważ traktując funkcje jak wartości, lepiej wykorzystuje się siłę abstrakcji i unika duplikacji kodu.

Wyobraź sobie, że masz dwa podobne fragmenty kodu realizujące podobne zadania (na przykład wyszukiwanie określonych elementów kolekcji), jednak różniące się szczegółami (np. sposobem identyfikowania elementów). W takim wypadku wspólną logikę obu części można łatwo zapisać w postaci funkcji i umieścić ją w argumentach dwóch innych funkcji. Argumenty są funkcjami, ale można je opisać zwięzłą składnią w postaci funkcji anonimowych, tzw. **wyrażeń lambda**, jak niżej:

```
fun findAlicja() = findPerson { it.name == "Alicja" }
fun findBartek() = findPerson { it.name == "Bartek" }
```

Funkcja findPerson()
zawiera wspólny algorytm
wyszukiwania osoby.

←

Fragment w nawiasach klamrowych
opisuje osobę, którą chcemy znaleźć.

←

Inną zaletą kodu funkcyjnego jest **bezpieczeństwo międzywątkowe**. Jednym z głównych źródeł błędów w programach wielowątkowych jest modyfikowanie tych samych danych w różnych wątkach bez odpowiedniej synchronizacji. Jeżeli natomiast stosuje się niezmiennie struktury danych i czyste funkcje, można mieć pewność, że tego typu niebezpieczne modyfikacje nie będą się zdarzać, więc nie trzeba stosować skomplikowanych mechanizmów synchronizacyjnych.

Ponadto programowanie funkcyjne oznacza **prostsze testowanie** kodu. Funkcje niewywołujące efektów ubocznych można testować w izolacji bez konieczności tworzenia mnóstwa kodu konfiguracyjnego implementującego otoczenie, od którego testowany kod jest uzależniony.

Podsumowując, programowanie funkcyjne można stosować w każdym języku programowania, również w Javie, i w dużej mierze jest to dobry styl. Jednak nie wszystkie języki posiadają składnię i biblioteki umożliwiające łatwe stosowanie tego stylu. Na przykład nie było to możliwe w wersjach starszych niż Java 8. Kotlin oferuje bogaty zestaw funkcjonalności umożliwiających programowanie funkcyjne już na starcie. Są to między innymi:

- *Typy funkcyjne* umożliwiające umieszczanie funkcji w argumentach i wynikach innych funkcji.
- *Wyrażenia lambda* umożliwiające umieszczanie bloków kodu z minimalnym narzutem.
- *Klasy danych* umożliwiające tworzenie zwięzłych, niezmiennych obiektów.
- Standardowa biblioteka z bogatym zestawem *interfejsów API* do przetwarzania obiektów i kolekcji.

Kotlin umożliwia programowanie funkcyjne, ale go nie narzuca. Można w nim bez dodatkowego wysiłku tworzyć zmienne obiekty i funkcje z efektami ubocznymi, jeżeli jest taka potrzeba. Oczywiście korzystanie z platform opartych na hierarchii interfejsów i klas jest równie proste jak w Javie. Pisząc kod w języku Kotlin, można łączyć style programowania obiektowego i funkcyjnego oraz używać narzędzi najbardziej odpowiednich do rozwiązania danego problemu.

1.2.4. Bezpłatne i otwarte oprogramowanie

Środowisko Kotlin, obejmujące kompilator, biblioteki i narzędzia, jest w całości otwartym oprogramowaniem do bezpłatnego korzystania w dowolnych celach. Udostępniane jest w ramach licencji Apache 2, jego kod jest dostępny w serwisie GitHub (<http://github.com/jetbrains/kotlin>), a nowi członkowie społeczności są zawsze mile widziani. Do wyboru są trzy otwarte, w pełni obsługujące język Kotlin środowiska programistyczne: IntelliJ IDEA Community Edition, Android Studio i Eclipse (oczywiście IntelliJ IDEA Ultimate też).

Teraz, gdy wiesz już, jakiego rodzaju językiem jest Kotlin, poznaj jego zalety ujawniające się w wybranych zastosowaniach.

1.3. Zastosowania języka Kotlin

Jak wspomniałem wcześniej, dwa główne obszary zastosowań języka Kotlin to aplikacje serwerowe i aplikacje dla systemu Android. Przyjrzyjmy się teraz obu tym obszarom i sprawdźmy, dlaczego Kotlin tak dobrze się w nich sprawdza.

1.3.1. Kotlin na serwerach

Programowanie serwerów to bardzo szeroki temat, obejmujący między innymi następujące zagadnienia:

- aplikacje WWW generujące kody HTML stron,
- silniki aplikacji przenośnych udostępniające interfejs JSON API za pomocą protokołu HTTP,
- mikrousługi komunikujące się ze sobą za pomocą protokołu RPC.

Od wielu lat programiści tworzą tego rodzaju aplikacje w języku Java. W efekcie powstało mnóstwo platform i technologii ułatwiających programowanie. Zazwyczaj takie aplikacje nie są tworzone w odosobnieniu, od podstaw. Niemal zawsze istnieje jakiś system, który jest rozbudowywany, ulepszany lub wymieniany, a nowy kod integrowany jest z już istniejącymi elementami systemu, napisanego wiele lat wcześniej.

Pod tym względem ogromną zaletą Kotlina jest jego kompatybilność z istniejącym kodem Java. Niezależnie od tego, czy tworzy się nowy komponent, czy migruje istniejącą usługę, Kotlin nadaje się do tego celu znakomicie. Nie ma problemów z rozszerzaniem istniejących klas Java ani dodawaniem adnotacji metod i właściwości klas. Dodatkową korzyścią jest bardziej zwarty, stabilny i łatwiejszy w utrzymaniu kod.

Ponadto Kotlin oferuje kilka nowych technik ułatwiających programowanie. Na przykład za pomocą szablonów Builder można w zwięzły sposób tworzyć dowolne obiekty graficzne przy zachowaniu pełnej abstrakcyjności kodu i wykorzystaniu narzędzi do tworzenia jego powtarzalnych części.

Jednym z najprostszych przykładów wykorzystania tej cechy jest biblioteka generująca kod HTML, dzięki której kod zewnętrzny napisany w języku szablonowym można zastąpić zwięzłym kodem zachowującym zgodność typów, jak niżej:

```
fun renderPersonList(persons: Collection<Person>) =
    createHTML().table {
        for (person in persons) {
            tr {
                td { +person.name }
                td { +person.age }
            }
        }
    }
}
```

Można w ten sposób łatwo łączyć funkcje reprezentujące znaczniki HTML z typowymi konstrukcjami języka Kotlin. Nie trzeba stosować osobnego języka szablonowego z własną składnią. Wystarczy zastosować pętlę generującą kod HTML strony.

Innym obszarem, w którym stosuje się czyste, zwarte języki DSL (ang. *domain-specific language*, język dziedzinowy), są platformy obsługujące bazy danych. Na przykład platforma Exposed (<https://github.com/jetbrains/exposed>) oferuje czytelny język DSL umożliwiający tworzenie i wywoływanie zapytań SQL z pełnym zachowaniem kontroli typów danych. Poniżej przedstawiony jest przykład prostego kodu wykorzystującego tę funkcjonalność:

```
object CountryTable : IdTable() {
    val name = varchar("name", 250).uniqueIndex()
    val iso = varchar("iso", 2).uniqueIndex()
}
class Country(id: EntityID) : Entity(id) {
    var name: String by CountryTable.name
    var iso: String by CountryTable.iso
}
val poland = Country.find {
    CountryTable.iso.eq("pl")
}.first()
println(poland.name)
```

← **Opis tabeli w bazie danych.**

← **Utworzenie klasy odpowiadającej obiektowi w bazie danych.**

← **Bazę można odpytywać bezpośrednio za pomocą kodu Kotlin.**

Powyższe techniki opiszemy bardziej szczegółowo w podrozdziale 7.5 oraz w rozdziale 11.

1.3.2. Kotlin w Androidzie

Typowa aplikacja dla urządzenia przenośnego bardzo różni się od aplikacji dla komputera. Jest znacznie mniejsza i w ograniczonym stopniu wykorzystuje istniejący kod. Ponadto zazwyczaj aplikacje przenośne trzeba tworzyć szybko i muszą one działać niezawodnie na urządzeniach bardzo różnych rodzajów. Do tworzenia tego typu projektów Kotlin również nadaje się bardzo dobrze.

Struktura języka Kotlin oraz dostępność specjalnej wtyczki dla kompilatora powodują, że programowanie w systemie Android jest efektywniejsze i przyjemniejsze. Najczęstsze zadania, na przykład kodowanie obsługi kontrolki lub wiązanie elementów interfejsu użytkownika z właściwościami klas, realizuje się za pomocą znacznie krótszego kodu, a czasami bez kodu w ogóle (generuje go kompilator). Biblioteka Anko (<https://github.com/kotlin/anko>), również opracowana przez zespół autorów Kotlin, jeszcze bardziej ułatwia to zadanie, oferując adaptery przystosowane do wielu standardowych interfejsów API systemu Android.

Poniżej znajduje się prosty kod wykorzystujący bibliotekę Anko i pokazujący, co powinien zrobić programista, aby w języku Kotlin utworzyć aplikację dla systemu Android. Musi ten kod umieścić w aktywności i prosta aplikacja gotowa!

```
verticalLayout {
    val name = editText()
    button("Powiedz: cześć!") {
        onClick { toast("Cześć, ${name.text}!") }
    }
}
```

← **Utworzenie prostego pola tekstowego.**

← **Zwarta składnia wiążąca kod nasłuchujący, który wyświetla komunikat.**

← **Po kliknięciu tego przycisku wyświetlane jest imię wpisane w polu tekstowym.**

Inną ogromną zaletą języka Kotlin jest większa stabilność tworzonych w nim aplikacji. Jeżeli masz doświadczenie w pisaniu aplikacji dla systemu Android, na pewno dobrze znasz komunikat: „Niestety, aplikacja została zatrzymana”. Komunikat ten pojawia się po zgłoszeniu nieobsługiwanego wyjątku, zazwyczaj `NullPointerException`. System typów w języku Kotlin, które umożliwiają precyzyjne śledzenie wartości `null`, łagodzi problem zerowych wskaźników. W większości przypadków kod, który w języku Java powoduje zgłoszenie wyjątku `NullPointerException`, w języku Kotlin w ogóle się nie kompiluje, dzięki czemu przed udostępnieniem aplikacji użytkownikom można błąd poprawić.

Ponieważ Kotlin jest w pełni kompatybilny z Java 6, nie występują problemy z integracją. Można korzystać ze wszystkich nowych, ciekawych funkcjonalności Kotlin, a użytkownicy będą mogli uruchamiać aplikacje na swoich urządzeniach wyposażonych nawet w starsze wersje systemu Android.

Pod względem wydajności Kotlin również wypada niezgorzej. Wygenerowany przez kompilator kod działa równie wydajnie jak kod Java. Biblioteka uruchomieniowa Kotlin jest bardzo mała, więc skompilowane pakiety aplikacyjne nie są duże. Ponadto wiele standardowych bibliotek Kotlin zamienia funkcje lambda na kod procesora. Dzięki temu nie są tworzone nowe obiekty, a działanie aplikacji nie jest wstrzymywane na czas porządkowania pamięci.

Teraz, gdy wiesz, jaka jest przewaga Kotlin nad Javą, poznaj jego filozofię, odróżniającą go od innych nowoczesnych języków dla środowiska JVM.

1.4. Filozofia języka Kotlin

O Kotlinie zwykliśmy mówić, że jest to pragmatyczny, zwięzły, bezpieczny, kompatybilny język programowania. Co konkretnie znaczą te słowa?

1.4.1. Pragmatyzm

Pragmatyzm oznacza prostą cechę: Kotlin jest językiem przeznaczonym do rozwiązywania praktycznych problemów. Jego kształt wynika z wieloletniego doświadczenia autorów w tworzeniu dużych systemów, a oferowane funkcjonalności zostały opracowane pod kątem typowych przypadków, z którymi mierzą się programiści. Co więcej, zarówno zespół JetBrains, jak i społeczność użytkowników od wielu lat korzystają ze wstępnych wersji Kotlin i ich opinie zostały uwzględnione w oficjalnej wersji. Dlatego twierdzimy, że Kotlin umożliwia rozwiązywanie problemów w rzeczywistych projektach.

Kotlin nie jest językiem akademickim. Nie staramy się wzbogacać sztuki programowania i implementować innowacji informatycznych. Zamiast tego tam, gdzie jest to możliwe, wykorzystujemy funkcjonalności i rozwiązania spotykane w innych językach, które sprawdziły się w praktyce. Dzięki temu nasz język jest mniej skomplikowany, prostszy w opanowaniu i wykorzystujący sprawdzone koncepcje.

Co więcej, Kotlin nie narzuca żadnego stylu ani zasad programowania. Rozpoczynając naukę tego języka, możesz stosować style i techniki, które znasz z Javy. W miarę postępów będziesz stopniowo odkrywał przydatne cechy Kotlin i uczył się wykorzystywać je w swoim kodzie, dzięki czemu będzie on bardziej zwięzły i funkcjonalny.

Innym pragmatycznym aspektem Kotlina jest koncentracja na narzędziach. Dobre środowisko programistyczne jest równie ważne dla skutecznego programowania jak dobry język. Dlatego marginalne traktowanie środowiska IDE jest nie do przyjęcia. W przypadku języka Kotlin wtyczka dla środowiska IntelliJ IDEA była rozwijana równoległe z kompilatorem, a funkcjonalności języka są projektowane również pod kątem narzędzi.

Środowisko programistyczne odgrywa wielką rolę w poznawaniu funkcjonalności Kotlina. Często narzędzia automatycznie wykrywają najczęściej stosowane schematy kodowania, które można zastępować bardziej zwartymi konstrukcjami, i oferują stosowne zmiany. Analizując składnię sugerowaną przez zautomatyzowane narzędzia, możesz uczyć się wykorzystywać nowe funkcjonalności w swoim kodzie.

1.4.2. Zwięzłość

Powszechnie wiadomo, że poznanie istniejącego kodu zajmuje programiście więcej czasu niż napisanie nowego. Wyobraź sobie, że jesteś członkiem zespołu pracującego nad ogromnym projektem i musisz zakodować nową funkcjonalność lub poprawić błąd. Jaki jest Twój pierwszy krok? Musisz najpierw znaleźć sekcję kodu, którą masz zmienić. Dopiero wtedy możesz poprawić błąd. Aby określić, co powinieneś zrobić, musisz przeanalizować mnóstwo kodu, który mógł zostać napisany stosunkowo niedawno przez Twoich kolegów, ale równie dobrze dużo wcześniej przez Ciebie samego lub kogoś, kto już nie pracuje nad projektem. Dopiero po zrozumieniu sąsiedniego kodu jesteś w stanie wprowadzić niezbędne zmiany.

Im prostszy i krótszy jest kod, tym szybciej można się dowiedzieć, co się w nim dzieje. Oczywiście przemyślana struktura i opisowe nazwy mają tu istotne znaczenie. Ale ważny jest też sam język i jego *zwięzłość*. Język jest zwięzły, jeżeli jego składnia jasno oddaje przeznaczenie kodu i nie zaciemniają go dodatkowe instrukcje niezbędne do osiągnięcia zamierzonego celu.

Dokładamy wszelkich starań, aby cały kod Kotlin niósł treść, a nie tylko spełniał wymogi struktur programistycznych. Niektóre standardowe konstrukcje z języka Java, takie jak gettery, settery lub parametry konstruktorów powiązane z właściwościami, są w języku Kotlin stosowane niejawnie i nie zaciemniają kodu.

Niekiedy przyczyną niepotrzebnego wydłużenia kodu jest konieczność jawnego implementowania często wykonywanych zadań, na przykład wyszukiwania elementu w kolekcji. Podobnie jak wiele innych nowoczesnych języków, Kotlin również zawiera bogatą standardową bibliotekę umożliwiającą zastępowanie długich, powtarzalnych sekcji kodu gotowymi metodami z biblioteki. Dzięki funkcjom lambda można w funkcjach biblioteki łatwo umieszczać małe bloki kodu. W ten sposób wszystkie często wykonywane zadania można zlecać bibliotece i pisać tylko unikatowe, charakterystyczne dla danego zadania sekcje kodu.

Jednakże przeznaczeniem języka Kotlin nie jest ściśnięcie kodu do minimalnej liczby znaków. Na przykład mimo tego, że możliwe jest przeciążanie operatorów, nie można definiować własnych. Dlatego twórcy bibliotek nie mogą zastępować nazw metod tajemniczymi symbolami. Słowa są zazwyczaj bardziej zrozumiałe niż symbole i łatwiej jest znaleźć ich opis w dokumentacji.

Napisanie zwięzłego kodu zajmuje mniej czasu i, co ważne, mniej czasu potrzeba na jego przeczytanie. Dzięki temu programista może pracować wydajniej i kodować szybciej.

1.4.3. Bezpieczeństwo

Mówiąc, że język programowania jest *bezpieczny*, mamy zazwyczaj na myśli jego strukturę chroniącą programistę przed popełnianiem określonego rodzaju błędów. Oczywiście nie jest to bezwzględnym wyznacznikiem jakości języka. Żaden język nie chroni całkowicie przed błędami, a ponadto zazwyczaj taka ochrona kosztuje, ponieważ kompilatorowi trzeba dostarczyć dodatkowych informacji o operacjach, które ma wykonywać program. Kompilator porównuje te informacje z wykonywanymi operacjami. Z tego powodu zawsze istnieje pewien kompromis pomiędzy zapewnianym poziomem bezpieczeństwa a nakładem pracy, jakiego wymaga umieszczenie w kodzie szczegółowych adnotacji.

Naszym zamiarem było zapewnienie w języku Kotlin większego bezpieczeństwa niż w Javie, ale mniejszym kosztem. Sama maszyna JVM daje gwarancję bezpieczeństwa, na przykład zapewnia ochronę pamięci, zabezpiecza przed przepełnieniem buforów, jak również chroni przed różnymi problemami wynikającymi z niewłaściwego korzystania z dynamicznie przydzielanej pamięci. Kotlin jako język ze statycznym typowaniem przeznaczonym dla maszyny JVM zapewnia dodatkowo bezpieczeństwo typów danych. Odbywa się to jednak mniejszym kosztem niż w Javie, ponieważ kompilator automatycznie domniemywa typy i nie trzeba ich wszędzie specyfikować.

Kotlin oferuje więcej funkcjonalności umożliwiających wykrywanie błędów jeszcze na etapie kompilacji kodu, a nie dopiero po jego uruchomieniu. Przede wszystkim minimalizuje ryzyko zgłoszenia wyjątku `NullPointerException`. System typów umożliwia śledzenie zmiennych, które mogą przyjmować wartość `null`, oraz tych, które nie mogą jej przyjmować, i uniemożliwia wykonywanie na nich operacji prowadzących do zgłoszenia powyższego wyjątku. Związany z tym dodatkowy koszt jest minimalny, ponieważ w celu oznaczenia zerowalnej zmiennej wystarczy wpisać znak zapytania, jak niżej:

```
val s: String? = null ← Zmienna może przyjmować wartość null.
val s2: String = "" ← Zmienna nie może przyjmować wartości null.
```

Ponadto Kotlin oferuje wiele wygodnych sposobów przetwarzania zerowalnych danych, co znacznie zmniejsza ryzyko awarii aplikacji.

Innym wyjątkiem, przed którym chroni Kotlin, jest `ClassCastException`. Wyjątek ten jest zgłaszany podczas rzutowania typów bez uprzedniego sprawdzenia, czy jest to poprawna operacja. W Javie często programiści rezygnują ze sprawdzania typu, ponieważ jego nazwy trzeba użyć dwukrotnie: najpierw w instrukcji sprawdzającej, a następnie w instrukcji rzutującej typ. Natomiast w Kotlinie sprawdzanie i rzutowanie typu realizuje się za pomocą jednej instrukcji. Po sprawdzeniu typu można odwoływać się do elementów członkowskich bez dodatkowego rzutowania. Zatem nie ma powodu, aby rezygnować ze sprawdzania typu, nie ma również możliwości popełnienia błędu. Poniżej przedstawiony jest przykład:

```
if (value is String) ← Sprawdzenie typu.
    println(value.toUpperCase()) ← Użycie metody wybranego typu.
```

1.4.4. Kompatybilność

Twoje pierwsze pytanie dotyczące *kompatybilności* zapewne brzmi: „Czy mogę używać istniejących bibliotek?”. W języku Kotlin odpowiedź brzmi: „Tak, oczywiście”. Niezależnie od interfejsu API, który udostępnia biblioteka, można z niej korzystać. Można wywoływać metody Java, rozszerzać klasy, implementować interfejsy, stosować adnotacje itp.

Kotlin, w odróżnieniu od innych języków dla maszyny JVM, oferuje jeszcze większą kompatybilność umożliwiającą łatwe wywoływanie jego kodu w kodzie Java. Nie są do tego potrzebne żadne sztuczki: klasy i metody napisane w języku Kotlin wywołuje się tak samo jak klasy i metody napisane w Javie. Daje to ogromną elastyczność w mieszaniu kodów Kotlin i Java. Jeżeli trzeba użyć języka Kotlin w projekcie napisanym w języku Java, można za pomocą konwertera Java-Kotlin przekształcić dowolną wybraną klasę, a pozostałą część kodu skompilować i uruchomić bez dodatkowych modyfikacji. Konwersja dotyczy dowolnego typu klasy.

Innym obszarem kompatybilności, na którym skupia się Kotlin, jest jak najpełniejsze wykorzystanie istniejących bibliotek Java. Kotlin nie ma własnej kolekcji bibliotek, ponieważ w całości opiera się na standardowych bibliotekach Java i rozszerza je o dodatkowe metody, bardziej wygodne w użyciu (dokładniej ten mechanizm opisany jest w podrozdziale 3.3). Oznacza to, że nie trzeba opakowywać ani przekształcać klas z języka Java na Kotlin lub odwrotnie. Z bogactwa interfejsu API języka Kotlin można korzystać bez żadnych dodatkowych kosztów.

Narzędzia języka Kotlin również można w pełni wykorzystywać w projektach obejmujących dwa języki. Można kompilować dowolne kombinacje plików źródłowych Java i Kotlin, niezależnie od ich wzajemnych zależności. Środowisko programistyczne również obsługuje oba języki i umożliwi m.in.:

- swobodne przemieszczanie się pomiędzy plikami źródłowymi Java i Kotlin,
- diagnozowanie mieszanych projektów i przełączanie się pomiędzy kodami napisanymi w obu językach,
- refaktoryzowanie metod Java oraz odpowiednią modyfikację kodu Kotlin i odwrotnie.

Mamy nadzieję, że przekonałeś Cię, abyś wypróbował język Kotlin. Jak więc zacząć go używać? W następnej części rozdziału opisujemy proces kompilowania i uruchamiania kodu Kotlin za pomocą wiersza poleceń oraz innych narzędzi.

1.5. Narzędzia języka Kotlin

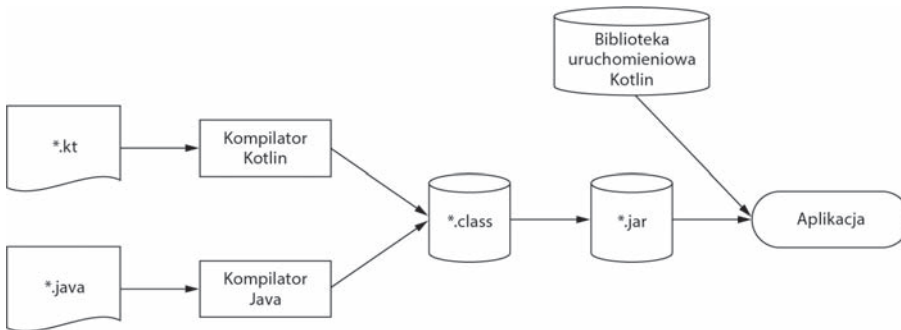
Kotlin, tak jak Java, jest językiem kompilowanym. Oznacza to, że aby uruchomić kod, trzeba go najpierw skompilować. Przeanalizujemy teraz proces kompilacji i przyjrzymy się różnym narzędziom, które zajmują się tym procesem za Ciebie. Jeżeli potrzebujesz dodatkowych informacji o przygotowaniu środowiska, zapoznaj się z sekcją *Tutorials* (podręczniki) na stronie Kotlin (<https://kotlinlang.org/docs/tutorials>).

1.5.1. Kompilator kodu

Kod źródłowy Kotlin zazwyczaj zapisuje się w plikach z rozszerzeniem *.kt*. Kompilator analizuje kod źródłowy i generuje pliki *.class* tak samo jak kompilator Java. Następnie wygenerowane pliki są pakowane i uruchamiane w sposób właściwy rodzajowi aplikacji. W najprostszym przypadku kod kompiluje się za pomocą polecenia `kotlinc`, a uruchamia poleceniem `java`, jak niżej:

```
kotlinc <plik źródłowy lub katalog> -include-runtime -d <nazwa pliku JAR>
java -jar <nazwa pliku JAR>
```

Rysunek 1.1 przedstawia uproszczony proces budowania aplikacji w języku Kotlin.



Rysunek 1.1. Proces budowania aplikacji w języku Kotlin

Kod wygenerowany za pomocą kompilatora Kotlin jest uzależniony od **biblioteki uruchomieniowej** tego języka zawierającej własne klasy i rozszerzenia, o które uzupełniane są standardowe interfejsy API języka Java. Biblioteka uruchomieniowa musi być dostarczana razem z aplikacją.

W większości przypadków będziesz kompilował kod za pomocą systemów Maven, Gradle i Ant. Kotlin jest kompatybilny ze wszystkimi wymienionymi systemami. Szczegółowe informacje na ten temat znajdziesz w dodatku A. Systemy te obsługują również projekty z mieszanym kodem. Ponadto Maven i Gradle dołączają bibliotekę uruchomieniową Kotlin jako zależność wymaganą przez aplikację.

1.5.2. Wtyczki dla IntelliJ IDAE i Android Studio

Wtyczka Kotlin dla środowiska IntelliJ IDEA była rozwijana równoległe z samym językiem. IntelliJ IDEA jest najlepiej wyposażonym środowiskiem programistycznym dla języka Kotlin. Jest ono dojrzałe, stabilne i oferuje pełny zestaw narzędzi programistycznych.

Wtyczka jest standardowo dostępna w wersjach IntelliJ IDEA 15 i nowszych, nie trzeba więc wykonywać dodatkowych czynności, aby z niej korzystać. Można jej używać w wersjach IntelliJ IDEA Community Edition i IntelliJ IDEA Ultimate. Wystarczy w oknie *New Project* (nowy projekt) wybrać opcję *Kotlin* i gotowe.

Jeżeli korzystasz z Android Studio, musisz zainstalować wtyczkę za pomocą menedżera wtyczek. W tym celu w oknie *Settings* (ustawienia) wybierz sekcję *Plugins* (wtyczki), kliknij przycisk *Install JetBrains plugin* (zainstaluj wtyczkę JetBrains) i wybierz z listy pozycję *Kotlin*.

1.5.3. Interaktywna powłoka

Aby szybko sprawdzić krótki kod, możesz użyć interaktywnej powłoki (tzw. REPL — ang. *read-eval-print loop*, pętla „wczytaj, wykonaj, wyświetl”). W tej powłoce można wpisywać kod Kotlin wiersz po wierszu i od razu sprawdzać wyniki jego wykonania. Powłokę uruchamia się, wpisując polecenie `kotlinc` bez argumentów lub klikając odpowiednią opcję menu w środowisku IntelliJ IDEA.

1.5.4. Wtyczka dla Eclipse

Jeżeli używasz środowiska Eclipse, również możesz korzystać z wtyczki dla języka Kotlin. Wtyczka ta oferuje najważniejsze funkcjonalności, takie jak poruszanie się po projekcie i uzupełnianie kodu. Jest dostępna w serwisie Eclipse Marketplace. Aby ją zainstalować, wybierz polecenie menu *Help/Eclipse Marketplace*, a następnie wyszukaj na liście pozycję *Kotlin*.

1.5.5. Internetowy „plac zabaw”

Najprostszy sposób wypróbowania języka Kotlin nie wymaga instalacji ani konfiguracji środowiska. Na stronie <http://try.kotl.in> dostępny jest „plac zabaw”, na którym można pisać, kompilować i uruchamiać proste programy. Strona zawiera również przykładowe kody prezentujące możliwości tego języka, między innymi wszystkie przykłady opisane w tej książce. Dostępne są też serie interaktywnych ćwiczeń do nauki języka.

1.5.6. Konwerter Java-Kotlin

Opanowanie nowego języka programowania nigdy nie przychodzi bez wysiłku. Dlatego przygotowaliśmy ciekawe, małe narzędzie, dzięki któremu, wykorzystując swoje doświadczenie z językiem Java, szybciej nauczysz się Kotlin. Tym narzędziem jest zautomatyzowany konwerter Java-Kotlin.

Na początku nauki, gdy nie będziesz jeszcze znał dokładnie składni języka, konwerter pomoże Ci formułować wyrażenia. Będziesz mógł pisać kod w Javie, zapisywać go w pliku i automatycznie konwertować na język Kotlin. Uzyskany kod nie zawsze będzie idealnie ścisły, ale będzie działał poprawnie i pozwoli Ci robić postępy w nauce.

Konwerter doskonale nadaje się również do wprowadzenia języka Kotlin do istniejącego projektu. Jeżeli będziesz chciał utworzyć nową klasę, najlepiej zrób to od razu w Kotlinie. W przypadku konieczności wprowadzenia większych zmian w istniejącej klasie również możesz użyć Kotlin. W tym miejscu pojawia się konwerter. Najpierw przekształć klasę na język Kotlin, a następnie wprowadź w niej zmiany, wykorzystując wszystkie zalety tego języka.

Używanie konwertera w środowisku IntelliJ IDEA jest wyjątkowo proste. Wystarczy skopiować kod Java i wkleić go do pliku Kotlin albo użyć polecenia menu *Convert Java File to Kotlin File* (przekształć plik Java w Kotlin), jeżeli trzeba przekształcić cały plik. Konwerter jest również dostępny w środowisku Eclipse i w internecie.

1.6. Podsumowanie

- Język Kotlin wykorzystuje statyczne typy danych i domniemanie typów, dzięki czemu kod jest poprawny, wydajny i zwięzły.
- Jest to zarówno język programowania obiektowego, jak i funkcyjnego o wysokim poziomie abstrakcji dzięki funkcjom pierwszej klasy. Obsługa niezmiennych wartości ułatwia tworzenie i testowanie kodu wielowątkowego.
- Język ten dobrze nadaje się do tworzenia aplikacji serwerowych, w pełni obsługuje wszystkie istniejące platformy Java i oferuje nowe narzędzia do wykonywania najczęstszych zadań, takich jak generowanie kodu HTML i obsługa baz danych.
- Można go używać do tworzenia aplikacji dla systemu Android, ponieważ jego biblioteka uruchomieniowa jest mała, kompilator obsługuje interfejs API systemu Android, a biblioteka funkcji realizujących najczęstsze operacje w tym systemie jest bardzo bogata.
- Kotlin jest bezpłatnym, otwartym oprogramowaniem, w pełni obsługiwanym przez najważniejsze środowiska programistyczne i systemy kompilujące.
- Jest to język pragmatyczny, bezpieczny, zwięzły i kompatybilny. Wykorzystuje sprawdzone sposoby realizacji typowych zadań i zapobiega powstawaniu najczęstszych problemów, na przykład zgłaszaniu wyjątku `NullPointerException`. Kod jest zwarty i czytelny, w pełni integrujący się z kodem Java.

Skorowidz

A

- abstrakcje, 313
- adnotacja
 - @Retention, 278
 - o zerowalności, 169
- adnotacje, 271, 272, 332
- adres, 273
- argumenty, 278
- deklarowanie, 277
- dostosowywanie serializacji, 286
- klasy generyczne, 279
- proces przetwarzania, 277
- serializacja JSON, 275
- stosowanie, 272
- adres adnotacji, 273
- analiza
 - danych JSON, 289
 - odwiedzin stron, 228
- Android, 31, 325, 332
- Android Studio, 36
- anonimowe klasy wewnętrzne, 121
- Ant, 333
- aplikacje
 - stacjonarne, 341
 - WWW, 340
- argumenty
 - domyślne wartości, 68
 - nazwane, 68
 - typowane, 167, 223, 245–248
 - wartość domyślna, 223
 - wartość null, 223
- ASCII, 83
- asercja
 - niezerowa, 161
 - should, 319
- automatyczne generowanie
 - uniwersalnych metod, 111

B

- badanie obiektów, 280
- bazy danych, 341
- bezpieczeństwo, 34
- bezpieczne rzutowanie typów, 160
- biblioteka
 - Anko, 31, 325, 326
 - Bootstrap, 313
 - Exposed, 323, 324
 - JKid, 275, 340
 - kotlintest, 320, 321
 - Retrofit, 340
 - uruchomieniowa, 36
- bloki inicjatora, 100

C

- ciało
 - blokowe, 41
 - wyrażeń, 41
- ciągi znaków, 80
- dzielenie, 81
- wielowierszowe, 83
- cudzysłowy potrójne, 81, 82

D

- daty, 321
- definiowanie języka DSL, 299
- deklaracja
 - adnotacji, 272, 277
 - destrukturyzująca, 77, 206
 - funkcji, 69, 70
 - funkcji wysokopoziomowej, 221
 - interfejsu, 96
 - klas, 100
 - klas generycznych, 244

deklaracja
 mutowalnej właściwości, 46
 obiektu, 114
 obiektu towarzyszącego, 118
 typu funkcyjnego, 223
 właściwości rozszerzającej, 77
 dekorowanie klas, 112
 delegat, 214
 delegowanie
 klas, 108, 112
 operacji, 208
 właściwości, 208–216
 deserializacja, 293
 obiektów, 289
 Dokka, 336
 dokumentowanie kodu, 335
 komentarze, 335
 narzędzie Dokka, 336
 domniemanie typów, 27, 41
 dostęp
 do elementu, 202
 do mapy walidatorów, 269
 do pól, 105
 dowolne obiekty, 51
 DRY, Don't Repeat Yourself, 84
 DSL, domain-specific language, 31, 299
 dziedziczenie klas, 171

E

elementy map, 59
 etykieta, 237

F

filtrowanie
 elementów kolekcji, 233
 właściwości, 287
 filtry, 133
 formatowanie ciągów znaków, 43
 funkcja, 40
 all(), 135
 any(), 135
 apply(), 147, 149
 buildString(), 150
 count(), 136
 filter(), 222, 239
 find(), 136

 flatMap(), 137
 groupBy(), 136
 item(), 314
 lazy, 209
 let(), 163
 should(), 320
 startActivity(), 253
 use(), 235
 with(), 147, 148
 funkcje
 anonimowe, 239
 bloki kodu, 124
 deserializujące, 292
 generujące kod HTML, 311
 generujące tag, 311
 generyczne, 243
 kopiujące, 264
 lokalne, 84
 nadrzędne, 85
 najwyższego poziomu, 70, 315
 nienadpiszalność, 75
 o dowolnej liczbie argumentów, 78
 pomocnicze, 74
 rozszerzające, 72, 86, 149, 166, 322
 serializujące obiekt, 285
 serializujące właściwość, 289
 śródwierszowe, 229, 230
 ograniczenia, 232
 stosowanie, 234
 tworzące kolekcje, 183
 wysokopoziomowe, 219
 wywoływanie, 67
 z urzeczowionymi
 argumentami typowanymi, 250
 zawierające funkcje, 226

G

generator HTML, 309, 312, 314
 generowanie
 dokumentacji, 336
 kodu HTML, 304
 generyczne funkcje, 243
 gettery, 45, 105
 Gradle, 318, 331

H

- hierarchia
 - interfejsów kolekcji, 183
 - klas, 53, 90

I

- identyfikator this, 238
- implementacja
 - funkcji, 67
 - interfejsu, 90, 92, 97
 - właściwości interfejsu, 104
 - wrażenia lambda, 145
- importowanie klas, 73
- inicjalizacja z opóźnieniem, 209
- inicjowanie
 - klas, 100
 - klas nadrzędnych, 102
 - mapy, 59
 - właściwości, 164
- instrukcja
 - catch, 63
 - return, 236, 237, 239
 - return z etykietą, 237
 - switch, 50, 51
 - synchronized, 230
 - throw, 159
 - try, 62, 235
- inteligentne rzutowanie, 53, 54
- IntelliJ IDEA, 36, 41
- interaktywna powłoka, 37
- interfejs
 - API, 78, 300
 - API refleksji, 281
 - Compare, 115
 - DataParser, 186
 - FileContentProcessor, 185
 - KCallable, 281
 - KClass, 281
 - KFunction, 281
 - KProperty, 281
 - SAM, 143
- interfejsy, 90
 - funkcyjne, 133
 - funkcyjne Java, 143
 - kolekcji, 183
 - mutowalne, 181
 - tylko do odczytu, 181

- użytkownika, 325
 - z serializowanym stanem, 96
- inwariantne klasy kolekcji, 258
- iteracje
 - dat, 205
 - elementów map, 59
 - liczb, 57

J

- jawna
 - implementacja powiadamiania, 212
 - konwersja, 146
- język DSL, 31, 299
 - generowanie kodu HTML, 304
 - konwencja invoke, 318
 - obsługa zapytań SQL, 322
 - przetwarzanie dat, 322
 - struktura, 303
 - wewnętrzny, 302
- języki
 - domenowe, 301
 - ogólnego przeznaczenia, 301
- JSON, JavaScript Object Notation, 275, 340

K

- katalogi, 48
- klasa, 44
 - Button, 92
 - ObservableProperty, 213
 - PropertyChangeSupport, 211
 - String, 81
- klasy
 - abstrakcyjne, 94
 - danych, 29, 108, 111
 - deserializujące obiekt, 292
 - generyczne, 244, 279
 - inwariantne, 258, 262
 - jako argumenty adnotacji, 278
 - kolekcji, 67
 - kontrawariantne, 262
 - kowariantne, 257, 262
 - przechowujące informacje, 295
 - wewnętrzne, 96
 - wyliczeniowe, 49, 50
 - zagnieżdżone, 96
 - zapieczętowane, 98

klienci HTTP, 340
 kolekcje, 66, 125, 133, 178
 jako typy platformowe, 184
 leniwe operacje, 138
 mutowalne, 181
 obronne kopiowanie, 181
 tylko do odczytu, 153, 181
 wartości zerowalnych, 179
 zagnieżdżone, 137
 zerowalnych wartości, 180
 komentarze dokumentacyjne, 335
 kompatybilność, 35
 kompilator, 36
 kompilowanie projektów
 adnotacje, 332
 narzędzie
 Ant, 333
 Gradle, 318, 331, 332
 Maven, 333
 konflikt nazw metod, 149
 konstruktor
 dodatkowy, 117
 główny, 100
 SAM, 146
 kontrawariancja, 261
 konwencja, 193
 invoke, 316, 317, 318
 iterator, 205
 operatora in, 203
 konwencje
 w kolekcjach, 202
 w zakresach, 202
 zagnieżdżanie bloków kodu, 316
 konwersja
 listy na mapę, 136
 wartości, 176
 konwerter Java-Kotlin, 37
 kowariancja, 257
 kowariantne klasy kolekcji, 258

L

lambda, *Patrz* wyrażenia lambda
 leniwe operacje, 138
 lista rozwijana, 314
 literały typów prostych, 175

Ł

łączenie
 tabel, 324
 wywołań infix, 319, 321

M

mapy, 59, 133, 215
 Maven, 333
 metaadnotacje, 277
 metoda
 callBy(), 293
 compareTo(), 200
 contains(), 204
 copy(), 112
 Delegates.observable(), 214
 equals(), 109, 160
 get(), 202
 hashCode(), 110
 invoke(), 316
 rangeTo(), 204
 set(), 202, 203
 split(), 81
 toString(), 108
 metody
 dostępowe, 45, 106
 getterzy, 45, 105
 setterzy, 45, 105
 własne, 47
 komponentowe, 206
 nasłuchujące, 124
 operatorów, 195
 uniwersalnych obiektów, 108
 wytwórcze, 116, 118
 modyfikator
 abstract, 92, 94
 final, 92, 94
 inline, 230
 internal, 95
 open, 92, 94
 override, 90, 94
 private, 95
 protecte, 95
 public, 45, 95

N

nadpisanie metody, 75
 narzędzia, 341
 języka, 35
 narzędzie
 Ant, 333
 Dokka, 336
 Gradle, 318, 331, 332
 Maven, 333
 nawiasy klamrowe, 41
 nazwa argumentu, 68, 129, 221
 niemutowalność zmiennych, 111
 nienadpiszalność funkcji, 75
 niestandardowy serializator, 289

O

obiekt anonimowy, 121
 obiekty
 expando, 215
 nasłuchujące, 147
 towarzyszące, 114, 116, 118
 implementacja interfejsu, 119
 rozszerzenia, 119
 wartości, 45
 obronne kopiowanie kolekcji, 181
 obsługa wartości null, 159
 odbiorniki, 147, 149
 odwołania
 do elementów, 123, 126, 131
 do zmiennych, 129
 do właściwości zerowalnej, 164
 ograniczenia
 argumentów typowanych, 245
 funkcji śródwierszowych, 232
 urzeczowionych argumentów
 typowanych, 253
 operacje
 na bazach danych, 341
 na kolekcjach, 233
 na sekwencjach, 139
 operator
 ..., 59
 bezpiecznego rzutowania, 160
 bezpiecznego wywołania, 158

dodawania, 195
 Elvisa, 158
 identyczności, 200
 in, 60, 203
 indeksu, 202
 is, 160
 porównania, 199
 przypisania, 196
 rozciągania, 79
 równości, 199
 wywołania, 157
 operatory
 arytmetyczne, 194
 bitowe, 197
 dwuargumentowe, 195
 jednoargumentowe, 198, 199
 złożone, 196
 opóźnienie, 209
 oprogramowanie, 29
 otwarty zakres, 204

P

pakiety, 48
 parser danych JSON, 290
 pętla, 207
 for ... in, 59
 loop, 205
 while, 57
 podtypy, 254, 255
 postępy, 57
 powiadamianie, 212
 o zmianie właściwości, 212, 214
 powtarzalny kod, 313
 pragmatyzm, 32
 problem kruchej klasy bazowej, 92
 programowanie
 funkcyjne, 28
 obiektywne, 28
 reaktywne, 341
 serwerów, 30
 projekcja
 typu, 265
 z gwiazdką, 266
 protokół REST, 340

przeciążanie
 operatorów, 193
 arytmetycznych, 194
 jednoargumentowych, 198
 nierówności, 200
 porównania, 199
 metody, 317
 przekształcanie liczb, 174
 przetwarzanie
 dat, 321
 kolekcji, 77

R

refaktoryzacja kodu, 55
 refleksja, 271, 280
 serializacja obiektów, 285
 tworzenie obiektu, 293
 reprezentacja obiektu, 108
 rozszerzający typ funkcyjny, 305
 rozszerzenia, 74, 84
 członkowskie, 322, 324
 typów
 prostych, 321
 zerowalnych, 166
 funkcyjnych, 317
 równość obiektów, 109
 rzutowanie typów, 39, 53, 160, 248
 generycznych, 249

S

SAM, Single Abstract Method, 143
 sekwencje, 139, 142
 serializacja
 JSON, 275, 340
 obiektów, 285, 287
 pojedynczej właściwości, 288
 wartości logicznej, 294
 settery, 45, 105
 silnik Multi-OS Engine, 27
 singleton, 114, 116
 składnia
 infix, 77
 kodu, 300
 wyrażenia lambda, 126

słowa kluczowe
 miękkie, 50
 słowo kluczowe
 by, 112
 class, 50
 enum, 50
 extends, 90, 245
 finally, 62
 implements, 90
 in, 60
 object, 114
 catch, 62
 static, 116
 try, 62, 63
 out, 257
 val, 42
 var, 42
 varargs, 77
 sprawdzanie
 typów, 248, 250
 zakresu, 60
 stosowanie adnotacji, 272
 struktura
 języka DSL, 303
 danych, 341
 system typów danych, 153
 szablon znakowy, 44

Ś

śródwierszowe wyrażenia lambda, 234

T

tablice, 178
 obiektów, 186
 typów prostych, 186, 188
 znaków, 187
 testowanie kodu, 339
 translacja delegowanych właściwości, 215
 tworzenie
 interfejsu użytkownika, 325
 kolekcji, 66
 sekwencji, 142

- typ danych, 153
 - Any, 176
 - Any?, 176
 - Boolean, 172
 - Boolean?, 173
 - Int, 172
 - Int?, 173
 - Nothing, 178
 - Unit, 177
 - typowanie danych
 - dynamiczne, 27
 - statyczne, 27
 - typy
 - funkcyjne, 28, 220–223, 317
 - rozszerzające, 305, 307
 - generyczne, 241, 248, 254
 - odbiornika, 307
 - opakowujące, 176
 - platformowe, 168, 170, 184
 - projekcja, 265
 - proste, 172
 - rzutowanie, 248
 - sprawdzanie, 248
 - surowe, 242
 - wariancja w miejscu deklaracji, 263
 - zerowalne, 28, 153, 166, 173, 178
 - znaczenie, 156
- U**
- urzeczowione argumenty typowane, 252
 - użycie klasy, 46
- W**
- walidator, 267, 268
 - wariancja typu
 - w miejscu deklaracji, 241, 263
 - w miejscu użycia, 242
 - wariancje, 254
 - wartość
 - domyślna, 223
 - null, 159, 169, 223
 - właściwości, 45
 - inicjowane z opóźnieniem, 164
 - najwyższego poziomu, 70
 - rozszerzające, 72, 76
 - zerowalne, 164
 - współdzielenie metod dostępowych, 208
 - wstawianie
 - kodu funkcji, 230
 - operacji, 233
 - wstrzykiwanie zależności, 116, 340
 - wtyczki
 - Android Studio, 36
 - dla Eclipse, 37
 - dla IntelliJ IDAE, 36
 - wydajność wyrażeń lambda, 229
 - wygładzanie kodu, 84
 - wyjątek, 61
 - NullPointerException, 32, 156, 169
 - NumberFormatException, 62
 - wyjście
 - lokalne domyślne, 239
 - nielokalne, 237
 - z wyrażenia lambda, 237
 - wyrażenia, 41
 - lambda, 28, 123, 147, 227
 - instrukcja return, 236
 - jawna konwersja, 146
 - nielokalne wyjście, 237
 - składnia, 126
 - usunięcie typu argumentu, 128
 - w argumentach, 144
 - wydajność, 229
 - z odbiornikami, 147, 305, 309
 - zarządzanie zasobami, 234
 - obiektowe, 114, 121
 - regularne, 80, 81
 - wyrażenie
 - if, 55
 - if zwracające wartość, 55
 - try, 63
 - when, 49–56
 - when bez argumentów, 52
 - wywołanie infix, 79
 - wywoływanie funkcji, 67
 - rozszerzających, 74
- Z**
- zagnieżdżanie bloków kodu, 316
 - zakresy, 57
 - zależności między podtypami, 257
 - zapisywanie w mapie, 216
 - zapytania SQL, 322

- zastosowania języka, 30
- zdarzenia, 124
- zerowalne typy
 - danych, 153, 168, 178
 - proste, 173
- zerowalność argumentów typowanych, 167
- zmienne, 42
 - lokalne, 130
 - mutowalne, 42
 - niemutowalne, 42
- związłość kodu, 33

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 

Kotlin został zaprojektowany jako obiektowy język w pełni interoperacyjny z kodem napisanym w Javie. Został zaprezentowany w 2011 roku. Od tego czasu jest rozwijany, a jego popularność stale rośnie. Kotlin jest językiem o statycznie typowanych zmiennych. Charakteryzuje się czytelną składnią oraz ścisłą integracją z istniejącymi projektami, bibliotekami i platformami utworzonymi w Javie. Kompiluje się do postaci binarnego kodu JVM, więc można go uruchamiać wszędzie tam, gdzie jest Java, również w Androidzie. Kotlin zainteresuje programistów aplikacji mobilnych także dlatego, że kod napisany w tym języku obciąża system w minimalnym stopniu. Przy tym wszystkim środowisko Kotlinia jest w całości otwartym oprogramowaniem do bezpłatnego korzystania w dowolnych celach!

Ta książka jest przeznaczona dla osób, które mają pewne doświadczenia z Javą i chcą szybko poznać Kotlinia w stopniu pozwalającym na tworzenie aplikacji serwerowych, dla systemu Android i maszyn JVM. Znalazł się tu opis podstawowych cech języka i jego najważniejszych struktur, a następnie przedstawiono bardziej zaawansowane zagadnienia, takie jak tworzenie wysokopoziomowych abstrakcji i języków domenowych. Duży nacisk położono na integrację kodu Kotlin z istniejącymi projektami Java oraz na sposoby wprowadzania Kotlinia do aktualnie użytkowanego środowiska. Zaprezentowano kilka przydatnych bibliotek i narzędzi, znakomicie ułatwiających pracę programiście. Nie zabrakło również licznych przykładów kodu, ilustrujących omawiane zagadnienia.

W tej książce między innymi:

- solidne wprowadzenie do Kotlinia
- klasy, klasy danych i funkcje lambda
- typy danych, w tym kolekcje i puste zmienne
- własne interfejsy i abstrakcje w Kotlinie
- parametry reifikowanego typu, adnotacje i refleksje

Dmitry Jemerov pracuje w JetBrains (firma programistyczna z siedzibą w Pradze) od 2003 roku. Jest jednym ze współautorów Kotlinia, dla którego stworzył pierwszą wersję generatora kodu na maszynie JVM. Prezentował ten język na konferencjach na całym świecie. Obecnie kieruje zespołem rozwijającym wtyczkę Kotlin dla środowiska IntelliJ IDEA.

Svetlana Isakova dołączyła do zespołu Kotlinia w 2011 roku. Pracowała nad domniemaniem typów i rozpoznawaniem przeciążania w kompilatorze. Obecnie zajmuje się popularyzacją języka Kotlin, prezentując go na konferencjach i szkoleniach.

Kotlin – idealny dla Androida!

 Helion	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej!</i>	
 helion.pl	 SZKOLENIA	ISBN 978-83-283-4720-5	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	 9 788328 347205	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 67,00 zł	