

Rusz głową!

Kotlin

Pobuszuj
w standardowej
bibliotece Kotlina



Odkryj
tajniki typów
sparametry-
zowanych



Dowiedz się,
jak Elvis może
zmienić Twoje życie

Poradnik programowania
w Kotlinie dla tych, którzy
chcą się uczyć



Uniknij
żenujących
błędów
z wyrażeniami
lambda



Pisz
nieziemskie funkcje
wyższego rzędu



Zbadaj kolekcje
pod mikroskopem

Dawn Griffiths, David Griffiths

Tytuł oryginału: Head First Kotlin: A Brain-Friendly Guide

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-5869-0

© 2020 Helion SA

Authorized Polish translation of the English edition of *Head First Kotlin*
ISBN 9781491996690 © 2019 Dawn Griffiths and David Griffiths.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

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 jakiejkolwiek 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/kotrug>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści (podsumowanie)

	Wprowadzenie	xxi
1	Zaczynamy. Szybki skok	1
2	Typy proste i zmienne. Być zmienną	31
3	Funkcje. Wychodzimy poza main	59
4	Klasy i obiekty. Trochę klasy	91
5	Klasy pochodne i bazowe. Stosowanie dziedziczenia	121
6	Klasy abstrakcyjne i interfejsy. Poważny polimorfizm	155
7	Klasy danych. Używanie danych	191
8	Wartość null i wyjątki. Cały i zdrów	219
9	Kolekcje. Zorganizuj się	251
10	Typy sparametryzowane. Odróżniaj wariację od kontrawariancji	289
11	Lambdy i funkcje wyższego rzędu. Kod używany jak dane	325
12	Wbudowane funkcje wyższego rzędu. Wzmocnij swój kod	363
A	Koprocedury. Współbieżne wykonywanie kodu	397
B	Testowanie. Pociągnij swój kod do odpowiedzialności	409
C	Pozostałości. Dziesięć najważniejszych rzeczy (których nie opisaliśmy)	415

Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg myśli o Kotlinie. Choć Ty starasz się czegoś *nauczyć*, Twój mózg robi Ci przysługę, dbając o to, by te informacje się *nie utrwały*. Twój mózg myśli sobie: „Lepiej zostawię miejsce na coś naprawdę ważnego, na przykład: jakich dzikich zwierząt lepiej unikać albo dlaczego jeżdżenie na snowboardzie nago nie jest najlepszym pomysłem”. A zatem jak możesz *oszukać* swój mózg, by myślał, że Twoje życie zależy od nauczania się programowania w Kotlinie?

Dla kogo jest przeznaczona ta książka?	xxii
Wiemy, co sobie myślisz	xxiii
Wiemy, co sobie myśli Twój mózg	xxiii
Metapoznanie — myślenie o myśleniu	xxv
Oto co zrobiliśmy	xxvi
Przeczytaj to	xxviii
Zespół recenzentów technicznych	xxx
Podziękowania	xxxi

Zaczynamy

Szybki skok

1

Kotlin robi dużo szumu.

Od momentu pojawienia się pierwszej wersji Kotlina język ten urzekł programistów swoją *przyjazną składnią, spójnością, elastycznością* oraz *możliwościami*. W tej książce nauczymy Cię **pisać w Kotlinie własne aplikacje**, a zaczniemy od pokazania, jak stworzyć i uruchomić prosty program. Po drodze przedstawimy podstawy składni Kotlina, takie jak *instrukcje, pętle* oraz *konstrukcje warunkowe*. Twoja podróż właśnie się zaczyna...

Możliwość wyboru platformy docelowej, na którą zostanie skompilowany kod napisany w Kotlinie, oznacza, że można go uruchamiać na serwerach, w chmurze, w przeglądarce, na urządzeniach mobilnych i nie tylko.



Witamy w Kotlinicynie	2
Kotlina można używać niemal wszędzie	3
Co zrobimy w tym rozdziale	4
Instalowanie IntelliJ IDEA (Community Edition)	7
Stworzymy prostą aplikację	8
Właśnie utworzyłeś swój pierwszy projekt w Kotlinie	11
Dodaj do projektu nowy plik Kotlina	12
Anatomia funkcji main	13
Dodaj funkcję main do pliku App.kt	14
Jazda próbna	15
Co możemy nakazać w funkcji main?	16
W kółko i w kółko, i w kółko...	17
Przykład pętli	18
Rozgałęzienia warunkowe	19
Używanie if do zwracania wartości	20
Aktualizujemy funkcję main	21
Stosowanie interaktywnej powłoki Kotlina	23
W REPL można wpisywać fragmenty mające wiele wierszy kodu	24
Wymieszane komunikaty	27
Twój przybornik do Kotlinia	30

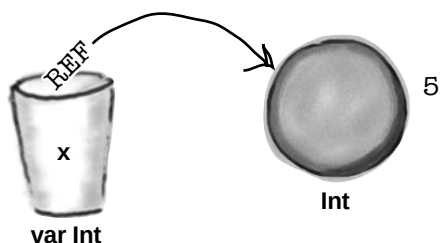
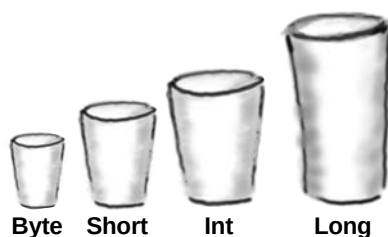
Typy proste i zmienne

Być zmienną

2

Jest takie coś, od czego zależy każdy kod — zmienne.

W tym rozdziale zajrzymy za kulisy i pokażemy Ci, *jak naprawdę działają zmienne w Kotlinie*. Poznasz w nim **proste typy danych**, takie jak *Int*, *Float* oraz *Boolean*, i dowiesz się, jak kompilator Kotliny może **inteligentnie wywnioskować typ zmiennej na podstawie wartości, którą jej przypiszemy**. Dowiesz się także, jak można używać **szablonów łańcuchowych** do tworzenia złożonych łańcuchów przy wykorzystaniu krótkiego kodu oraz jak tworzyć **tablice**, w których będzie można przechowywać wiele wartości. I w końcu dowiesz się także, jak *wielkie znaczenie dla życia w Kotlinizynie mają obiekty*.



Twój kod potrzebuje zmiennych	32
Co się dzieje, kiedy zadeklarujesz zmienną?	33
Zmienna zawiera referencję do obiektu	34
Typy proste Kotliny	35
Jak jawnie zadeklarować typ zmiennej?	37
Używaj wartości dostosowanej do typu zmiennej	38
Przypisywanie wartości innej zmiennej	39
Trzeba skonwertować wartość	40
Co się dzieje podczas konwertowania wartości?	41
Uważaj na przepełnienie	42
Zapisywanie wielu wartości w tablicy	45
Tworzymy aplikację HasłoMator	46
Dodaj kod do pliku HasłoMator.kt	47
Kompilator domyśla się typu tablicy na podstawie jej wartości	49
var oznacza, że zmienna może wskazywać na inną tablicę	50
val oznacza, że zmienna zawsze będzie wskazywać tę samą tablicę...	51
Zamotane referencje	54
Twój przybornik do Kotliny	58

Funkcje

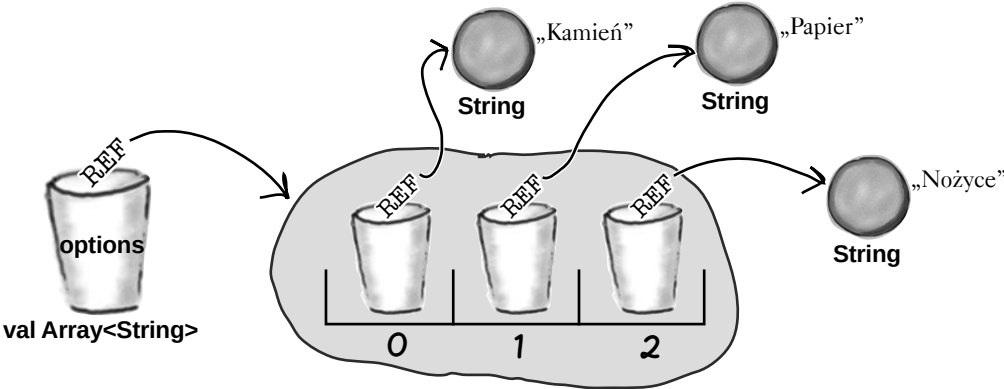
3 Wychodzimy poza main

Czas zrobić kolejny krok i poznać funkcje.

Jak na razie cały kod, który pisałeś, był umieszczany wewnątrz funkcji *main*. Jednak chcąc pisać kod, który będzie **lepiej zorganizowany i łatwiejszy do utrzymania**, musisz nauczyć się, ***jak dzielić kod na części i umieszczać go w odrębnych funkcjach***. W tym rozdziale nauczysz się, *jak pisać funkcje* oraz *jak prowadzić interakcje z aplikacją*, a wszystko to w ramach tworzenia gry! Dowiesz się także, jak pisać zwarte ***funkcje jednowyrażeniowe***. W tym rozdziale zobaczysz także, jak ***przeglądać zakresy i kolekcje***, używając potężnej pętli *for*.



Napiszmy grę: Kamień, nożyce, papier	60
Ogólny projekt gry	61
Wybieranie opcji przez grę	63
Jak się tworzy funkcje?	64
Do funkcji można przekazywać więcej informacji	65
Pobieranie informacji z funkcji	66
Funkcje, których ciałem jest jedno wyrażenie	67
Dodajemy funkcję <code>getGameChoice</code> do pliku <code>Game.kt</code>	68
Funkcja <code>getUserChoice</code>	75
Jak działają pętle <code>for</code> ?	76
Zapytaj gracza, jaką opcję wybiera	78
Pomieszane komunikaty	79
Musimy sprawdzić dane wpisane przez gracza	81
Dodajemy <code>getUserChoice</code> do pliku <code>Game.kt</code>	83
Dodajemy funkcję <code>printResult</code> do pliku <code>Game.kt</code>	87
Twój przybornik do Kotlina	89



4

Klasy i obiekty
Trochę klasy

Nadszedł czas, byśmy przestali ograniczać się do prostych typów Kotlina.

Wcześniej czy później nadejdzie czas, kiedy zechcesz użyć czegoś *więcej* niż jedynie typów prostych Kotlina. I właśnie w tym momencie pojawiają się **klasy**. Klasy są *wzorcami* pozwalającymi na **tworzenie własnych typów obiektów** i definiowanie ich właściwości oraz funkcji. W tym rozdziale dowiesz się, **jak projektować i definiować klasy** oraz jak używać ich do **tworzenia nowych typów obiektów**. Przy okazji spotkasz **konstruktory**, **bloki inicjalizatora**, **akcesory *get i set*** oraz dowiesz się, jak można ich używać do zabezpieczania właściwości. W końcu dowiesz się także, że **ukrywanie danych jest wbudowane w cały kod pisany w Kotlinie**, co oszczędza nam czasu, wysiłku i mnóstwa pisania na klawiaturze.

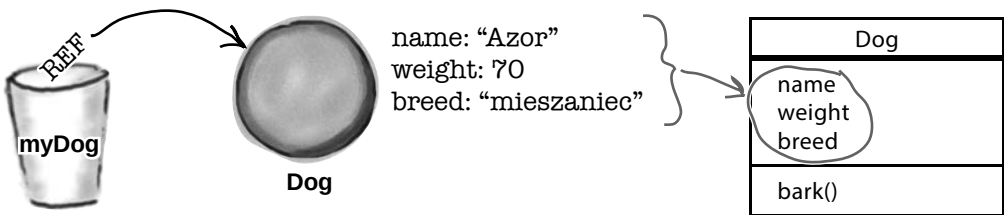
Jedna klasa

Dog
name weight breed
bark()

Wiele obiektów



Typy obiektowe są definiowane przy użyciu klas	92
Jak projektować własne klasy?	93
Zdefiniujmy klasę Dog	94
Jak utworzyć obiekt Dog?	95
Jak można odwoływać się do właściwości i funkcji?	96
Tworzymy aplikację Piosenki	97
Cud tworzenia obiektu	98
Jak są tworzone obiekty?	99
Za kulisami: Wywoływanie konstruktora	100
Dokładniejsze poznawanie właściwości	105
Elastyczna inicjalizacja właściwości	106
Jak używać bloków inicjalizatora?	107
Właściwości MUSZĄ zostać zainicjowane	108
Jak można weryfikować wartości właściwości?	111
Pisanie własnego akcesora get	112
Pisanie własnego akcesora set	113
Kompletny kod projektu Psy	115
Twój przybornik do Kotlinia	120

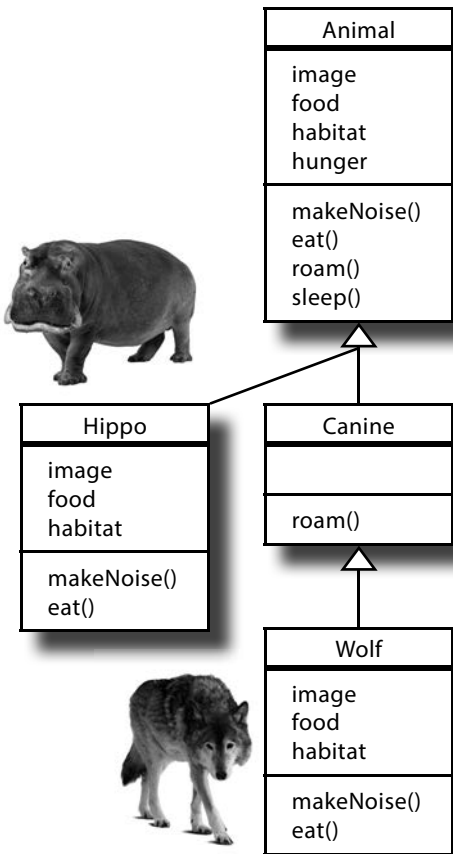


5

Klasy pochodne i bazowe
Stosowanie dziedziczenia

Czy kiedykolwiek zdarzyło Ci się uznać, że jakiś obiekt byłby wprost idealny do Twoich celów, gdyby tylko zmienić w nim parę rzeczy?

No cóż, to jest właśnie jedna z zalet **dziedziczenia**. W tym rozdziale pokażemy Ci, jak można tworzyć **klasy pochodne** i dziedziczyć właściwości i funkcje po **klasach bazowych**. Dowiesz się w nim, **jak przesłaniać funkcje i właściwości**, aby Twoje klasy działały tak, jak sobie tego życzysz, a przy okazji uzyskasz wiedzę na temat tego, kiedy to będzie właściwe (a kiedy nie). I w końcu przekonasz się, jak dziedziczenie pomaga w **unikaniu powielania kodu** oraz jak dzięki **polimorfizmowi** można poprawiać elastyczność.



Dziedziczenie umożliwia unikanie powielania kodu	122
Co mamy zamiar zrobić?	123
Projektujemy strukturę dziedziczenia klas zwierząt	124
Używaj dziedziczenia, by unikać powielania kodu w klasach pochodnych	125
Co powinny przesłaniać klasy pochodne?	126
Możemy pogrupować niektóre zwierzęta	127
Dodajemy klasy Canine i Feline	128
Sprawdź hierarchie klas, używając testu JEST	129
Test JEST działa w dowolnym miejscu drzewa dziedziczenia	130
Stworzymy parę kotliczyńskich zwierząt	133
Deklarujemy klasę bazową oraz jej właściwości i funkcje jako utworzone	134
Jak klasa pochodna dziedziczy po bazowej?	135
Jak (i kiedy) przesłaniać właściwości?	136
Przesłanianie właściwości pozwala na więcej niż tylko określanie wartości domyślnych	137
Jak przesłaniać funkcje?	138
Przesłonięte funkcje i właściwości pozostają utworzone...	139
Dodajemy klasę Hippo do projektu Zwierzęta	140
Dodajemy klasy Canine i Wolf	143
Która funkcja zostanie wywołana?	144
Kiedy wywołujemy funkcję na rzecz zmiennej, wywoływana jest funkcja obiektu	146
Typów bazowych można używać w parametrach funkcji i jako typu zwracanego wyniku	147
Zaktualizowany kod pliku Animals.kt	148
Twój przybornik do Kotliny	153

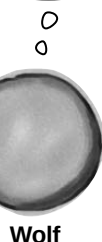
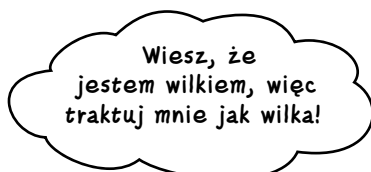
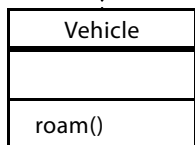
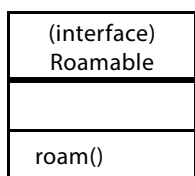
Klasy abstrakcyjne i interfejsy

Poważny polimorfizm

6

Hierarchia dziedziczenia klasy bazowej to dopiero początek.

Jeśli chcesz w *pełni wykorzystać możliwości, jakie zapewnia polimorfizm*, musisz zacząć projektować, używając **klas abstrakcyjnych** i **interfejsów**. W tym rozdziale odkryjesz, jak używać klas abstrakcyjnych, by kontrolować klasy projektowanej hierarchii i określać, które z nich będą **pozwalać na tworzenie obiektów, a które nie**. Przekonasz się, w jaki sposób mogą one wymuszać, by klasy pochodne **dostarczały własne implementacje funkcji**. Dowiesz się także, jak używać interfejsów do **współdzielenia zachowań pomiędzy niezależnymi klasami**. Przy okazji przekonasz się także, czym są: *is*, *as* oraz *when*.



Wolf

val Roamable

Hierarchia klasy Animal raz jeszcze	156
Obiektów niektórych klas po prostu nie powinno się tworzyć	157
Abstrakcyjna czy konkretna?	158
Klasy abstrakcyjne mogą mieć abstrakcyjne właściwości i funkcje	159
Klasa Animal ma dwie funkcje abstrakcyjne	160
Jak zaimplementować klasę abstrakcyjną?	162
TRZEBA zaimplementować wszystkie abstrakcyjne właściwości i funkcje	163
Zaktualizujemy kod projektu Zwierzęta	164
Niezależne klasy mogą mieć wspólne zachowania	169
Interfejsy pozwalają definiować wspólne zachowania	
POZA hierarchią klasy bazowej	170
Zdefiniujmy interfejs Roamable	171
Jak definiować właściwości w interfejsach?	172
Zadeklaruj, że klasa implementuje interfejs...	173
Jak implementować wiele interfejsów?	174
Jak określić, czy stworzyć klasę, klasę pochodną, klasę abstrakcyjną czy interfejs?	175
Aktualizujemy projekt Zwierzęta	176
Interfejsy pozwalają na stosowanie polimorfizmu	181
Gdzie używać operatora is?	182
Używaj when do porównywania zmiennej z grupą opcji	183
Operator is zazwyczaj wykonuje inteligentne rzutowanie	184
Używaj operatora as w celu jawnego rzutowania	185
Aktualizujemy projekt Zwierzęta	186
Twój przybornik do Kotlin	189

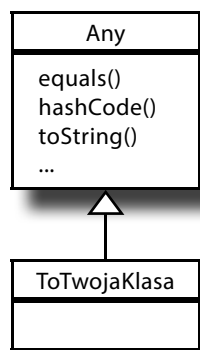
7

Klasy danych

Używanie danych

Nikt nie chce spędzać życia na ponownym wymyślaniu koła.

Większość aplikacji zawiera klasy, których jedynym zadaniem jest *przechowywanie danych*. Dlatego, aby ułatwić nam życie, twórcy Kotlina wymyślili pojęcie **klas danych**. W tym rozdziale dowiesz się, jak klasy danych pozwalają na pisanie kodu, który jest *bardziej przejrzysty i zwięzły*, niż mógłbyś to sobie wyobrazić. Poznasz w nim także *funkcje pomocnicze* klas danych i dowiesz się, jak przeprowadzać *destruktywizację obiektów danych na poszczególne komponenty*. A w międzyczasie dowiesz się, w jaki sposób *domyślne wartości parametrów* pozwalają pisać bardziej elastyczny kod, i poznasz **Any** — *matkę wszystkich klas bazowych*.



Domyślnie funkcja equals sprawdza, czy dwa porównywane obiekty są w rzeczywistości tym samym faktycznym obiektem.

Operator == wywołuje funkcję o nazwie equals	192
Funkcja equals pochodzi z klasy bazowej Any	193
Wspólne zachowania zdefiniowane w klasie Any	194
Możemy chcieć, by funkcja equals sprawdzała, czy obiekty są równoważne	195
Klasa danych pozwala na tworzenie obiektów danych	196
Klasy danych przesłaniają odziedziczone zachowania	197
Kopiowanie obiektów danych przy użyciu funkcji copy	198
Klasy danych definiują funkcje componentN...	199
Tworzymy projekt Przepisy	201
Pomieszczone komunikaty	203
Wygenerowane funkcje używają jedynie właściwości zdefiniowanych w konstruktorze	205
Inicjalizowanie wielu właściwości może powodować powstawanie kłopotliwego kodu	206
Jak używać domyślnych wartości konstruktora?	207
Także funkcje mogą używać wartości domyślnych	210
Przeciążanie funkcji	211
Zaktualizujemy projekt Przepisy	212
Ciąg dalszy kodu...	213
Twój przybornik do Kotlin	217

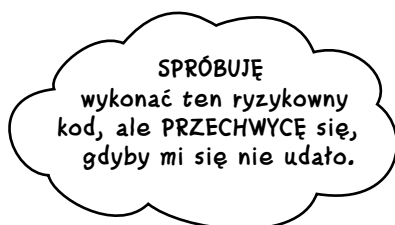
Wartość null i wyjątki

Cały i zdrowy

8

Nikt nie chce pisać kodu, który nie będzie bezpieczny.

Na szczęście mamy doskonałą wiadomość: Kotlin *został zaprojektowany pod kątem bezpieczeństwa kodu*. Zaczniemy od pokazania Ci, dlaczego stosowanie w Kotlinie **typów akceptujących wartości puste** oznacza, że *podczas swojego pobytu w Kotlinie prawie nigdy nie będziesz miał do czynienia z wyjątkami NullPointerException*. Dowiesz się także, jak wykonywać *bezpieczne wywołania* oraz jak to się dzieje, że dostępny w Kotlinie operator **Elvis** pozwoli Ci *spać spokojnie*. A kiedy już skończymy z wartościami pustymi, dowiesz się, jak **zgłaszać i obsługiwać wyjątki** jak stary wyjadacz.



Jak usuwać ze zmiennych referencje do obiektu?	220
Referencje usuwamy, używając null	221
Typów akceptujących null możesz używać wszędzie tam, gdzie typów, które null nie akceptują	222
Jak utworzyć tablice typu akceptującego null?	223
Jak odwoływać się do funkcji i właściwości typów akceptujących null?	224
Dbaj o bezpieczeństwo, używając bezpiecznych wywołań	225
Bezpieczne wywołania można łączyć w łańcuch	226
Ciąg dalszy historii...	227
Bezpiecznych wywołań można używać do przypisywania wartości...	228
Używaj let, by wykonywać kod, jeśli wartości będą różne od null	231
Stosowanie let z elementami tablic	232
Zamiast używać wyrażenia if...	233
Operator !! celowo zgłasza wyjątek NullPointerException	234
Tworzymy projekt Wartości Null	235
Ciąg dalszy kodu...	236
Wyjątki są zgłaszane w wyjątkowych sytuacjach	239
Przechwytyj wyjątki, używając try i catch	240
Użyj finally, by określić czynności, które mają być wykonane zawsze	241
Wyjątek to obiekt typu Exception	242
Wyjątki można zgłaszać jawnie	244
Try i catch to wyrażenia	245
Twój przyborek do Kotlinia	250

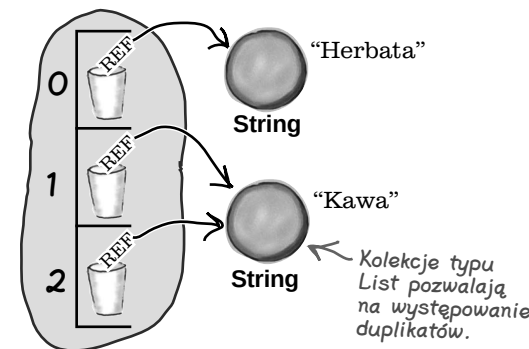
Kolekcje

Zorganizuj się!

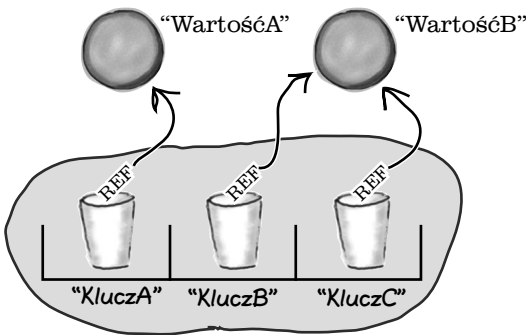
9

Czy kiedykolwiek marzyłeś o czymś elastyczniejszym niż tablice?

Kotlin udostępnia sporo użytecznych **kolekcji** zapewniających większą elastyczność i kontrolę nad sposobami **przechowywania i zarządzania grupami obiektów**. Może przydałaby Ci się **lista o zmiennej wielkości, do której bezustannie mógłbyś dodawać nowe elementy**? A może chciałbyś mieć możliwość sortowania kolekcji albo zapisywania jej elementów w odwrotnej lub losowej kolejności? Może chciałbyś móc **odnajdywać elementy na podstawie nazwy**? A może potrzebujesz kolekcji, która będzie **automatycznie odnajdywać i usuwać duplikaty** bez najmniejszej ingerencji z Twojej strony? Jeśli chciałbyś zyskać takie możliwości, a nawet jeszcze większe, to czytaj dalej... Znajdziesz tu wszystkie informacje...



List



Map

Kolekcje typu Map pozwalają na występowanie powtarzających się wartości, ale nie powtarzających się kluczy.

Tablice mogą być przydatne...	252
...są jednak rzeczy, których tablice nie potrafią	253
Jeśli masz wątpliwości, sięgnij do Biblioteki	254
List, Set oraz Map	255
Fantastyczne listy...	256
Tworzenie listy MutableList...	257
Możemy usunąć wartość...	258
Można zmieniać kolejność i operować na większych grupach elementów...	259
Utwórz projekt Kolekcje	260
Listy dopuszczają duplikaty	263
Jak tworzyć zbiory?	264
Jak zbiory wyszukują duplikaty?	265
Kody mieszające i równość	266
Reguły przesłaniania funkcji hashCode i equals	267
Jak używać klasy MutableSet?	268
Aktualizujemy projekt Kolekcje	270
Czas na mapy	276
Sposoby korzystania z map	277
Tworzenie map MutableMap	278
Z mapy MutableMap można usuwać elementy	279
Mapy Map i MutableMap można kopiować	280
Kompletny kod projektu Kolekcje	281
Wymieszane komunikaty	285
Twój przybornik do Kotlin	287

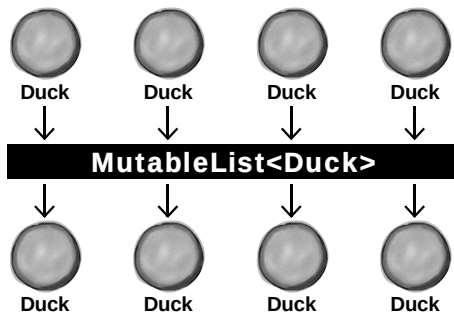
10

Typy sparametryzowane Odróżniaj wariację od kontrawariancji

Każdy lubi kod, który jest spójny.

A jednym ze sposobów pisania spójnego kodu, który będzie bardziej odporny na problemy, jest stosowanie *typów sparametryzowanych*. W tym rozdziale przyjrzymy się, w jaki sposób *kollekcje* Kotlina korzystają z *typów sparametryzowanych*, by uchronić nas przed zapisywaniem obiektów Kapusta w kolekcjach typu `List<Mewa>`. Dowiesz się, jak i kiedy *pisać własne klasy, interfejsy i funkcje sparametryzowane* oraz jak *ograniczyć typ parametru* do konkretnej klasy bazowej. I w końcu dowiesz się także, *jak używać kowariancji i kontrawariancji*, by **SAMEMU** kontrolować zachowania typów sparametryzowanych.

W razie zastosowania typów sparametryzowanych obiekty są zapisywane W kolekcji jedynie jako referencje do obiektów typu Duck...



...i są odczytywane z kolekcji także jako referencje do obiektów typu Duck.

Vet<T: Pet>
treat(t: T)



Kolekcje używają typów sparametryzowanych	290
Jak jest zdefiniowany typ MutableList?	291
Stosowanie parametrów typów w MutableList	292
Co można robić ze sparametryzowanymi klasami i interfejsami?	293
Oto co mamy zamiar zrobić	294
Tworzymy hierarchię klasy Pet	295
Definiujemy klasę Contest	296
Dodajemy właściwość scores	297
Tworzymy funkcję getWinners	298
Tworzymy kilka obiektów Contest	299
Tworzymy projekt TypySparametryzowane	301
Hierarchia typu Retailer	305
Definiujemy interfejs Retailer	306
Możemy tworzyć obiekty CatRetailer, DogRetailer i FishRetailer...	307
Użyj out, by typ sparametryzowany był kowariantny	308
Aktualizujemy projekt TypySparametryzowane	309
Potrzebujemy klasy Vet	313
Tworzenie obiektów Vet	314
Użyj in, by typ sparametryzowany był kontrawariantny	315
Typ sparametryzowany może być lokalnie kontrawariantny	316
Aktualizujemy projekt TypySparametryzowane	317
Twój przybornik do Kotlina	324

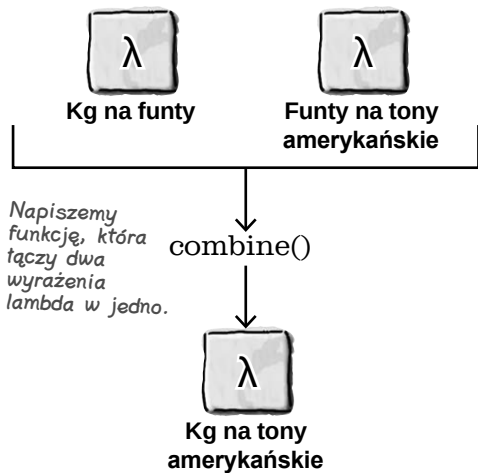
11

Lambdy i funkcje wyższego rzędu

Kod używany jak dane

Chciałbyś pisać kod, który jest jeszcze bardziej potężny i elastyczny?

Jeśli chcesz, będziesz do tego celu potrzebował **lambd**. *Lambdy*, czy też *wyrażenia lambda*, to bloki kodu, które można przekazywać jak zwyczajne obiekty. W tym rozdziale dowiesz się, **jak definiować lambdy**, **jak zapisywać je w zmiennych** oraz **jak je wykonywać**. Dowiesz się także o **typach funkcyjnych** oraz w jaki sposób pomagają one w tworzeniu **funkcji wyższego rzędu**, czyli funkcji, których parametrami lub wartościami wynikowymi są lambdy. Przy okazji przekonasz się także, że nieco *syntaktycznego cukru może osłodzić Twoje programistyczne życie*.



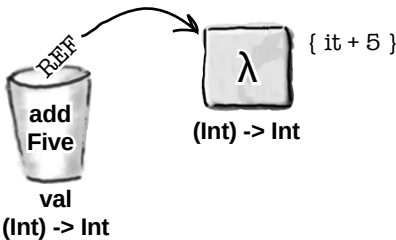
Mam dwa parametry typu Int o nazwach x i y. Zwracam sumę ich wartości.



Lambda

xvi

Prezentacja lambd	326
Jak wygląda kod lambdy?	327
Lambdy można zapisywać w zmiennych	328
Wyrażenia lambda mają typ	331
Kompilator może wywnioskować typ parametrów lambdy	332
Używaj lambdy dostosowanej do typu zmiennej	333
Tworzenie projektu Lambdy	334
Możesz przekazać lambdę do funkcji	339
Wywołanie lambdy w ciele funkcji	340
Co się dzieje podczas wywołania funkcji?	341
Lambdę można przenieść poza nawiasy ()...	343
Aktualizujemy projekt Lambdy	344
Funkcje mogą zwracać lambdy	347
Piszemy funkcję, która pobiera i zwraca lambdę	348
Jak można używać funkcji combine?	349
Użyj typealias, by nadać inną nazwę istniejącemu typowi danych	353
Aktualizujemy kod projektu	354
Twój przybornik do Kotlina	361



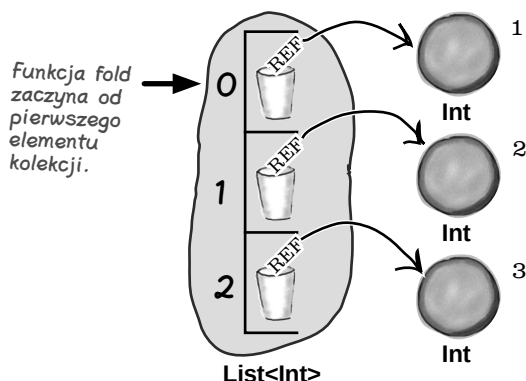
Wbudowane funkcje wyższego rzędu

Wzmocnij swój kod

12

Kotlin zawiera wiele wbudowanych funkcji wyższego rzędu.

A w tym rozdziale przedstawimy kilka najbardziej użytecznych spośród nich. Zapoznasz się tu z elastyczną **rodziną funkcji filter** i dowiesz, jak mogą Ci pomóc w ograniczaniu rozmiarów kolekcji. Dowiesz się, jak **przekształcać kolekcje, używając funkcji map**, jak **przeglądać wszystkie elementy kolekcji, używając funkcji forEach** oraz jak **grupować elementy kolekcji, używając funkcji groupBy**. Użyjesz nawet funkcji **fold** do wykonywania złożonych obliczeń w jednym wierszu kodu. Pod koniec tego rozdziału będziesz potrafił pisać kod, który będzie potężniejszy, niżbyś kiedykolwiek przypuszczał.



Kotlin udostępnia sporo funkcji wyższego rzędu	364
Funkcje min i max operują na prostych typach danych	365
Bliższe spojrzenie na parametry funkcji minBy i maxBy	366
Funkcje sumBy i sumByDouble	367
Tworzymy projekt Spożywcza	368
Poznaj funkcję filter	371
Używaj funkcji map, by przekształcać kolekcje	372
Co się dzieje podczas wykonywania kodu?	373
Ciąg dalszy historii...	374
Funkcja forEach działa jak pętla	375
Funkcja forEach nie zwraca wyniku	376
Aktualizujemy projekt Spożywcza	377
Użyj groupBy, by podzielić kolekcję na grupy	381
Funkcji groupBy można używać w łańcuchach wywołań	382
Jak używać funkcji fold	383
Za kulisami: funkcja fold	384
Więcej przykładów użycia funkcji fold	386
Aktualizujemy projekt Spożywcza	387
Pomieszczone komunikaty	391
Twój przybornik do Kotlinia	394
Wyjeżdżamy...	395

Koprocedury

Współbieżne wykonywanie kodu

A

Niektóre rzeczy najlepiej jest wykonywać w tle.

Jeśli musisz odczytać dane z wolnego serwera zewnętrznego, to najprawdopodobniej nie będziesz chciał, by reszta kodu czekała bezczynnie na zakończenie takiej operacji. W takich sytuacjach Twoim **najlepszym przyjacielem są koprocedury**. Koprocedury pozwalają na pisanie kodu, który jest **wykonywany asynchronicznie**. To z kolei oznacza *mniej czasu straconego na oczekiwanie, lepsze doznania użytkownika*, jak również *możliwość poprawienia skalowalności aplikacji*. Czytaj dalej, a poznasz sekret, jak rozmawiać z Kubą, jednocześnie słuchając Zuzanny.



Bam! Bam! Bam! Bam! Bam! Bam!
Tss! Tss!

↖ Dźwięki tom-tomu i talerzy są odtwarzane jednocześnie, jednak teraz używamy bardziej wydajnego sposobu odtwarzania plików dźwiękowych.

Testowanie

Pociągnij swój kod do odpowiedzialności.

B

Wszyscy wiedzą, że dobry kod musi działać.

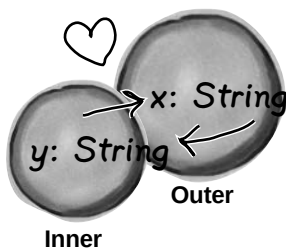
Jednak każda zmiana wprowadzana w kodzie pociąga za sobą ryzyko wystąpienia błędów, które uniemożliwią jego prawidłowe działanie. To właśnie dlatego konieczne jest *dokładne testowanie*: dzięki niemu dowiesz się o wszelkich problemach występujących w kodzie, jeszcze *zanim zostanie wdrożony w środowisku produkcyjnym*. W tym dodatku przedstawimy **JUnit** oraz **KotlinTest** — dwie biblioteki, których możesz używać do **tworzenia testów jednostkowych i sprawdzania za ich pomocą swojego kodu**, dzięki czemu *zawsze będziesz miał pod ręką koło ratunkowe*.

Pozostałości

Dziesięć najważniejszych rzeczy (których nie opisaliśmy)

Choć opisaliśmy już tak dużo, to wciąż jeszcze trochę zostało.

Jest jeszcze parę rzeczy, o których warto wiedzieć. Nie czulibyśmy się w porządku, gdybyśmy pominęli je bez słowa, a naprawdę zależy nam na tym, by oddać w Twoje ręce książkę, po której przeczytaniu nie będziesz musiał chodzić na lokalną siłownię. Dlatego zanim odłożysz tę książkę na półkę, **przeczytaj to, co jeszcze zostało.**



Obiekty Inner i Outer są ze sobą powiązane w szczególny sposób. Klasa Inner może używać składowych klasy Outer i na odwrót.

1. Pakiety i importowanie	416
2. Modyfikatory widoczności	418
3. Klasy wyliczeniowe	420
4. Klasy zapieczętowane	422
5. Klasy zagnieżdżone i wewnętrzne	424
6. Deklaracje obiektów i wyrażenia	426
7. Rozszerzenia	429
8. Return, break i continue	430
9. Więcej zabawy z funkcjami	432
10. Współdziałanie	434

1. Zaczynamy

Szybki skok

Wskakuj, woda jest świetna!
Poptyniemy od razu na głęboką
wodę, napiszemy trochę kodu
i przyjrzymy się podstawom
składni Kotlin. Nawet się nie
zorientujesz, kiedy zaczniesz
kodować.



Kotlin robi dużo szumu.

Od momentu pojawienia się pierwszej wersji Kotlinu język ten urzekł programistów swoją **przyjazną składnią, spójnością, elastycznością oraz możliwościami**. W tej książce nauczymy Cię **pisać w Kotlinie własne aplikacje**, a zaczniemy od pokazania, jak stworzyć i uruchomić prostą aplikację. Po drodze przedstawimy podstawy składni Kotlinu, takie jak *instrukcje, pętle* oraz *konstrukcje warunkowe*. Twoja podróż właśnie się zaczyna...

Witamy w Kotliczynie

Kotlin błyskawicznie podbija świat programowania. Choć jest to jeden z najmłodszych języków dostępnych w programistycznym świecie, to jednak wielu programistów go preferuje. Co zatem sprawia, że jest on tak wyjątkowy?

Kotlin dysponuje wieloma cechami nowoczesnych języków programowania, które sprawiają, że jest on atrakcyjny dla programistów. Cechy te przedstawimy bardziej szczegółowo w dalszej części książki, a na razie tylko ogólnie opiszemy kilka z nich.

Kotlin jest zwięzły, zrozumiały i czytelny

W odróżnieniu od niektórych innych języków kod pisany w Kotlinie jest bardzo zwięzły i pozwala na wykonywanie złożonych operacji nawet w jednym wierszu kodu. Udostępnia on skrócone sposoby realizacji najczęściej wykonywanych czynności, dzięki czemu nie trzeba pisać długich, powtarzających się fragmentów kodu, jak również dysponuje obszerną biblioteką funkcji, których można używać. Co więcej, ponieważ jest mniej kodu, przez który trzeba się przedzierać, kod w Kotlinie można szybciej czytać, pisać i analizować, dzięki czemu pozostaje nam więcej czasu na inne rzeczy.

Pozwala korzystać z programowania obiektowego i funkcyjnego

Nie możesz się zdecydować, czy chcesz uczyć się programowania obiektowego czy funkcyjnego? A czemu by nie obu tych sposobów programowania jednocześnie? Kotlin pozwala na pisanie kodu obiektowego korzystającego z klas, dziedziczenia i polimorfizmu, dokładnie tak samo jak Java. Jednocześnie daje także możliwość programowania w stylu funkcyjnym, łącząc w sobie najlepsze cechy obu tych światów.

Kompilator zapewnia nam bezpieczeństwo

Nikt nie lubi niebezpiecznego kodu rojącego się od błędów, a kompilator Kotliny wkłada wiele wysiłku w to, by zapewnić, że tworzony kod będzie możliwie jak najbardziej przejrzysty, i zapobiega wielu błędom, które mogłyby się pojawić w innych językach programowania. Kotlin jest językiem korzystającym ze statycznego określania typów, dzięki czemu nie możemy na przykład wykonać nieodpowiedniej czynności dla danego typu danych, doprowadzając przez to do awarii programu. Co więcej, w większości przypadków nie trzeba nawet jawnie podawać typów, gdyż kompilator może je określać samodzielnie.

A zatem Kotlin jest nowoczesnym, potężnym i elastycznym językiem programowania, zapewniającym bardzo dużo korzyści. Ale to jeszcze nie koniec historii.



Kotlin niemal w całości eliminuje wiele rodzajów błędów, które powszechnie występują w innych językach programowania. A to oznacza solidniejszy, bardziej niezawodny kod i mniej czasu spędzonego na tropieniu i usuwaniu błędów.

Kotlina można używać niemal wszędzie

Kotlin jest tak potężny i elastyczny, że można go używać w bardzo wielu kontekstach, jako języka ogólnego przeznaczenia. Jest to możliwe dlatego, że pozwala on **wybrać platformę docelową, na którą będzie kompilowany jego kod źródłowy**.

Wirtualne maszyny Javy (JVM)

Kod w Kotlinie można kompilować do postaci kodów bajtowych Javy i wykonywać w wirtualnej maszynie Javy (*JVM — Java Virtual Machine*). Oznacza to, że Kotlin można używać praktycznie wszędzie tam, gdzie Javy. Kotlin zapewnia stuprocentowe współdziałanie z Javą, dzięki czemu można w nim używać już istniejących bibliotek napisanych w Javie. Jeśli pracujemy nad aplikacją zawierającą wiele kodu w Javie, nie musimy wyrzucać tego kodu — nowy kod napisany w Kotlinie będzie z nim doskonale współdziałał. A jeśli zechcemy używać kodu napisanego w Kotlinie w kodzie Javy, to z tym także nie będziemy mieć żadnego problemu.

Android

Wraz z innymi językami, takimi jak Java, Kotlin ma pełne wsparcie ze strony twórców Androida. Można go używać do pisania w Android Studio, jak również korzystać z jego wszystkich zalet podczas pisania aplikacji na Androida.

JavaScript — po stronie klienta i serwera

Można także tłumaczyć i kompilować (czyli „transpilować”) kod napisany w Kotlinie do postaci kodu napisanego w JavaScriptcie i wykonywać na przykład w przeglądarkach WWW. Takiego kodu można używać do korzystania zarówno z technologii serwerowych, jak i klienckich, takich jak Node.js czy WebGL.

Aplikacje rodzime

Jeśli chcemy pisać kod, który będzie błyskawicznie wykonywany na słabszych urządzeniach, to kod w Kotlinie można kompilować bezpośrednio do rodzimego kodu maszynowego. Dzięki temu można pisać kod, który będzie działał na przykład na iOS-ie lub Linuksie.

W tej książce skoncentrujemy się na tworzeniu w Kotlinie aplikacji przeznaczonych do wykonywania w JVM, gdyż to najprostszy sposób na opanowanie możliwości tego języka. Później jednak będziesz mógł zastosować tę wiedzę do pisania kodu działającego także na innych platformach.

A zatem dajmy nurka na głęboką wodę.

Możliwość wyboru platformy docelowej, na którą zostanie skompilowany kod napisany w Kotlinie, oznacza, że można go uruchamiać na serwerach, w chmurze, w przeglądarce, na urządzeniach mobilnych i nie tylko.



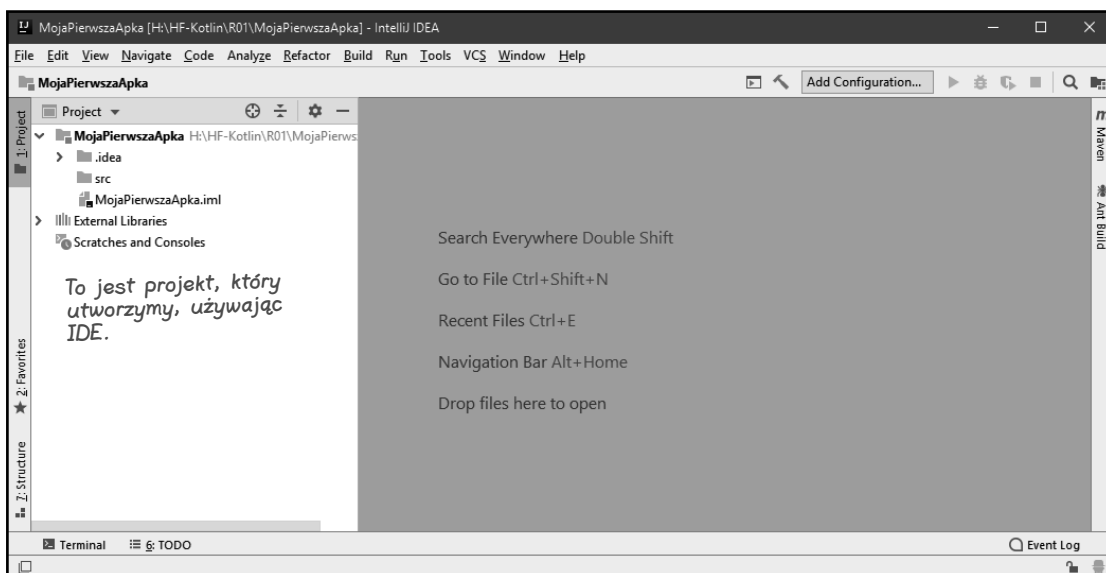
Choć będziemy pisać aplikacje działające w JVM, to by w pełni skorzystać z tej książki, wcale nie musisz znać Javy. Zakładamy co prawda, że dysponujesz pewnym ogólnym doświadczeniem z zakresu programowania, ale to wszystko.

Co zrobimy w tym rozdziale?

W tym rozdziale pokażemy Ci, jak napisać w Kotlinie bardzo prostą aplikację.

W tym celu trzeba będzie wykonać kilka czynności:

- 1 Utworzyć nowy projekt aplikacji w Kotlinie.**
Zacniemy od zainstalowania IntelliJ IDEA (Community Edition), bezpłatnego zintegrowanego środowiska programistycznego (IDE) obsługującego pisanie aplikacji w języku Kotlin. Następnie użyjemy go do utworzenia projektu aplikacji w Kotlinie:



- 2 Dodać funkcję, która wyświetla jakiś tekst.**
Do projektu aplikacji dodamy nowy plik źródłowy Kotlinu i napiszemy prostą funkcję `main`, która wyświetla tekst „Hau!”.
- 3 Zaktualizować funkcję, tak by robiła nieco więcej.**
Kotlin udostępnia podstawowe konstrukcje językowe, takie jak instrukcje, pętle oraz rozgałęzienia warunkowe. Skorzystamy z nich, by nasza funkcja robiła coś więcej.
- 4 Wypróbować kod w interaktywnej powłoce Kotlinu.**
I w końcu dowiemy się, w jaki sposób można sprawdzać fragmenty kodu w interaktywnej powłoce Kotlinu (REPL).

Już zaraz zainstalujemy IDE, najpierw jednak wykonaj pewne ćwiczenie.



Zaostrz ołówek

Wiemy, że jeszcze nie pokazaliśmy Ci żadnego kodu napisanego w Kotlinie, przekonajmy się jednak, czy będziesz potrafił odgadnąć, co robią poszczególne wiersze poniższego fragmentu kodu. Aby nieco Ci to ułatwić, opiszemy pierwszy wiersz kodu.

```
val name = "Zouza" ..... Deklaruje zmienną o nazwie „name” i nadaje jej wartość „Zouza”.
val height = 27 .....

println("Siema") .....
println("Mój kot nazywa się $name") .....
println("Mój kot ma $height centymetrów wysokości") .....

val a = 6 .....
val b = 7 .....
val c = a + b + 10 .....
val str = c.toString() .....

val numList = arrayOf(1, 2, 3) .....
var x = 0 .....
while (x < 3) { .....
    println("Elementem nr $x jest ${numList[x]}") .....
    x = x + 1 .....
} .....

val myCat = Cat(name, height) .....
val y = height - 3 .....
if (y < 5) myCat.miaow(4) .....

while (y < 26) { .....
    myCat.play() .....
    y = y + 1 .....
} .....
```

Zaostrz ołówek
Rozwiązanie

Wiemy, że jeszcze nie pokazaliśmy Ci żadnego kodu napisanego w Kotlinie, przekonajmy się jednak, czy będziesz potrafił odgadnąć, co robią poszczególne wiersze poniższego fragmentu kodu. Aby nieco Ci to ułatwić, opiszemy pierwszy wiersz kodu.

```
val name = "Zouza"    Deklaruje zmienną o nazwie „name” i nadaje jej wartość „Zouza”.
val height = 27       Deklaruje zmienną o nazwie „height” i nadaje jej wartość 27.

println("Siema")      Wyświetla „Siema” na standardowym wyjściu.
println("Mój kot nazywa się $name")  Wyświetla „Mój kot nazywa się Zouza”.
println("Mój kot ma $height centymetrów wysokości")  Wyświetla „Mój kot ma 27 centymetrów wysokości”.

val a = 6             Deklaruje zmienną o nazwie „a” i nadaje jej wartość 6.
val b = 7             Deklaruje zmienną o nazwie „b” i nadaje jej wartość 7.
val c = a + b + 10     Deklaruje zmienną o nazwie „c” i nadaje jej wartość 23.
val str = c.toString() Deklaruje zmienną o nazwie „str” i przypisuje jej tekstową wartość „23”.

val numList = arrayOf(1, 2, 3)  Tworzy tablicę zawierającą wartości: 1, 2 i 3.
var x = 0                    Deklaruje zmienną o nazwie „x” i nadaje jej wartość 0.
while (x < 3) {              Kaze wykonywać pętlę, dopóki x jest mniejsze od 3.
    println("Elementem nr $x jest ${numList[x]}")  Wyświetla indeks i wartość każdego elementu tablicy.
    x = x + 1                Dodaje 1 do x.
}                             To jest koniec pętli.

val myCat = Cat(name, height) Deklaruje zmienną o nazwie „myCat” i tworzy obiekt Cat.
val y = height - 3          Deklaruje zmienną o nazwie „y” i nadaje jej wartość 24.
if (y < 5) myCat.miaow(4)    Jeśli y jest mniejsze od 5, obiekt Cat ma miauknąć 4 razy.

while (y < 26) {             Kaze wykonywać pętlę, dopóki y jest mniejsze od 26.
    myCat.play()             Wywołuje funkcję play() obiektu myCat.
    y = y + 1                Dodaje 1 do y.
}                             To jest koniec pętli.
```

Instalowanie IntelliJ IDEA (Community Edition)

Najprostszym sposobem pisania i wykonywania kodu w języku Kotlin jest skorzystanie z IntelliJ IDEA (Community Edition). To darmowe zintegrowane środowisko programistyczne (IDE) zostało stworzone przez firmę JetBrains, która opracowała także sam język Kotlin i dysponuje wszystkim, czego potrzeba do pisania aplikacji w Kotlinie, w tym:

Jesteś tutaj

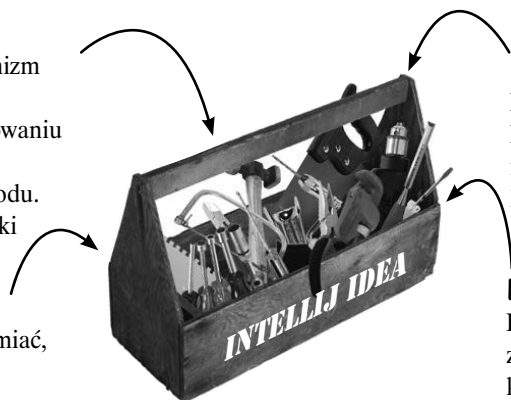
- ☐ Zbudowanie aplikacji
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

Edytorem kodu

Edytor jest wyposażony w mechanizm uzupełniania kodu, który pomaga w pisaniu kodu w Kotlinie, kolorowaniu składni i podkreślaniu kolorami ułatwiającymi czytanie i analizę kodu. Potrafi także pokazywać wskazówki dotyczące usprawniania kodu.

Narzędziami do budowy

Kod można kompilować i uruchamiać, korzystając z łatwo dostępnych skrótów.



IntelliJ IDEA ma wiele innych możliwości i narzędzi, a wszystkie ułatwiają życie programistom.

Kotlin REPL

IntelliJ IDEA zawiera narzędzie Kotlin REPL, które pozwala na testowanie fragmentów kodu poza główną bazą kodu projektu.

Kontrolą wersji

IntelliJ IDEA potrafi współpracować ze wszystkimi głównymi systemami kontroli wersji, takimi jak: Git, SVN, CVS i inne.

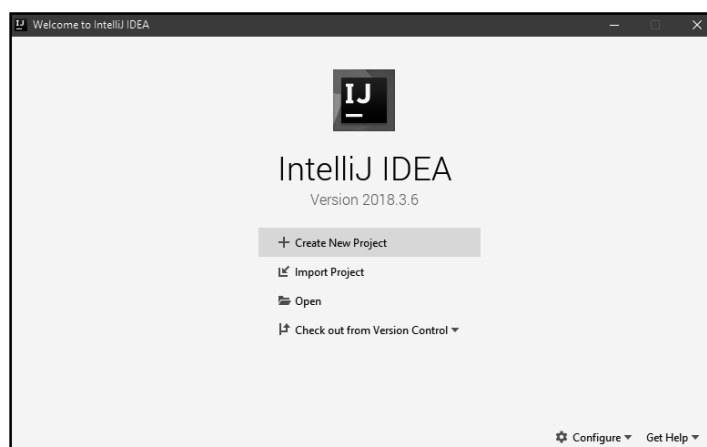
Abyś mógł razem z nami wykonywać przykłady i ćwiczenia prezentowane w książce, musisz zainstalować IntelliJ IDEA (Community Edition). Program ten można pobrać ze strony:

<https://www.jetbrains.com/idea/download/index.html>

Konieczniewybraćopcję dopobrania bezpłatnej wersji Community Edition IntelliJ IDEA.

Po zainstalowaniu IDE uruchom je. Na ekranie zostanie wyświetlone powitalne okno dialogowe. Teraz masz już wszystko, czego Ci potrzeba do napisania pierwszej aplikacji w Kotlinie.

To jest powitalne okno dialogowe IntelliJ IDEA.



Stwórzmy prostą aplikację

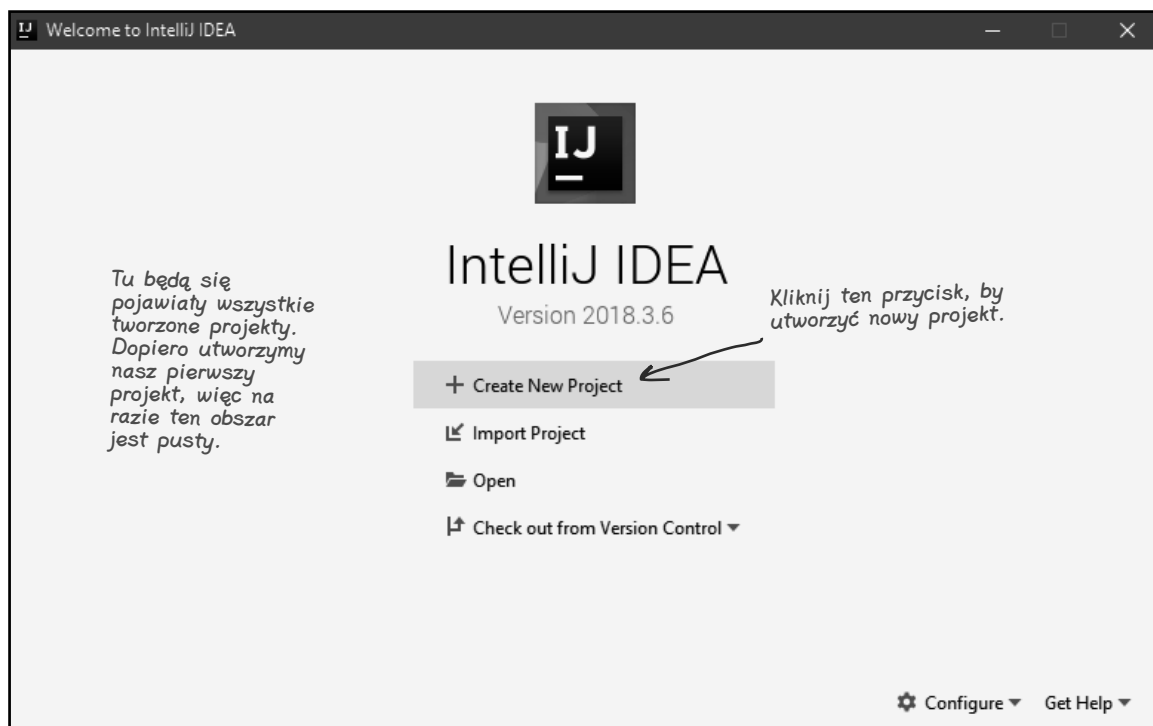
Skoro już masz zintegrowane środowisko programistyczne, jesteś gotów, by napisać swoją pierwszą aplikację w Kotlinie. Będzie to naprawdę bardzo prosta aplikacja, której działanie sprowadza się do wyświetlenia na ekranie tekstu „Hau!”.

Pisanie aplikacji w IntelliJ IDEA zawsze należy zacząć od utworzenia nowego projektu. Upewnij się zatem, że IDE działa, i wykonuj wraz z nami opisywane czynności.

- ☐ **Zbudowanie aplikacji**
- ☐ **Dodanie funkcji**
- ☐ **Aktualizacja funkcji**
- ☐ **Zastosowanie REPL**

1. Utworzenie projektu

Powitalne okno dialogowe IntelliJ IDEA udostępnia kilka opcji pozwalających określić, co chcemy zrobić. Ponieważ chcemy utworzyć nowy projekt, musimy kliknąć przycisk *Create New Project*.



Tworzenie prostej aplikacji (ciąg dalszy)

2. Określenie typu projektu

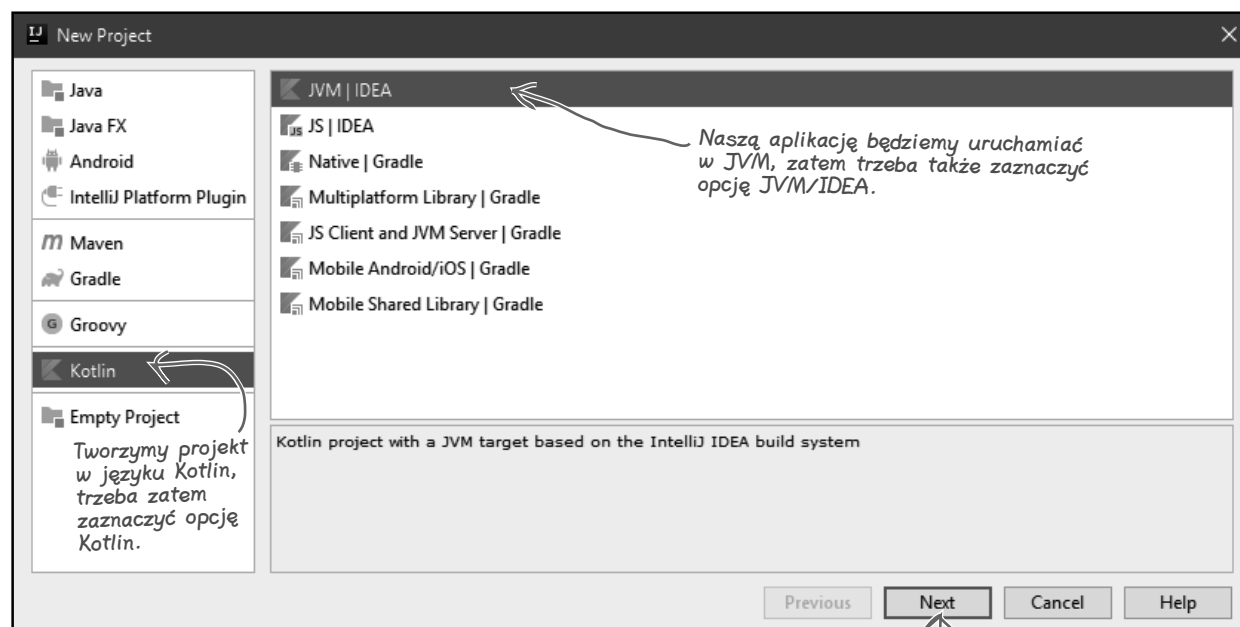
Teraz będziesz musiał powiedzieć IntelliJ IDEA, jakiego rodzaju projekt chcesz utworzyć.

IntelliJ IDEA pozwala na tworzenie projektów pisanych w różnych językach i przeznaczonych na różne platformy, na przykład pisanych w Javie aplikacji na Androida. Nam chodzi o utworzenie projektu w Kotlinie, więc zaznacz opcję *Kotlin*.

Oprócz tego trzeba określić, jaka ma być docelowa platforma projektu. My utworzymy aplikację przeznaczoną do wykonywania wirtualnej maszynie Javy, a zatem musisz także zaznaczyć opcję *JVM/IDEA*.

← Dostępne są także inne opcje, jednak w tej książce skoncentrujemy się na tworzeniu aplikacji przeznaczonych na wirtualną maszynę Javy (JVM).

- ☐ Zbudowanie aplikacji
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL



Kliknij Next, by przejść do następnego kroku.

Tworzenie prostej aplikacji (ciąg dalszy)

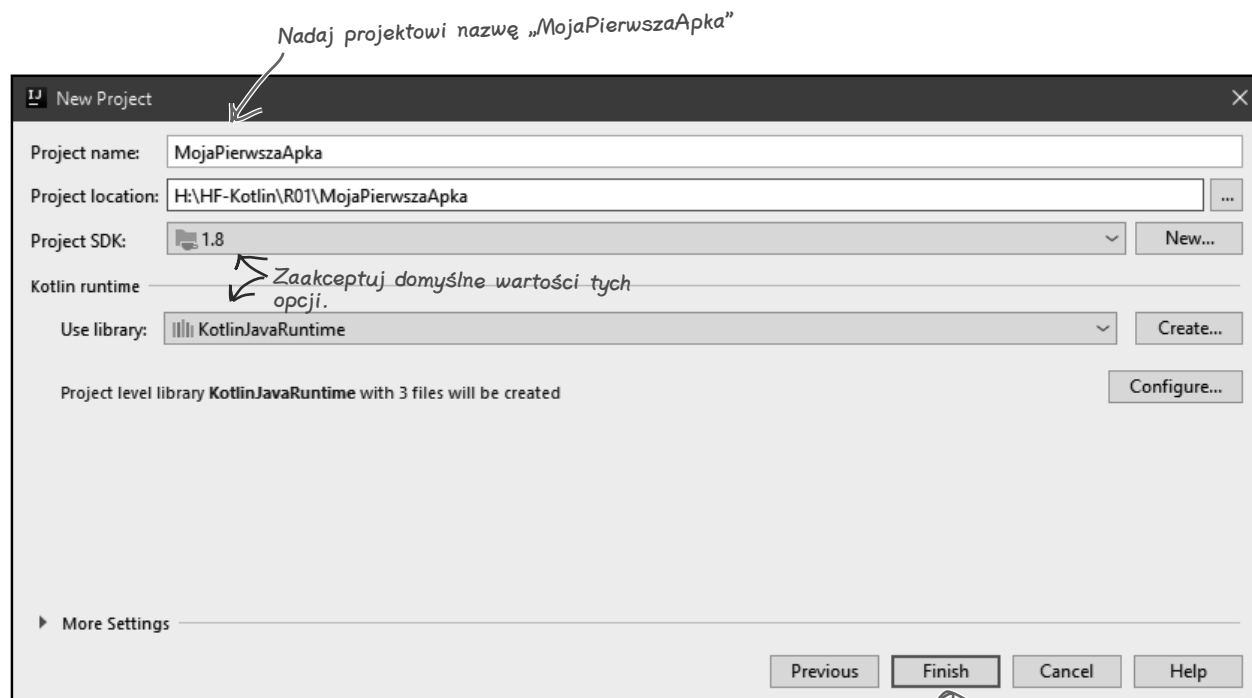
- ☐ **Zbudowanie aplikacji**
- ☐ **Dodanie funkcji**
- ☐ **Aktualizacja funkcji**
- ☐ **Zastosowanie REPL**

3. Konfiguracja projektu

Teraz musisz skonfigurować projekt — określić, jak ma się on nazywać, gdzie mają być przechowywane jego pliki oraz jakie pliki mają być przez niego używane. Dostępne opcje pozwalają także określić, jakiej wersji Javy ma używać JVM oraz jakiej biblioteki Kotlina należy użyć.

Nadaj projektowi nazwę *MojaPierwszaApka* i zaakceptuj domyślne ustawienia wszystkich innych opcji.

Po kliknięciu przycisku *Finish* IntelliJ IDEA utworzy projekt.

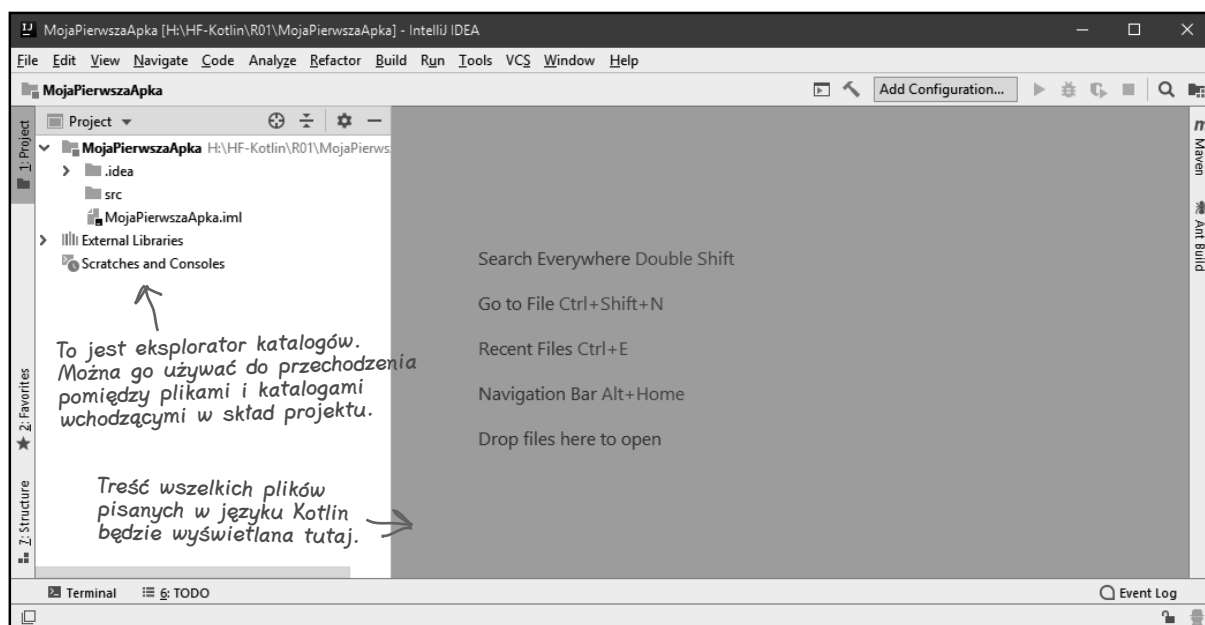


Właśnie utworzyłeś swój pierwszy projekt w Kotlinie

Po wykonaniu wszystkich czynności związanych z tworzeniem nowego projektu IntelliJ IDEA przygotuje go i wyświetli. Tak wygląda projekt przygotowany dla nas przez IDE:

Właśnie zakończyliśmy ten etap prac, zaznaczamy go zatem jako wykonany.

- ☒ **Zbudowanie aplikacji**
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

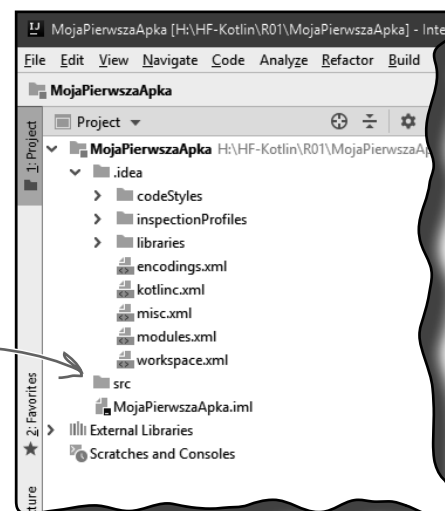


Jak widać, IntelliJ IDEA udostępnia narzędzie eksploratora, którego można używać do poruszania się po plikach i katalogach wchodzących w skład projektu. IDE tworzy tę strukturę katalogów podczas zakładania projektu.

W skład struktury katalogów wchodzi pliki konfiguracyjne, używane zarówno przez samo IDE, jak i przez niektóre biblioteki zewnętrzne stosowane w aplikacji. Znajduje się w niej także katalog `src`, używany do przechowywania kodów źródłowych. Większość naszego pobytu w Kotlinicynie spędzimy właśnie, pracując w tym katalogu.

Na razie katalog `src` jest jeszcze pusty, gdyż nie dodaliśmy do niego żadnego pliku źródłowego. Już niebawem się tym zajmiemy.

Wszelkie pliki źródłowe pisane w Kotlinie muszą być dodawane do katalogu `src`.

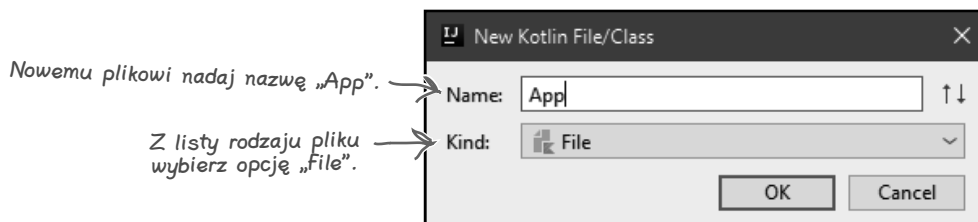


Dodaj do projektu nowy plik Kotlina

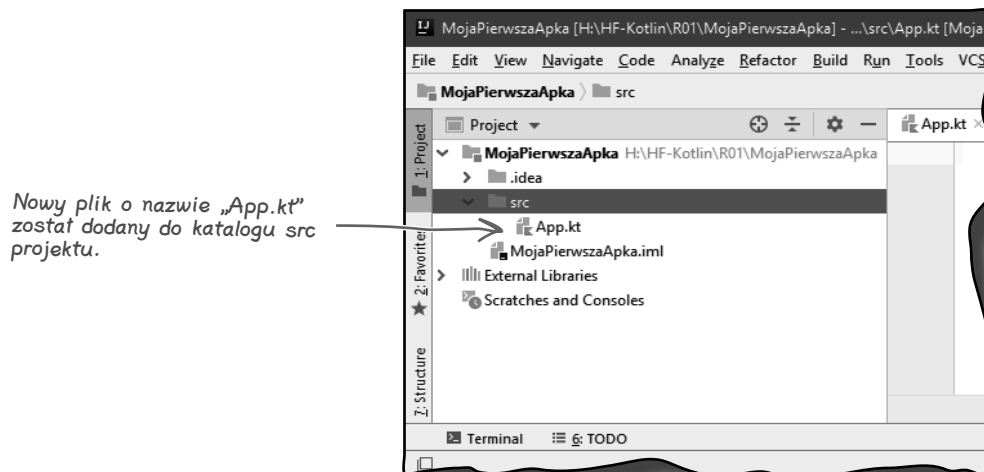
Zanim będziesz mógł zacząć pisać kod w Kotlinie, musisz utworzyć dla niego plik źródłowy.

Aby dodać do projektu nowy plik źródłowy języka Kotlin, musisz zaznaczyć katalog *src* w eksploratorze IntelliJ IDEA, kliknąć w menu głównym opcję *File*, a następnie wybrać opcję *New/Kotlin File/Class*. W efekcie zostaniesz poproszony o podanie nazwy i typu pliku. W polu *Name* wpisz *App*, a z listy *Kind* wybierz opcję *File*, jak pokazaliśmy na poniższym rysunku:

- ☒ Zbudowanie aplikacji
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL



Po kliknięciu przycisku *OK* IntelliJ IDEA utworzy nowy plik Kotlin o nazwie *App.kt* i doda go do katalogu *src* projektu:



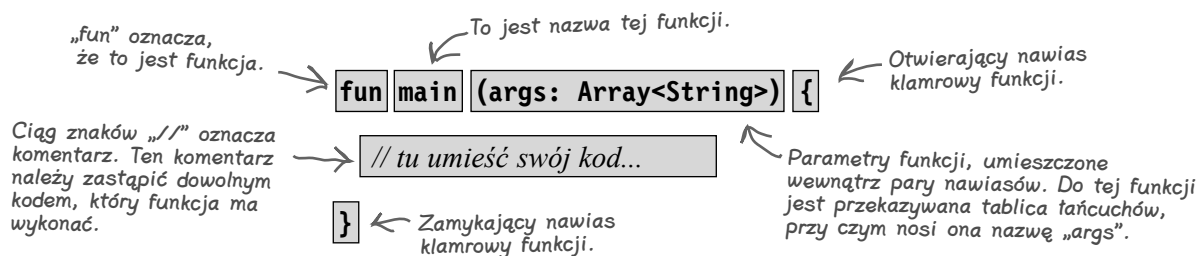
Zobaczmy teraz, jaki kod musimy dodać do pliku *App.kt*, aby zaczął on cokolwiek robić.

Anatomia funkcji main

Aktualnie chodzi nam o napisanie w Kotlinie kodu, który w oknie wyników IDE wyświetli łańcuch „Hau!”. W tym celu dodamy do pliku *App.kt* funkcję.

Pisząc aplikację w języku Kotlin, zawsze *musisz* dodać do niej funkcję o nazwie `main`, której zadaniem jest uruchomienie tej aplikacji. Podczas uruchamiania kodu JVM szuka tej funkcji i wykonuje ją.

Funkcja `main` wygląda tak jak na poniższym przykładzie:



Funkcja rozpoczyna się od słowa **fun**, które informuje kompilator Kotlin, że to jest funkcja. Słowa kluczowego `fun` musisz używać podczas tworzenia w Kotlinie każdej nowej funkcji.

Za słowem kluczowym `fun` zostaje podana nazwa funkcji — w tym przypadku jest to **main**. Nadanie funkcji tej nazwy oznacza, że funkcja ta zostanie automatycznie wykonana po uruchomieniu aplikacji.

Kod umieszczony wewnątrz nawiasów `()` podanych za nazwą funkcji informuje kompilator, jakie argumenty funkcja będzie akceptować (o ile w ogóle jakieś będą). W naszym przypadku kod `args: Array<String>` określa, że funkcja przyjmuje tablicę łańcuchów (`String`) oraz że tablica ta będzie dostępna pod nazwą `args`.

Kod, który chcemy wykonywać, musi być umieszczony pomiędzy nawiasami klamrowymi `{ }` funkcji `main`. W naszym przypadku chcemy, by funkcja ta wyświetliła w IDE łańcuch „Hau!”. Możemy to zrobić w następujący sposób:

Ta nazwa informuje, że tekst należy wyświetlić na standardowym wyjściu.

```
fun main(args: Array<String>) {
    println("Hau!")
}
```

To jest tekst, który chcemy wyświetlić.

Funkcja `println("Hau!")` wyświetla łańcuch znaków, czy też `String`, na standardowym wyjściu. Ponieważ nasz program będziemy wykonywać w IDE, tekst zostanie wyświetlony w jego panelu wyników.

Skoro już wiemy, jak wygląda funkcja, dodajmy ją do naszego projektu.

- ☒ Zbudowanie aplikacji
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

Bezparametrowe funkcje main



Jeśli używasz Kotlin w wersji 1.2 lub starszej, to aby można było uruchomić aplikację, funkcja `main` *musi* przyjąć następującą postać:

```
fun main(args: Array<String>) {
    // tu umieść swój kod
}
```

Jednak zaczynając od wersji 1.3, parametry funkcji `main` można pomijać, a zatem może ona wyglądać jak na poniższym przykładzie:

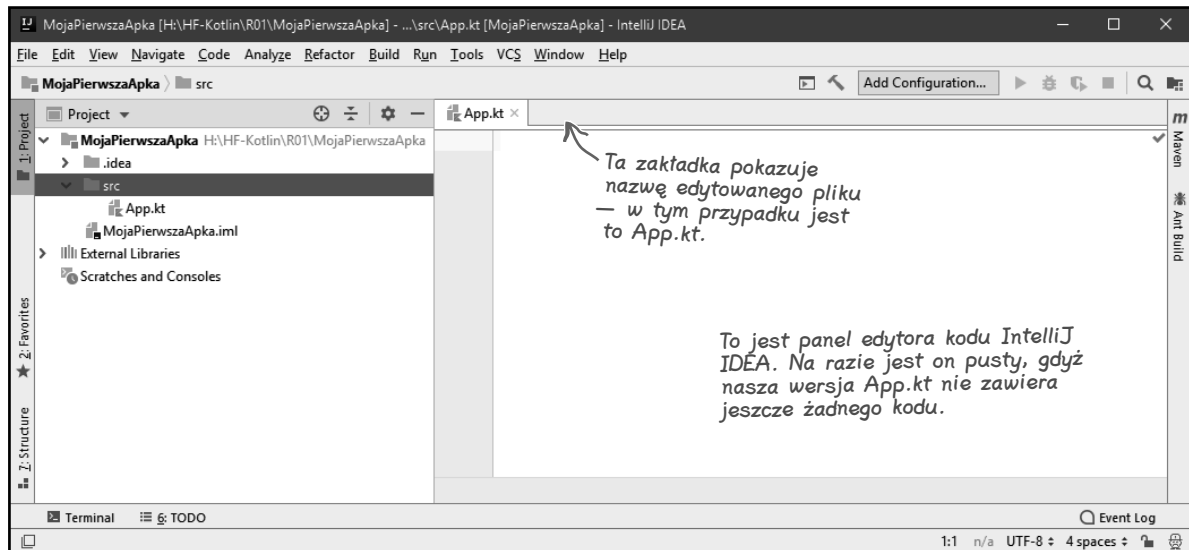
```
fun main() {
    // tu umieść swój kod
}
```

W dalszej części tej książki zazwyczaj będziemy używać tej dłuższej wersji funkcji `main`, gdyż będzie ona działać we wszystkich wersjach Kotlin.

Dodaj funkcję main do pliku App.kt

Aby dodać funkcję `main` do projektu, otwórz plik `App.kt`, dwukrotnie klikając go w eksploratorze IntelliJ IDEA. Spowoduje to otwarcie pliku w edytorze kodu, jak pokazaliśmy poniżej:

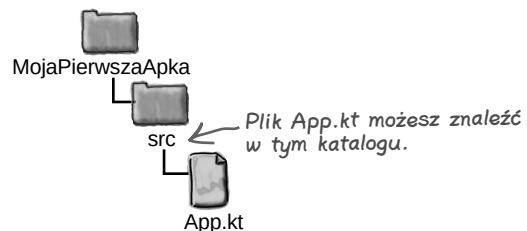
- ☒ Zbudowanie aplikacji
- ☐ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL



Następnie zaktualizuj swoją wersję pliku `App.kt` tak, by odpowiadała naszej, przedstawionej poniżej:

```
fun main(args: Array<String>) {
    println("Hau!")
}
```

Spróbuj teraz wykonać program.



Nie istnieją
grupy pytania

P: Czy funkcję `main` muszę dodawać do każdego tworzonego pliku Kotliny?

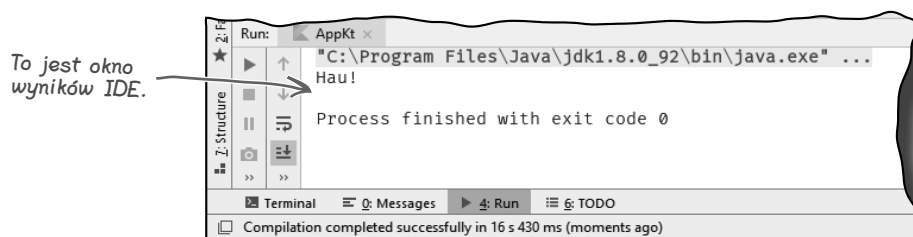
O: Nie. Aplikacje pisane w Kotlinie mogą składać się z dziesiątek, a nawet setek plików, jednak tylko w jednym z nich może się znajdować funkcja `main` — w tym, który ma rozpoczynać wykonywanie aplikacji.



Jazda próbna

Aby wykonać kod w IntelliJ IDEA, należy wybrać z menu głównego opcję *Run*, a następnie polecenie *Run*. Kiedy zostaniesz o to poproszony, wybierz opcję *AppKt*. To spowoduje zbudowanie projektu i wykonanie kodu.

Po krótkim oczekiwaniu w oknie wyników u dołu głównego okna IntelliJ IDEA zostanie wyświetlony tekst „Hau!”:



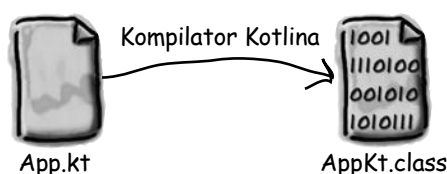
- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

Co robi polecenie Run?

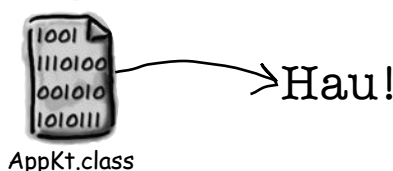
Po wykonaniu polecenia *Run*, zanim zostaną wyświetlone wyniki, IntelliJ IDEA wykonuje kilka czynności:

- 1 IDE kompiluje kod napisany w Kotlinie do postaci kodów bajtowych Javy. Zakładając, że w naszym kodzie nie będzie żadnych błędów, jego skompilowanie spowoduje utworzenie jednego lub wielu plików klasowych, które będzie można wykonać w JVM. W naszym przypadku skompilowanie pliku *App.kt* spowoduje utworzenie pliku klasowego *AppKt.class*.

Mówiąc bardziej konkretnie, IDE kompiluje nasz kod źródłowy do kodów bajtowych JVM, gdyż podczas tworzenia projektu zaznaczyliśmy opcję JVM. Gdybyśmy wybrali uruchamianie projektu w innym środowisku, to kompilator skompilowałby plik źródłowy do kodów dostosowanych do wskazanego środowiska.



- 2 IDE uruchamia JVM, a następnie wykonuje klasę *AppKt.class*. JVM przekształca plik *AppKt.class* w coś, co będzie w stanie zrozumieć używana platforma systemowa, a następnie to coś wykonuje. W efekcie w oknie wyników IDE zostanie wyświetlony napis „Hau!”.



Skoro już wiemy, że nasza funkcja działa, zobaczmy, jak sprawić, by mogła robić coś więcej.

Co możemy nakazać w funkcji main?

Kiedy już znajdziemy się wewnątrz funkcji `main` (czy też jakiegokolwiek innej), zaczyna się prawdziwa zabawa. Aby nasz program coś zrobił, możemy w niej nakazać wykonanie wszystkich działań, jakie znamy z większości innych języków programowania.

Oto co nasz kod może:

★ Coś zrobić (instrukcje)

```
var x = 3
val name = "Onufry"
x = x * 10
print("x ma wartość $x.")
// To jest komentarz
```

★ Zrobić coś znowu i znowu, i znowu (pętle)

```
while (x > 20) {
    x = x - 1
    print("Teraz x ma wartość $x.")
}
for (i in 1..10) {
    x = x + 1
    print("Teraz x ma wartość $x.")
}
```

★ Zrobić coś pod warunkiem (rozgałęzienia)

```
if (x == 20) {
    println("x musi wynosić 20.")
} else {
    println("x jest różne od 20.")
}
if (name.equals("Onufry")) {
    println("$name Mościpański")
}
```

Tym fragmentom przyjrzymy się nieco dokładniej na kilku następnych stronach książki.

- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

Składnia pod lupą



Poniżej zamieściliśmy kilka ogólnych wskazówek i rad dotyczących składni dla osób po raz pierwszy poruszających się po Kotlinie:

★ Komentarz zajmujący jeden wiersz rozpoczyna się od sekwencji dwóch znaków ukośnika:

// To jest komentarz

★ W większości przypadków odstęp nie mają znaczenia:

x = 3

★ Aby zdefiniować zmienną, należy użyć słowa kluczowego `var` lub `val`, a następnie podać nazwę zmiennej. Słowa `var` używamy do tworzenia zmiennych, których wartości chcemy zmieniać, a `val` — do tworzenia zmiennych, których wartość ma pozostać taka sama. Więcej o zmiennych dowiesz się w rozdziale 2.

```
var x = 100
val serialNo = "AS498HG"
```

W kółko i w kółko, i w kółko...

Kotlin udostępnia trzy standardowe instrukcje pętli: `while`, `do-while` oraz `for`. Na razie skoncentrujemy się na pętli `while`.

Jej składnia jest stosunkowo prosta. Tak długo, jak jakiś warunek jest spełniony, należy wykonywać wszystko, co jest umieszczone w *bloku* pętli. Blok pętli jest wyznaczany przez parę nawiasów klamrowych, a w nim musi się znaleźć to, co chcemy powtarzać.

Kluczowe znaczenie dla prawidłowego działania pętli ma jej *test warunkowy*, nazywany także po prostu *warunkiem*. Warunek pętli to wyrażenie, które zwraca wartość logiczną — coś, co jest *prawdą* (`true`) lub *falszem* (`false`). Na przykład jeśli powiemy coś takiego: „Tak długo, jak *są* Lody W Wafelku, liżemy”, będzie to zrozumiała test warunkowy. Lody w wafelku mogą być lub może ich w nim już nie być. Jeśli jednak powiemy: „Tak długo jak *Ferde*k, oglądamy”, to tak naprawdę nie jest to żaden test. Musielibyśmy go zmienić na coś w rodzaju: „Tak długo jak *Ferde*k *leci* w *TV*, oglądamy”, aby stwierdzenie miało sens.

- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

← Jeśli w bloku pętli znajduje się tylko jeden wiersz kodu, to nawiasy klamrowe można pominąć.

Proste testy logiczne

Prosty test logiczny można wykonać, sprawdzając zmienną przy użyciu operatora porównania. Oto operatory porównań:

`<` (mniejszy od)

`>` (większy od)

`==` (równy)

← Do sprawdzania równości używamy dwóch znaków równości, a nie jednego.

`<=` (mniejszy lub równy)

`>=` (większy lub równy)

Koniecznie musisz zwrócić uwagę na różnicę pomiędzy operatorem przypisania (pojedynczym znakiem równości) a operatorem równości (dwoma znakami równości).

Oto przykładowy kod, który używa testów logicznych:

```
var x = 4 // Przypisujemy x wartość 4.
while (x > 3) {
    // Kod w pętli będzie wykonywany tak długo, jak x jest większe od 3.
    println(x)
    x = x - 1
}
var z = 27
while (z == 10) {
    // Ta pętla nie zostanie wykonana, gdyż z jest równe 27.
    println(z)
    z = z + 6
}
```

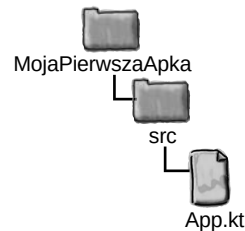
Przykład pętli

Zaktualizujemy kod w pliku *App.kt*, umieszczając w nim nową wersję funkcji *main*. Funkcję tę zmodyfikujemy w taki sposób, by wyświetlała komunikat przed uruchomieniem pętli, podczas każdej iteracji pętli oraz po jej zakończeniu.

Zaktualizuj zatem swoją wersję pliku *App.kt* tak, by odpowiadał naszej, przedstawionej poniżej (zmiany wyróżniliśmy pogrubioną czcionką).

```
fun main(args: Array<String>) {
    println("Ha!") ← Usuń ten wiersz — nie jest już nam potrzebny.
    var x = 1
    println("Przed pętlą. x = $x.")
    while (x < 4) {
        println("Wewnątrz pętli. x = $x.")
        x = x + 1
    }
    println("Za pętlą. x = $x.")
}
```

↑
Ten zapis wyświetla wartość zmiennej x.



- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL



Jazda próbna

Wykonaj kod, wybierając z menu głównego opcję *Run/Run 'AppKt'*. W oknie wyników u dołu okna IntelliJ IDEA zostaną wyświetlone następujące komunikaty:

```
Przed pętlą. x = 1.
Wewnątrz pętli. x = 1.
Wewnątrz pętli. x = 2.
Wewnątrz pętli. x = 3.
Za pętlą. x = 4.
```

Skoro już wiesz, jak działa pętla *while* i testy warunkowe, czas zająć się instrukcją *if*.

print kontra println

Pewnie zauważyłeś, że czasami używamy funkcji **print**, a czasami **println**. Na czym polega różnica pomiędzy nimi?



Otóż **println** wstawia *nowy* wiersz (pomyśl, że jej nazwa to *print line* — wyświetli nowy wiersz), natomiast **print** wyświetla *wszystko w jednym wierszu*. Jeśli chcesz, by każdy wyświetlany tekst znalazł się w odrębnym wierszu, to używaj *println*. Jeżeli chcesz wyświetlać kolejne teksty w jednym wierszu — używaj *print*.

Rozgałęzienia warunkowe

Test `if` jest podobny do testu logicznego używanego w pętli `while`, z tym że zamiast rozumieć go jako „*Tak długo jak* wciąż są lody...”, należy go rozumieć jako „*Jeśli* są lody...”.

Abyś mógł przekonać się, jak to działa, poniżej zamieściliśmy kod, który wyświetla łańcuch, jeśli jedna liczba jest większa od drugiej:

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x jest większe od y")
    }
    println("Ten wiersz zostanie wykonany niezależnie od wszystkiego...")
}
```

Jeśli w bloku `if` znajduje się jeden wiersz, to nawiasy klamrowe można pominąć.

Ten wiersz zostanie wykonany wyłącznie, jeśli `x` będzie większe od `y`.

Powyższy kod wykona wiersz wyświetlający komunikat „x jest większe od y” tylko wtedy, gdy warunek (`x` jest większe od `y`) będzie spełniony. Jednak ostatni wiersz, wyświetlający komunikat „Ten wiersz zostanie wykonany niezależnie od wszystkiego...”, zostanie wykonany bez względu na wartość tego warunku. Dlatego też zależnie od wartości zmiennych `x` i `y` zostanie wyświetlony jeden komunikat lub dwa.

Do warunku można także dodać klauzulę `else`, dzięki której można powiedzieć coś podobnego do: „*Jeśli* wciąż są lody w wafelku, to je zjedz, *a w przeciwnym razie* kup sobie następne”.

Poniżej przedstawiliśmy zaktualizowaną wersję kodu, w którym zastosowaliśmy klauzulę `else`:

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x jest większe od y")
    } else {
        println("x nie jest większe od y")
    }
    println("Ten wiersz zostanie wykonany niezależnie od wszystkiego...")
}
```

Ten wiersz kodu jest wykonywany, jeśli warunek `x > y` nie zostanie spełniony.

W większości innych języków programowania to by było mniej więcej wszystko, co można powiedzieć o instrukcji `if` — używamy jej do wykonania kodu, jeśli zostanie spełniony jakiś warunek. Jednak Kotlin idzie o krok dalej.

- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

Używanie if do zwracania wartości

W Kotlinie if można używać jako **wyrażenia**, czyli do zwracania wartości. To tak, jakby powiedzieć: „*Jeśli* w wafelku wciąż są lody, to zwróć jedną wartość, a w przeciwnym wypadku zwróć inną wartość”. Tej formy if można używać do pisania kodu, który będzie bardziej zwarty.

Aby zobaczyć, jak działa forma if, zmodyfikujemy kod przedstawiony na poprzedniej stronie. Wcześniej do wyświetlania komunikatu używaliśmy następującego fragmentu kodu:

```
if (x > y) {  
    println("x jest większe od y")  
} else {  
    println("x nie jest większe od y")  
}
```

Ten sam efekt co przy korzystaniu z wyrażeń if możemy uzyskać, używając kodu o następującej postaci:

```
println(if (x > y) "x jest większe od y" else "x nie jest większe od y")
```

Fragment kodu:

```
if (x > y) "x jest większe od y" else "x nie jest większe od y"
```

jest właśnie wyrażeniem if. W pierwszej kolejności sprawdza on podany warunek, czyli: $x > y$. Jeśli warunek ten zostanie *spełniony*, to wyrażenie zwróci łańcuch "x jest większe od y". W przeciwnym razie, czyli jeśli warunek *nie zostanie spełniony* (co odpowiada klauzuli else), wyrażenie zwróci łańcuch "x nie jest większe od y".

Po określeniu wartości wyrażenia if zostaje ona wyświetlona przez funkcję println:

```
println(if (x > y) "x jest większe od y" else "x nie jest większe od y")
```

A zatem jeśli x jest większe od y, to zostanie wyświetlony komunikat "x jest większe od y". Jeśli jednak x nie jest większe od y, to kod wyświetli komunikat "x nie jest większe od y".

Jak widzisz, taki sposób stosowania wyrażeń if daje dokładnie takie same efekty jak użycie kodu przedstawionego na poprzedniej stronie, jednak pozwala na pisanie kodu, który jest bardziej zwężły.

Na następnej stronie przedstawimy kompletny kod naszej funkcji main.

- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

W przypadku stosowania if jako wyrażenia, **MUSISZ** użyć także klauzuli else.

↖
Jeśli x jest większe od y, to kod wyświetli komunikat „x jest większe od y”. Jeżeli natomiast x nie jest większe od y, to kod wyświetli komunikat „x nie jest większe od y”.

Aktualizujemy funkcję main

Zaktualizujmy kod pliku *App.kt*, zapisując w nim nową wersję funkcji `main` korzystającą z wyrażenia `if`. Zastąp kod w swoim pliku tak, by był identyczny jak ten przedstawiony poniżej:

```
fun main(args: Array<String>) {
    var x = 1
    println("Przed pętlą. x = $x.")
    while (x < 4) {
            println("Wewnątrz pętli. x = $x.")
            x = x + 1
    }
    println("Za pętlą. x = $x.")
    val x = 3
    val y = 1
    println(if (x > y) "x jest większe od y" else "x nie jest większe od y")
    println("Ten wiersz zostanie wykonany niezależnie od wszystkiego...")
}
```

Usuń te wiersze kodu.

Weźmy teraz ten kod na jazdę próbną.



Jazda próbna

Uruchom kod, wybierając z menu głównego opcję *Run/Run 'AppKt'*.

W oknie wyników u dołu IDE powinny zostać wyświetlone następujące teksty:

```
x jest większe od y
Ten wiersz zostanie wykonany niezależnie od wszystkiego...
```

Teraz, kiedy już wiesz, jak tworzyć rozgałęzienia warunkowe przy użyciu instrukcji `if` oraz wyrażenia `if`, mamy dla Ciebie pewne zadanie.

- ☒ Zbudowanie aplikacji
- ☒ Dodanie funkcji
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

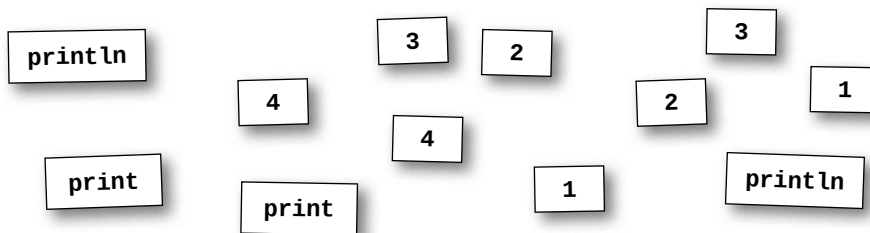


Magnesiki z kodem

Ktoś użył magnesików, by ułożyć na lodówce nową, przydatną wersję funkcji **main**, która wyświetla tekst „YabbaDabbaDo”. Niestety niewyjaśnione kuchenne tornado porzuciło magnesiki. Czy potrafisz umieścić je z powrotem w odpowiednich miejscach kodu?

Nie będziesz potrzebował wszystkich magnesików.

```
fun main(args: Array<String>) {  
    var x = 1  
  
    while (x < ..... ) {  
        ..... (if (x == ..... ) "Yab" else "Dab")  
        ..... ("ba")  
        x = x + 1  
    }  
    if (x == ..... ) println("Do")  
}
```



—————> Odpowiedź znajdziesz na stronie 29.

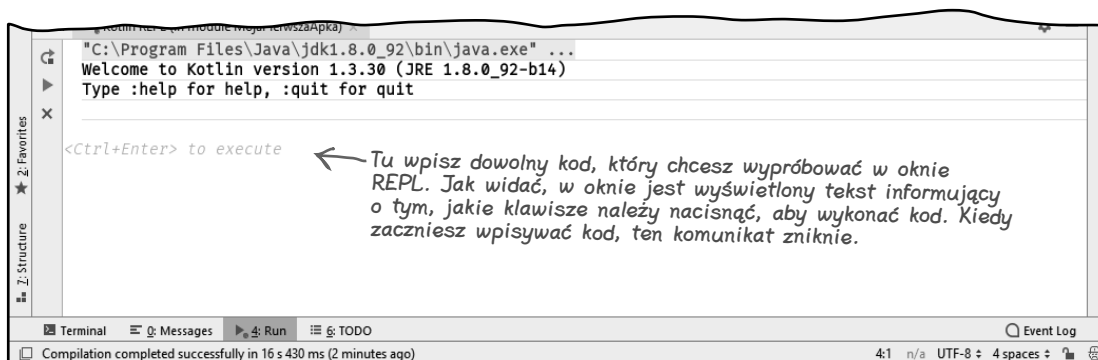
Stosowanie interaktywnej powłoki Kotlina

Zbliżamy się już do końca tego rozdziału, jednak zanim go definitywnie skończymy, jest jeszcze jedna rzecz, którą chcieliśmy Ci przedstawić — jest nią interaktywna powłoka Kotlina, czyli REPL. REPL pozwala na szybkie wypróbowanie działania fragmentu kodu poza właściwym kodem pisanego programu.

Narzędzie REPL można otworzyć, wyświetlając w menu głównym opcję *Tools*, a następnie wybierając opcję *Kotlin/Kotlin REPL*. Po jej wybraniu w dolnej części IntelliJ IDEA pojawi się okno przedstawione poniżej.

- ✓ Zbudowanie aplikacji
- ✓ Dodanie funkcji
- ✓ Aktualizacja funkcji
- ☐ Zastosowanie REPL

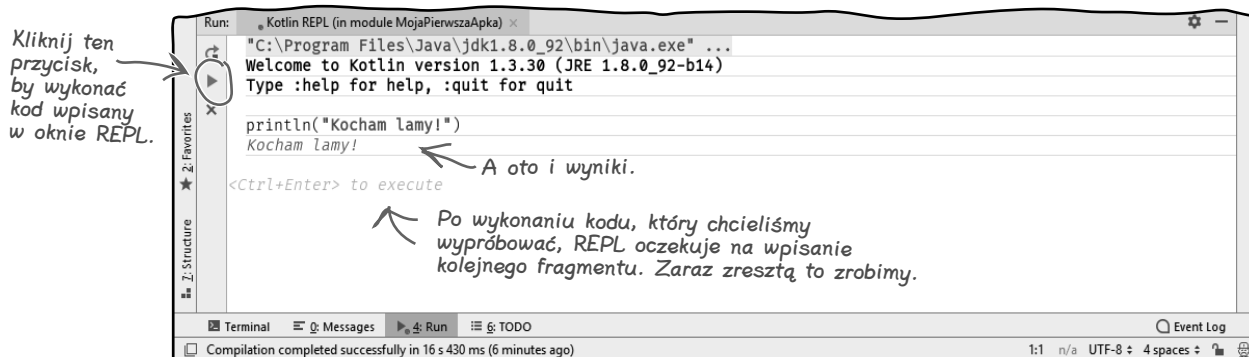
REPL to skrót angielskich słów: *Read-Eval-Print-Loop* (przeczytaj-przetwórz-wyświetl-powtórz), ale tak naprawdę nikt go w ten sposób nie nazywa.



Aby użyć narzędzia REPL, wystarczy wpisać kod w jego oknie. W ramach testu wpisz poniższy wiersz kodu:

```
println("Kocham lamy!")
```

Po wpisaniu kodu można go wykonać, klikając przycisk *Run* wyświetlony z lewej strony okna REPL. Po krótkiej chwili w oknie powinien zostać wyświetlony tekst „Kocham lamy!”, tak jak na kolejnym rysunku.



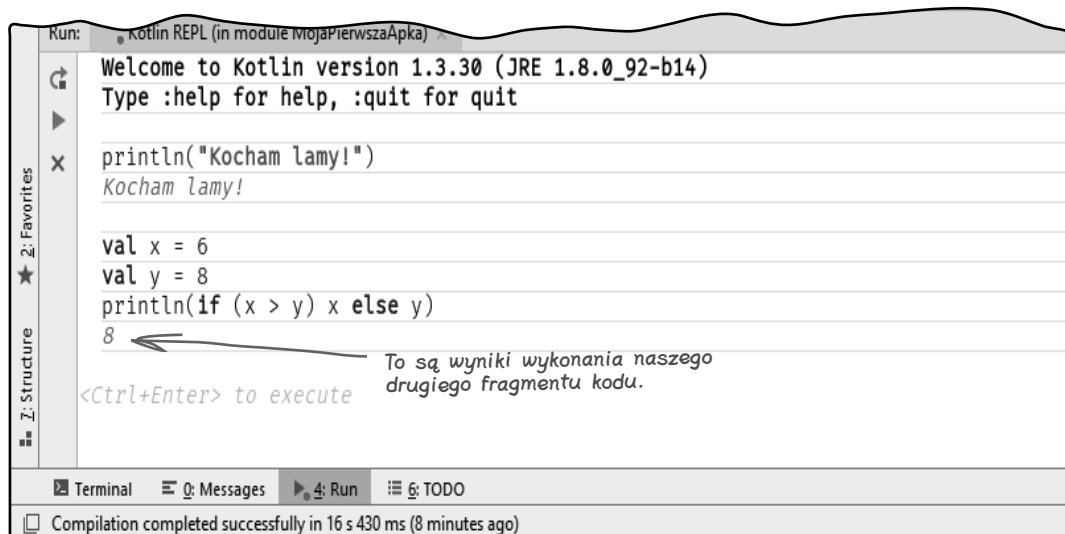
W REPL można wpisywać fragmenty mające wiele wierszy kodu

Oprócz wpisywania w REPL pojedynczych wierszy kodu, jak zrobiliśmy na poprzedniej stronie, można w nim także sprawdzać dłuższe fragmenty, zawierające wiele wierszy. W ramach przykładu spróbujmy dodać do okna REPL poniższy fragment:

```
val x = 6
val y = 8
println(if (x > y) x else y)
```

← Ta instrukcja wyświetla większą z dwóch liczb.

Po wykonaniu tego kodu w oknie REPL powinien zostać wyświetlony wynik przedstawiony na poniższym rysunku:



Czas na ćwiczenia

Teraz, kiedy już nauczyłeś się, jak pisać kod w języku Kotlin, oraz poznałeś podstawy jego składni, mamy dla Ciebie kilka ćwiczeń. Pamiętaj, że jeśli nie jesteś czegoś pewien, możesz sprawdzić działanie fragmentu kodu w oknie REPL.

- ✓ Zbudowanie aplikacji
- ✓ Dodanie funkcji
- ✓ Aktualizacja funkcji
- ✓ Zastosowanie REPL

Wykonaliśmy już wszystkie czynności przewidziane w tym rozdziale.



Bądź kompilatorem

Każdy z plików Kotlina przedstawionych poniżej reprezentuje jeden, kompletny plik źródłowy. Twoim zadaniem jest wcielić się w rolę kompilatora i dla każdego z tych plików określić, czy uda się go skompilować, czy nie. Jeśli uważasz, że pliku nie da się skompilować, to jak byś go poprawił?

- ✓ Zbudowanie aplikacji
- ✓ Dodanie funkcji
- ✓ Aktualizacja funkcji
- ✓ Zastosowanie REPL

A

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        if (x > 3) {
            println("duże x")
        }
    }
}
```

B

```
fun main(args: Array<String>) {
    val x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("małe x")
    }
}
```

C

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "małe x")
    }
}
```



Bądź kompilatorem. Rozwiązanie

Każdy z plików Kotlina przedstawionych poniżej reprezentuje jeden, kompletny plik źródłowy. Twoim zadaniem jest wcielić się w rolę kompilatora i dla każdego z tych plików określić, czy uda się go skompilować, czy nie. Jeśli uważasz, że pliku nie da się skompilować, to jak byś go poprawił?

A

```
fun main(args: Array<String>) {  
    var x = 1  
    while (x < 10) {  
        x = x + 1  
        if (x > 3) {  
            println("duże x")  
        }  
    }  
}
```

Ten fragment można skompilować, ale bez dodania jednego wiersza kodu po uruchomieniu nie wyświetli on żadnych wyników — wpadnie w nieskończoną pętlę „while”.

B

```
fun main(args: Array<String>) {  
    var var x = 10  
    while (x > 1) {  
        x = x - 1  
        if (x < 3) println("małe x")  
    }  
}
```

Tego fragmentu nie można skompilować. Zmienna `x` została zdefiniowana z użyciem `val`, a to oznacza, że jej wartości nie można zmieniać. Fragment kodu nie może zatem aktualizować wartości zmiennej `x` wewnątrz pętli „while”. Aby rozwiązać ten problem, wystarczy zmienić `val` na `var`.

C

```
fun main(args: Array<String>) {  
    var x = 10  
    while (x > 1) {  
        x = x - 1  
        print(if (x < 3) "małe x" else "duże x")  
    }  
}
```

Tego fragmentu nie można skompilować, gdyż użyto w nim wyrażenia `if` bez klauzuli `else`. Aby rozwiązać ten problem, wystarczy dopisać klauzulę `else`.



Wymieszane komunikaty

Poniżej przedstawiliśmy krótki program w Kotlinie. Brakuje w nim jednego bloku kodu. Twoim zadaniem jest dopasowanie zaproponowanych (po lewej) bloków kodu z wynikami, które zostałyby wyświetlone, gdyby fragment ten został umieszczony w zaznaczonym miejscu programu. Nie wszystkie wiersze wyników są potrzebne, a niektóre z nich mogą zostać użyte więcej niż jeden raz. Narysuj linie łączące bloki kodu z generowanymi przez nie wynikami.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        
        print("$x$y ")
        x = x + 1
    }
}
```

← Proponowany blok kodu

Proponowane bloki kodu

y = x - y

y = y + x

y = y + 3
if (y > 4) y = y - 1

x = x + 2
y = y + x

if (y < 5) {
 x = x + 1
 if (y < 3) x = x - 1
}
y = y + 3

Możliwe wyniki

00 11 23 36 410

00 11 22 33 44

00 11 21 32 42

03 15 27 39 411

22 57

02 14 25 36 47

03 26 39 412

Dopasuj każdy z proponowanych bloków kodu z możliwymi wynikami.

Wymieszane komunikaty. Rozwiązanie



Wymieszane komunikaty. Rozwiązanie

Poniżej przedstawiliśmy krótki program w Kotlinie. Brakuje w nim jednego bloku kodu. Twoim zadaniem jest dopasowanie zaproponowanych (po lewej) bloków kodu z wynikami, które zostałyby wyświetlone, gdyby fragment ten został umieszczony w zaznaczonym miejscu programu. Nie wszystkie wiersze wyników są potrzebne, a niektóre z nich mogą zostać użyte więcej niż jeden raz. Narysuj linie łączące bloki kodu z generowanymi przez nie wynikami.

```
fun main(args: Array<String>) {  
    var x = 0  
    var y = 0  
    while (x < 5) {  
          
    }  
    print("$x$y ")  
    x = x + 1  
}
```

Proponowane bloki kodu

y = x - y

y = y + x

y = y + 3
if (y > 4) y = y - 1

x = x + 2
y = y + x

if (y < 5) {
 x = x + 1
 if (y < 3) x = x - 1
}
y = y + 3

Możliwe wyniki

00 11 23 36 410

00 11 22 33 44

00 11 21 32 42

03 15 27 39 411

22 57

02 14 25 36 47

03 26 39 412



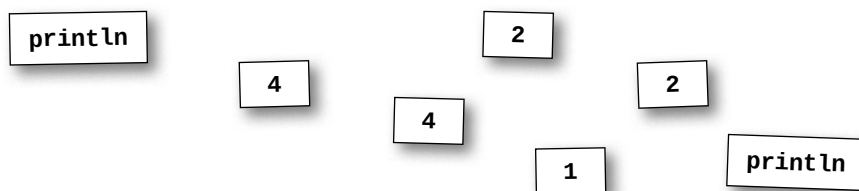
Magnesiki z kodem. Rozwiązanie

Ktoś użył magnesików, by ułożyć na lodówce nową, przydatną wersję funkcji **main**, która wyświetla tekst „YabbaDabbaDo”. Niestety niewyjaśnione kuchenne tornado porzuciło magnesiki. Czy potrafisz umieścić je z powrotem w odpowiednich miejscach kodu?

Nie będziesz potrzebował wszystkich magnesików.

```
fun main(args: Array<String>) {
    var x = 1

    while (x < .. 3 ..) {
        print ... (if (x == ... 1 ... ) "Yab" else "Dab")
        .. print .. ("ba")
        x = x + 1
    }
    if (x == ... 3 .... ) println("Do")
}
```



Te magnesiki nie były potrzebne.



Twój przybornik do Kotlina

Opanowałeś już materiał przedstawiony w rozdziale 1. i do swojego przybornika dodałeś znajomość podstawowej składni języka Kotlin.

Kompletne kody do rozdziału 1. można pobrać z serwera FTP wydawnictwa Helion:
<ftp://ftp.helion.pl/przyklady/kotrug.zip>.



CELNE SPOSTRZEŻENIA

- Do definiowania funkcji służy słowo kluczowe `fun`.
- Każda aplikacja musi zawierać funkcję o nazwie `main`.
- Komentarze zajmujące jeden wiersz rozpoczynają się od sekwencji znaków `//`.
- `String` to łańcuch znaków. Wartość takiego łańcucha zapisujemy pomiędzy dwoma znakami cudzysłowu.
- Bloki kodu są zapisywane pomiędzy nawiasami klamrowymi — `{ }`.
- Operator przypisania ma postać *jednego znaku równości* — `=`.
- Operator równości ma postać *dwóch znaków równości* — `==`.
- Do definiowania zmiennych, których wartość może się zmieniać, należy używać słowa kluczowego `var`.
- Do definiowania zmiennych, których wartość ma pozostawać taka sama, należy używać słowa kluczowego `val`.
- Pętla `while` wykonuje wszystko, co jest umieszczone w jej bloku kodu, tak długo, jak podany test warunkowy będzie *spełniony*.
- Jeśli test warunkowy *nie będzie spełniony*, blok kodu pętli `while` nie zostanie wykonany, a realizacja programu zostanie przeniesiona do instrukcji umieszczonej bezpośrednio za pętlą.
- Test warunkowy należy umieszczać wewnątrz pary nawiasów — `( )`.
- Używając `if else`, można dodawać do kodu rozgałęzienia warunkowe.
- `if` można także używać jako wyrażenia, które zwraca wartość. W takim przypadku zastosowanie klauzuli `else` jest obowiązkowe.

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 

Rusz głową! Kotlin



Kariera Kotliny nabrała rozpędu, gdy w 2017 roku na konferencji Google ogłoszono go jednym z oficjalnie wspieranych języków programowania aplikacji dla Androida. Jest lubiany za zwartą składnię, elastyczność i ścisłą integrację z kodem Javy. Oczywiście odpowiada też kryteriom nowoczesnego i innowacyjnego języka programowania. Jest bardzo dobrym wyborem dla każdego, kto chce się stać profesjonalnym programistą, ale też osoby programujące hobbystycznie będą zadowolone z jego możliwości. Problemów z tym idealnym językiem programowania jest niewiele: trzeba poznać Kotliną, zrozumieć kilka niuansów i nauczyć się kodowania...

Ta książka, podobnie jak inne pozycje z serii *Rusz głową!*, została przygotowana zgodnie z jedyną w swoim rodzaju metodyką nauczania wykorzystującą zasady funkcjonowania ludzkiego mózgu. Dzięki niej nauczysz się myśleć jak najlepsi programiści i niepostrzeżenie zaczniesz tworzyć wydajny kod w Kotlinie. Autorzy zastosowali najlepsze osiągnięcia psychologii, neurologii i innych nauk o uczeniu się, stąd niecodzienny wygląd i struktura książki. W efekcie zamiast klasycznego podręcznika otrzymujesz polisensoryczne doświadczenie poznawcze zaprojektowane tak, aby już wkrótce Kotlin stał się pewnym, niezawodnym narzędziem w Twoich rękach!

W tej książce między innymi:

- solidne podstawy tworzenia kodu i pisanie pierwszych projektów
- typy sparametryzowane w Kotlinie
- praca na obiektach: dziedziczenie, klasy, kolekcje
- funkcje wyższego rzędu i wyrażenia lambda
- współbieżność wykonywania kodu i najciekawsze rozszerzenia

**Kotlin:
oto radość
z pisania kodu!**

Dawn Griffiths — jest znakomitą i bardzo doświadczoną programistką oraz autorką wielu książek z serii *Rusz głową!* Wraz z mężem Davidem opracowała także animowany kurs wideo *The Agile Sketchpad*, stanowiący próbę uczenia kluczowych pojęć i technik w sposób zapewniający aktywną pracę mózgu i utrzymanie zaangażowania.

David Griffiths — jest trenerem agile, programistą. W wieku 15 lat napisał implementację języka Logo. Jest autorem kilku książek z serii *Rusz głową!*

Helion



hellon.pl



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

Sprawdź nasze szkolenia!



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Slegnij po więcej!



ISBN 978-83-283-5869-0



9 788328 358690

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł