O'REILLY®

Helion

Rusz głową!

Kotlin

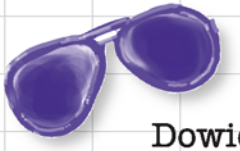
Poradnik programowania w Kotlinie dla tych, którzy chcą się uczyć

Pobuszuj w standardowej bibliotece Kotlina

Uniknij żenujących błędów z wyrażeniami lambda

Odkryj tajniki typów sparametryzowanych

Pisz nieziemskie funkcje wyższego rzędu



Dowiedz się, jak Elvis może zmienić Twoje życie

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

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.

Spis treści (podsumowanie)

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 utrwaliły*. 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 nauczenia się programowania w Kotlinie?

Zaczynamy

1 Szybki skok

Kotlin robi dużo szumu.

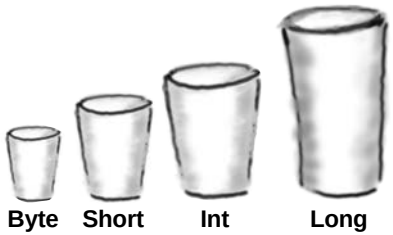
Od momentu pojawienia się pierwszej wersji Kotlina język ten urzekł programistów swoją ***przyjazną składnią***, ***spójnością***, ***elastycznością*** oraz ***możliwościami***. W tej książce nauczymy Cię **pisać w Kotlinie własne aplikacje**, a zaczniemy od pokazania, jak stworzyć i uruchomić prosty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

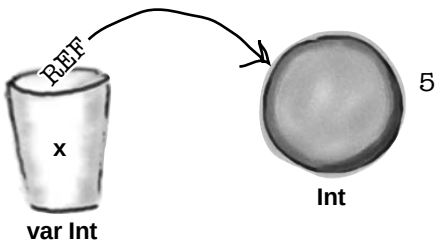
2 Typy proste i zmienne

Być zmienną

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 Kotlina 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 Kotliczynie mają obiekty*.





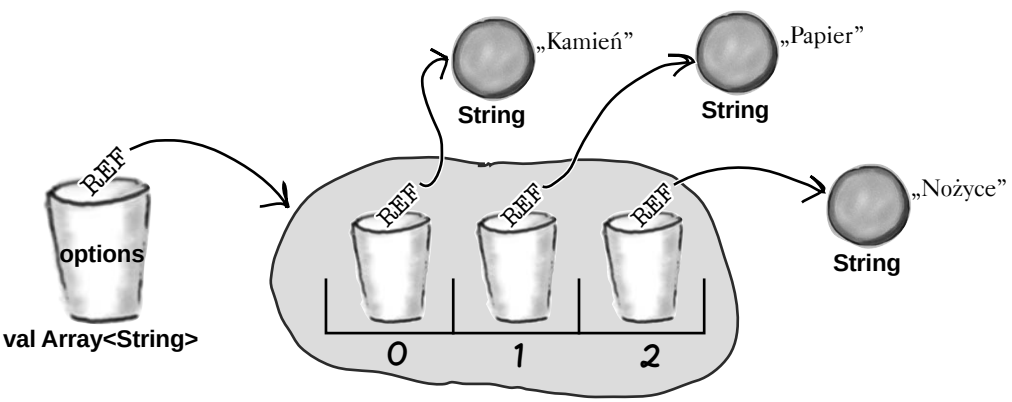
3

Funkcje

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`.



4 Klasy i obiekty

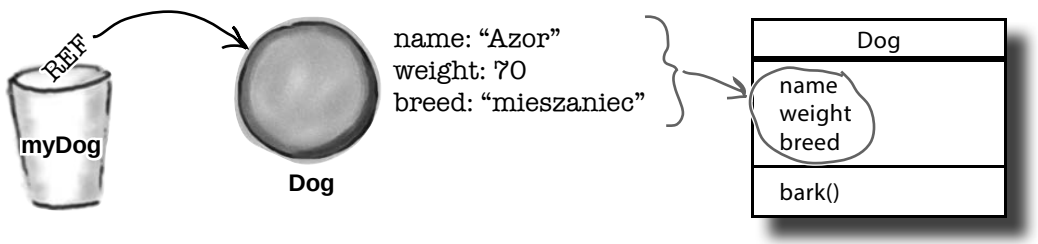
Trochę klasy

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

Wiele obiektów

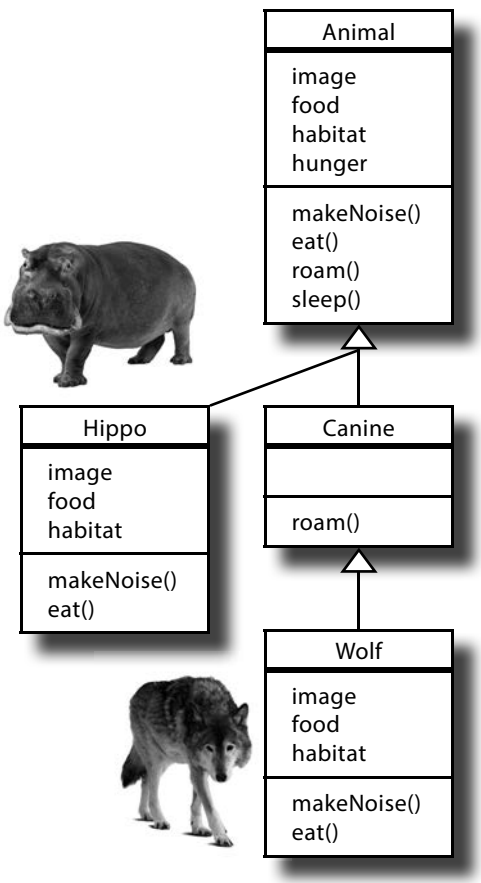


5 Klasy pochodne i bazowe

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.

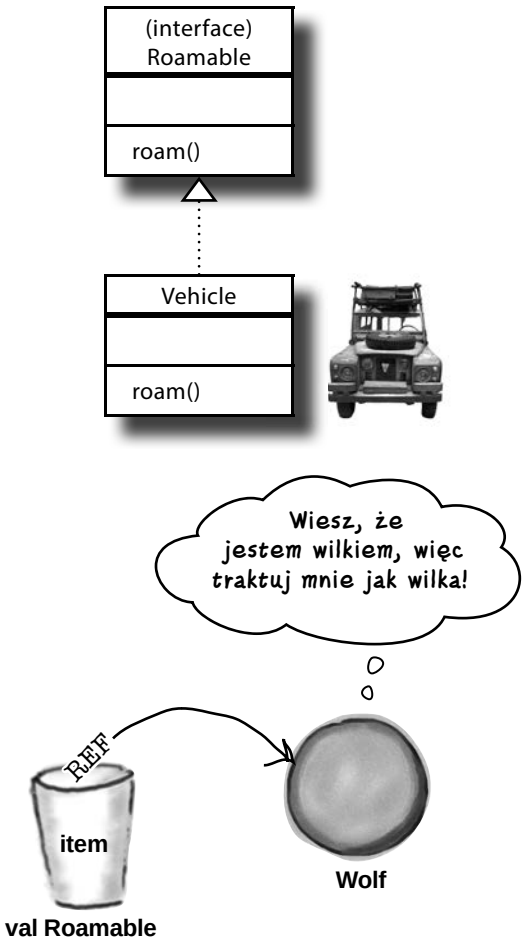


6 Klasy abstrakcyjne i interfejsy

Poważny polimorfizm

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***.

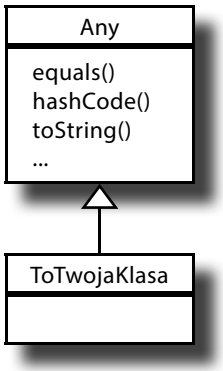


7 Klasy danych

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 ***destrukturyzację 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.

Wartość null i wyjątki

8 Cały i zdrów

Nikt nie chce pisać kodu, który nie będzie bezpieczny.

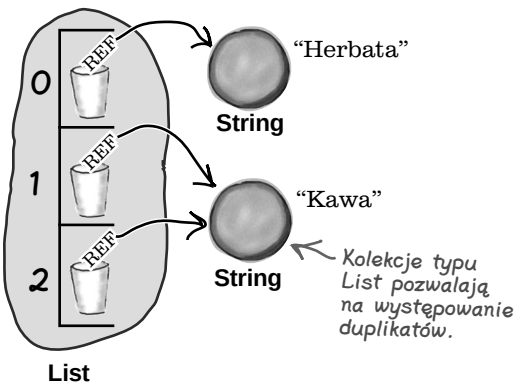
Na szczęście mamy doskonałą wiadomość: Kotlin *został zaprojektowany pod kątem bezpieczeństwa kodu*. Zaczniemy od pokazania Ci, dlaczego stosowanie w Kotlinie **typów akceptujących wartości puste** oznacza, że *podczas swojego pobytu w Kotliczynie prawie nigdy nie będziesz miał do czynienia z wyjątkami 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.

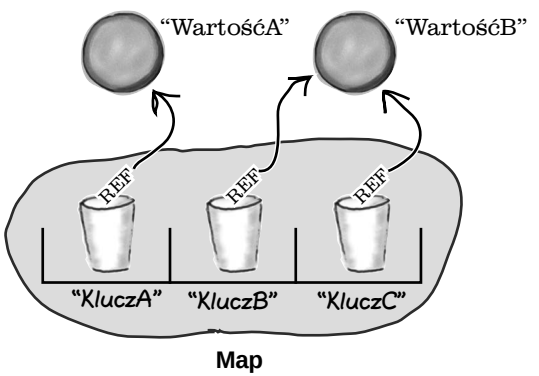
9 Kolekcje

Zorganizuj się!

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...





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

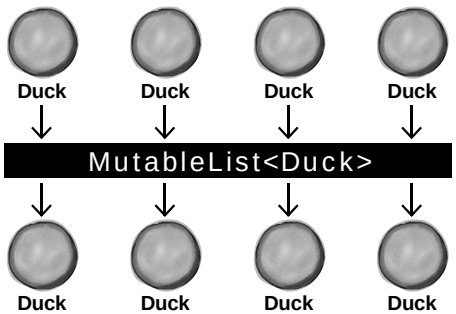
10 Typy sparametryzowane

Odróżniaj wariancję od kontrawariancji

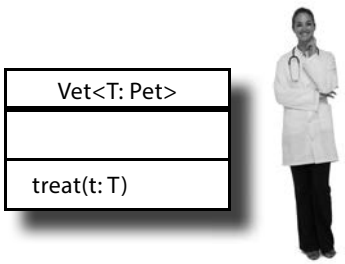
Każdy lubi kod, który jest spójny.

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 **kolekcje 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.

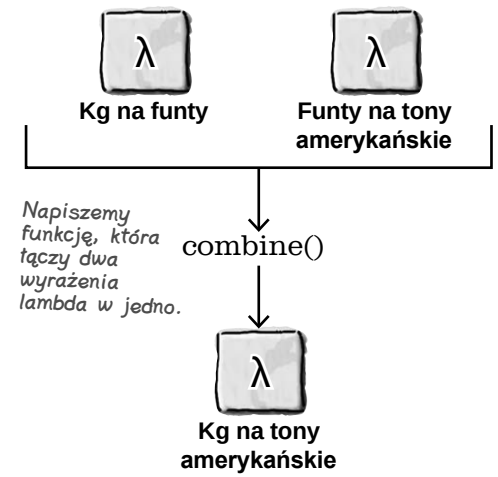


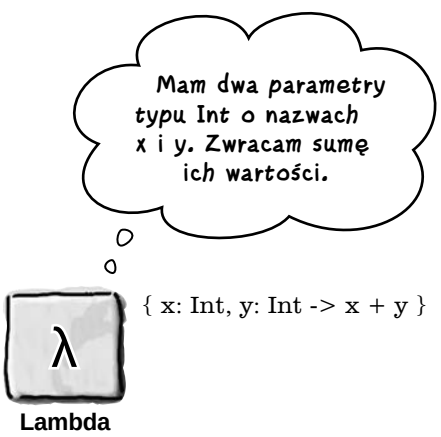
11 Lambdy i funkcje wyższego rzędu

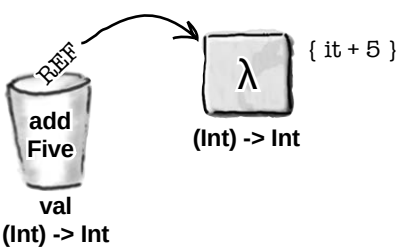
Kod używany jak dane

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***.





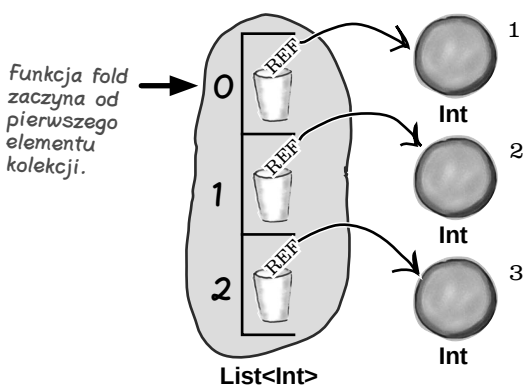


12 Wbudowane funkcje wyższego rzędu

Wzmocnij swój kod

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**.

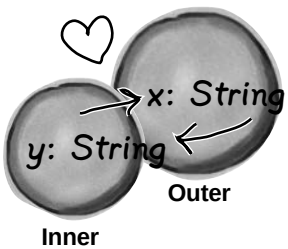


Pozostałości

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.

Szybki skok

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 prostą aplikację. Po drodze przedstawimy podstawy składni Kotlina, 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 światku, 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 Kotlina 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 tropienie i usuwanie 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***.

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.

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 Kotlina 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 JavaScripcie 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.

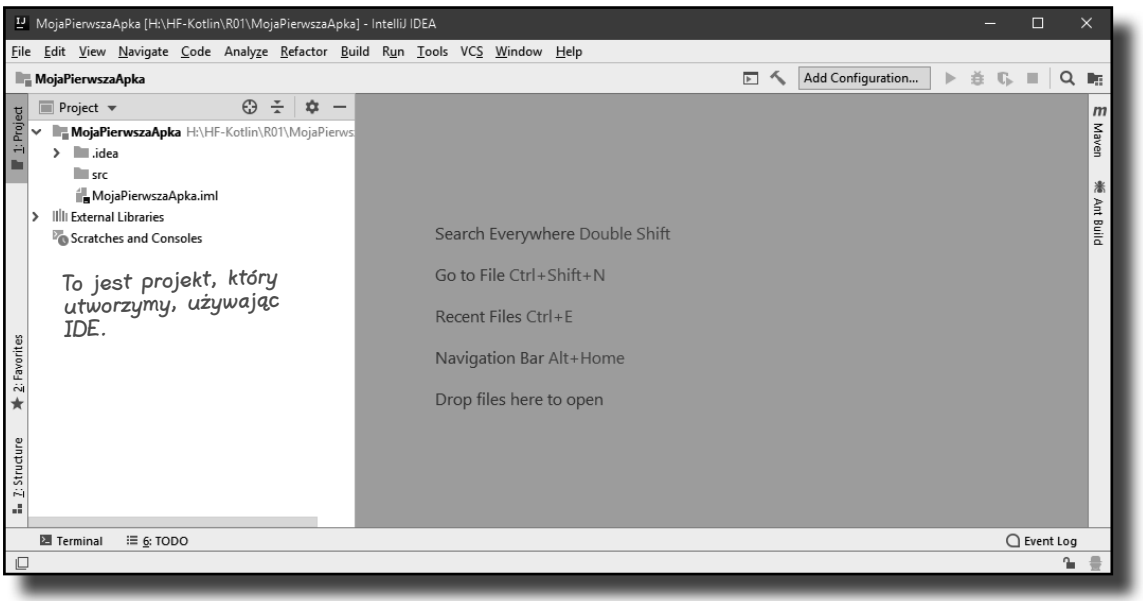
Choć będziemy pisać aplikacje działające w JVM, to by w pełni skorzystać z tej książki, wcale nie musisz znać Javy. Zakładamy co prawda, że dysponujesz pewnym ogólnym doświadczeniem z zakresu programowania, ale to wszystko.

A zatem dajmy nurka na głęboką wodę.

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.**
   Zaczniemy 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 Kotlina 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 Kotlina.**
   I w końcu dowiemy się, w jaki sposób można sprawdzać fragmenty kodu w interaktywnej powłoce Kotlina (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) {             Każe 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) {            Każe 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.
```

Jesteś tutaj

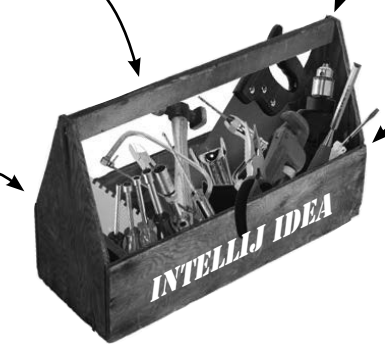
- [x] Zbudowanie aplikacji
- [ ] Dodanie funkcji
- [ ] Aktualizacja funkcji
- [ ] Zastosowanie REPL

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:

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 ← Koniecznie wybierz opcję do pobrania 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.

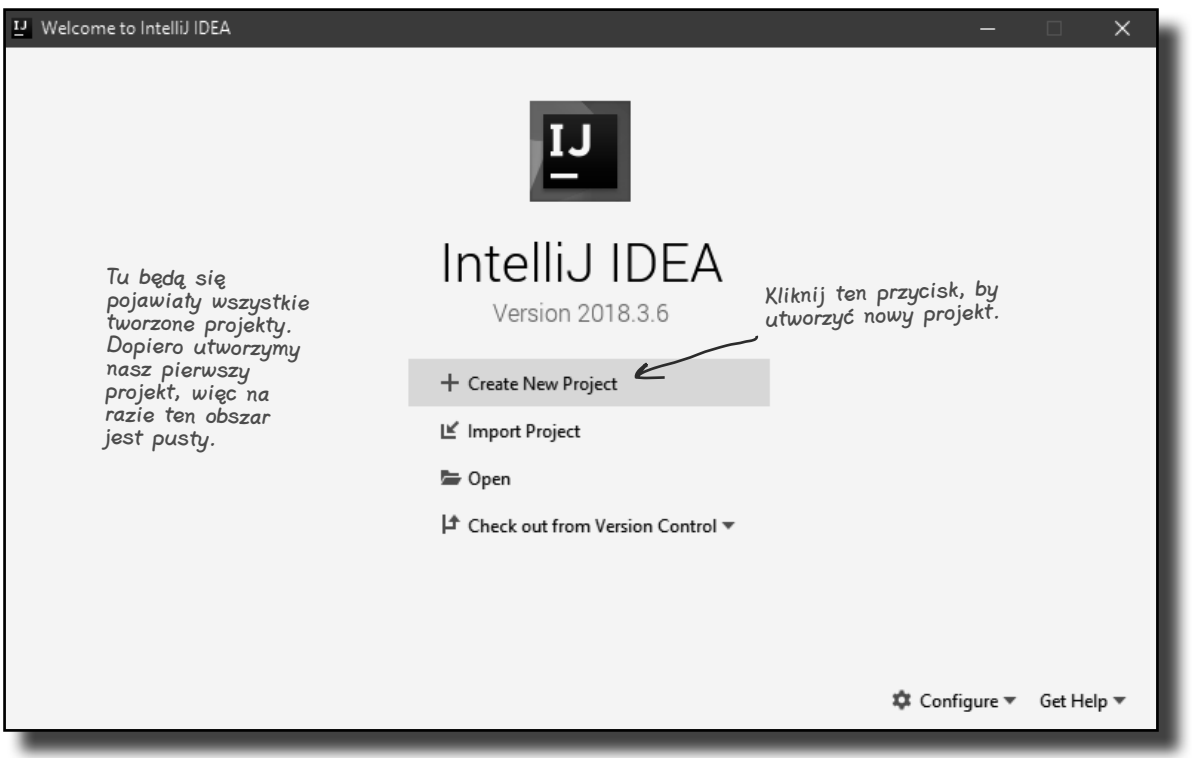
Stwórzmy prostą aplikację

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.

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)

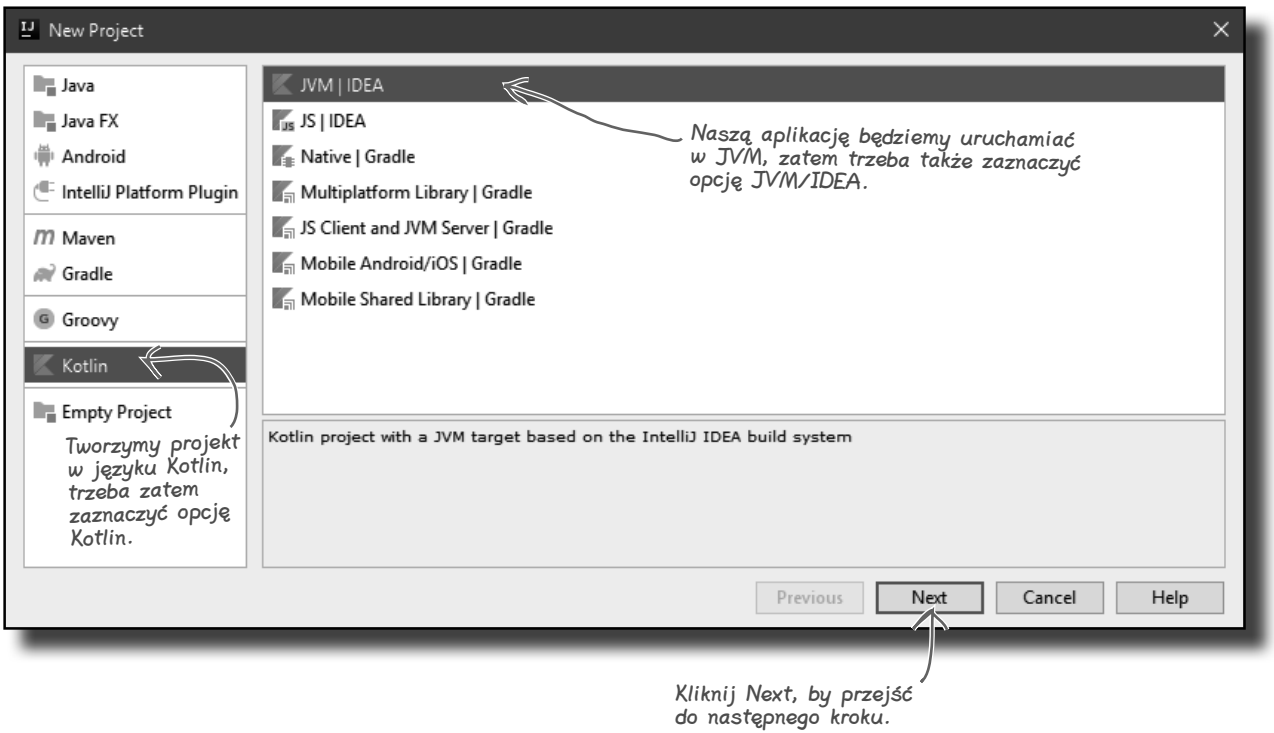
2. Określenie typu projektu

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 w 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).



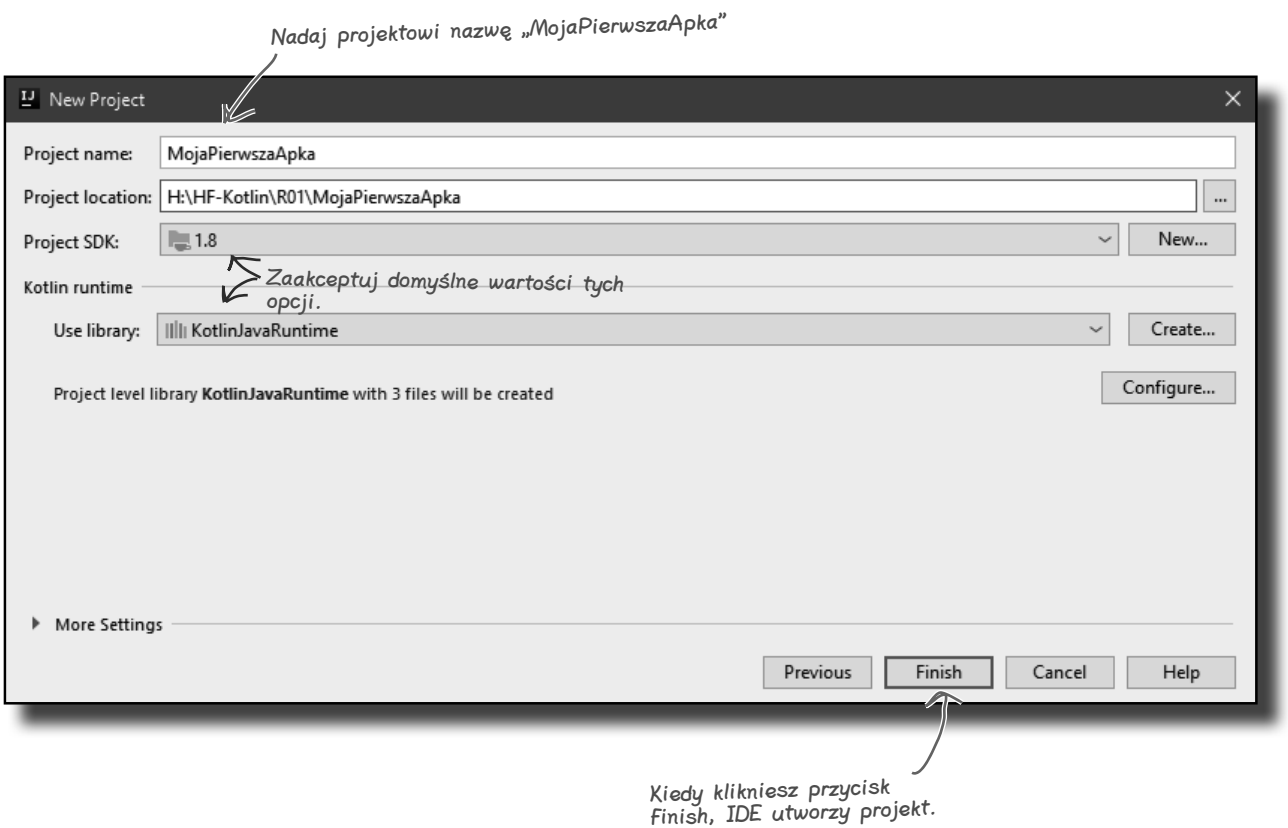
Tworzenie prostej aplikacji (ciąg dalszy)

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.

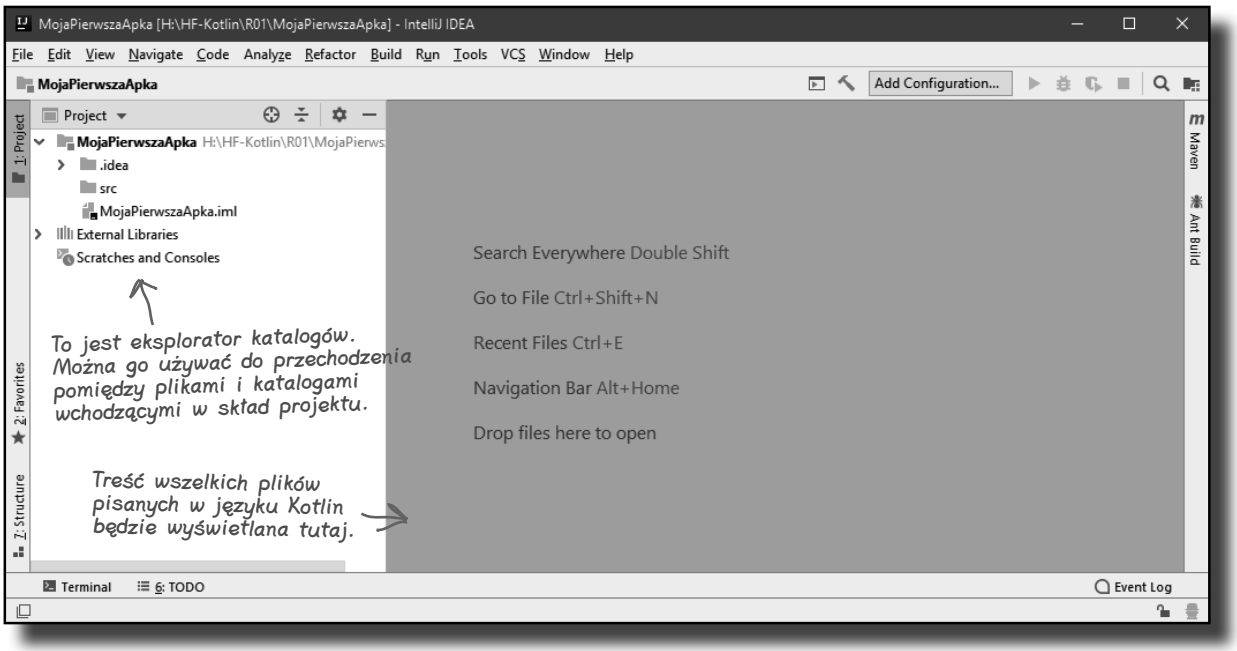


Właśnie utworzyłeś swój pierwszy projekt w Kotlinie

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

- [x] **Zbudowanie aplikacji**
- [ ] **Dodanie funkcji**
- [ ] **Aktualizacja funkcji**
- [ ] **Zastosowanie REPL**

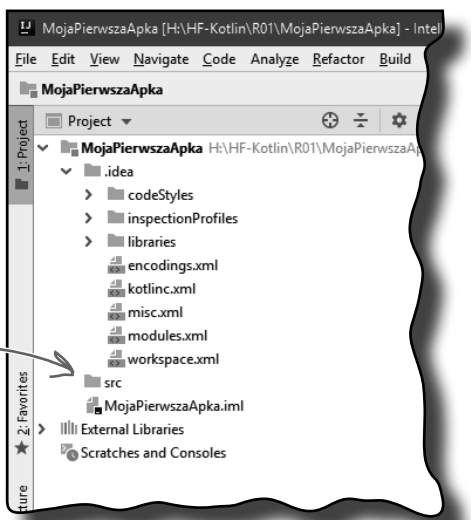
Po wykonaniu wszystkich czynności związanych z tworzeniem nowego projektu IntelliJ IDEA przygotuje go i wyświetli. Tak wygląda projekt przygotowany dla nas przez IDE:



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 wchodzą 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 Kotliczynie 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.

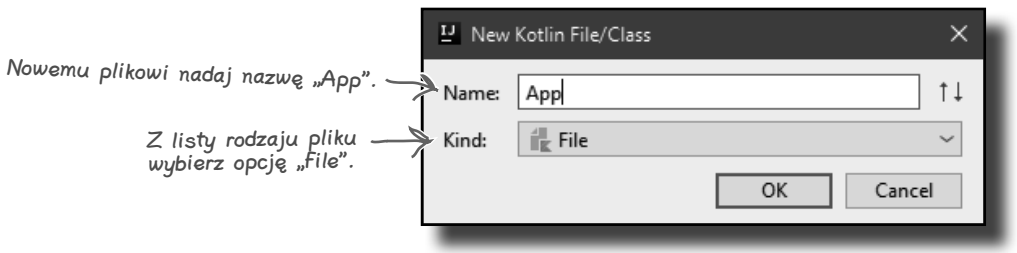


Dodaj do projektu nowy plik Kotlina

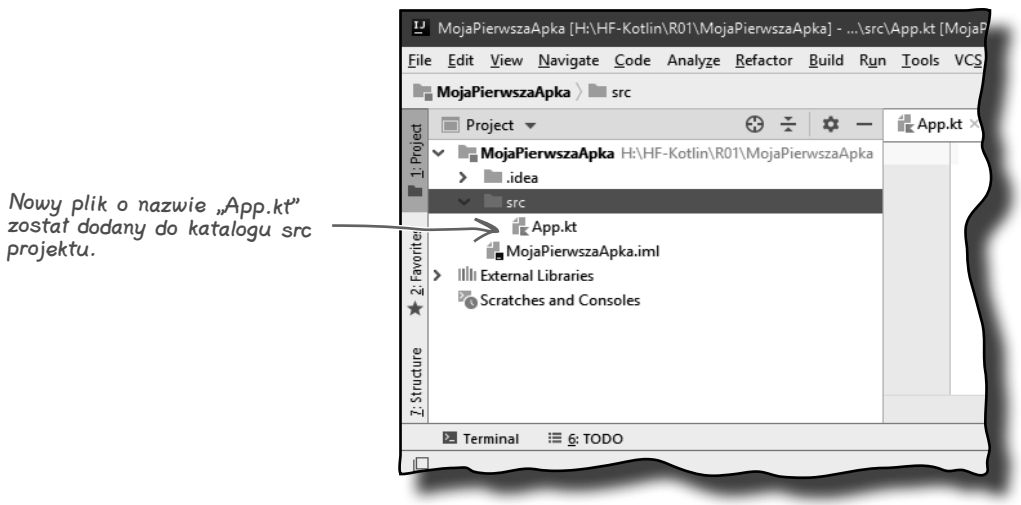
- ☑ Zbudowanie aplikacji
- → ☐ **Dodanie funkcji**
- ☐ Aktualizacja funkcji
- ☐ Zastosowanie REPL

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 InetlliJ 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:



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



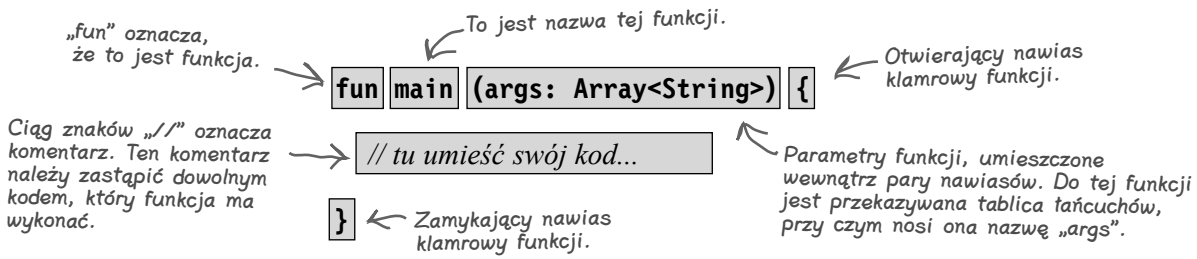
Zobaczmy teraz, jaki kod musimy dodać do pliku *App.kt*, aby zaczął on cokolwiek robić.

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:

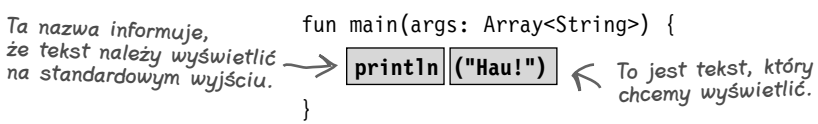


Funkcja rozpoczyna się od słowa **`fun`**, które informuje kompilator Kotlina, że to jest funkcja. Słowa kluczowego `fun` musisz używać podczas tworzenia w Kotlinie każdej nowej funkcji.

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:



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.

Bezparametrowe funkcje main

Jeśli używasz Kotlina 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:

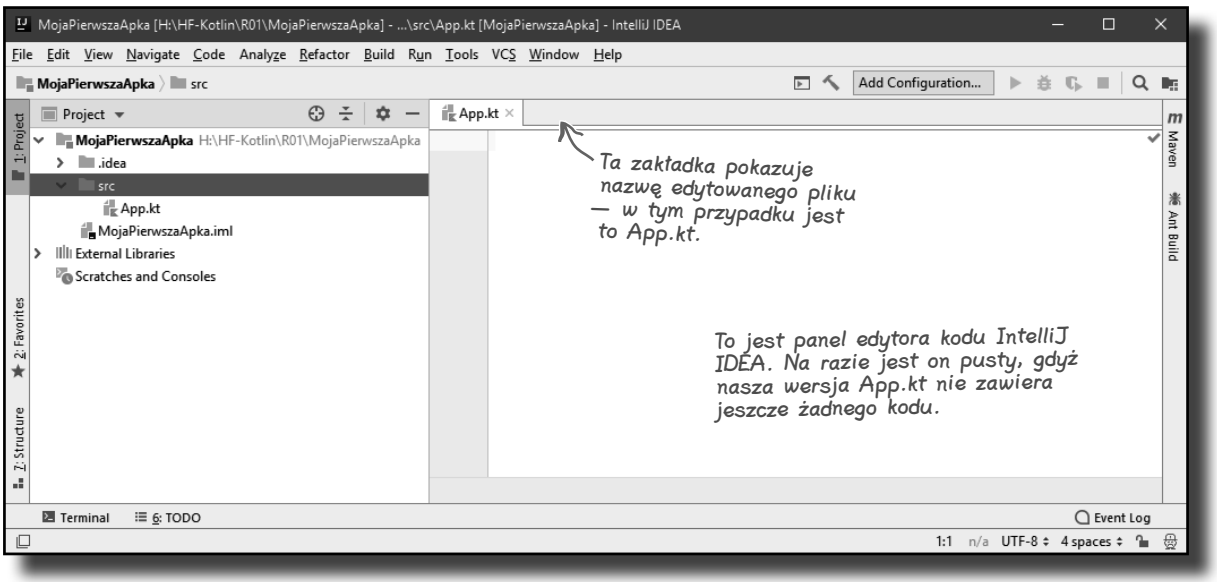
```
fun main() {
    // tu umieść swój kod
}
```

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

Dodaj funkcję main do pliku App.kt

- [x] Zbudowanie aplikacji
- [ ] **Dodanie funkcji**
- [ ] Aktualizacja funkcji
- [ ] Zastosowanie REPL

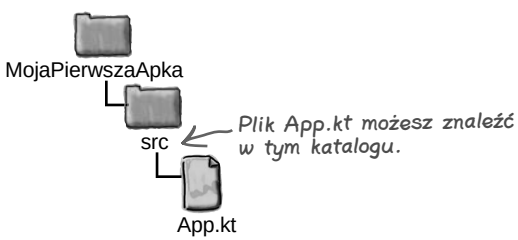
Aby dodać funkcję `main` do projektu, otwórz plik *App.kt*, dwukrotnie klikając go w eksploratorze IntelliJ IDEA. Spowoduje to otworzenie pliku w edytorze kodu, jak pokazaliśmy poniżej:



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 głupie pytania

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

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.

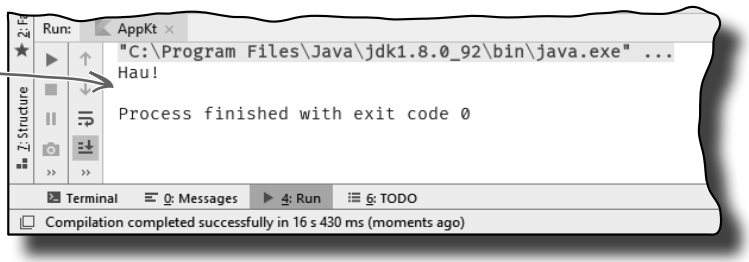
Jazda próbna

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [ ] Aktualizacja funkcji
- [ ] Zastosowanie REPL

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!”:

To jest okno wyników IDE.

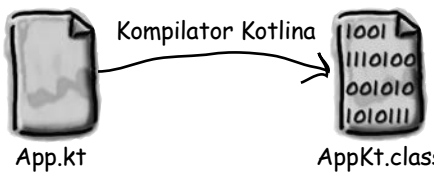


Co robi polecenie Run?

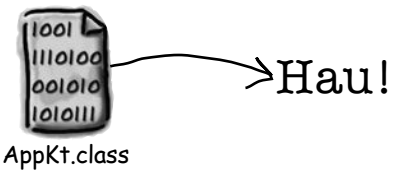
Po wykonaniu polecenia *Run*, zanim zostaną wyświetlone wyniki, IntelliJ IDEA wykonuje kilka czynności:

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?

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [ ] **Aktualizacja funkcji**
- [ ] Zastosowanie REPL

Kiedy już znajdziemy się wewnątrz funkcji `main` (czy też jakiejkolwiek 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.

Składnia pod lupą

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

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

// To jest komentarz

✶ W większości przypadków odstępy 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...

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [ ] **Aktualizacja funkcji**
- [ ] Zastosowanie REPL

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.

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

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 *fałszem* (`false`). Na przykład jeśli powiemy coś takiego: „Tak długo, jak *sąLodyWWafelku*, liżemy”, będzie to zrozumiały test warunkowy. Lody w wafelku mogą być lub może ich w nim już nie być. Jeśli jednak powiemy: „Tak długo jak *Ferdek*, oglądamy”, to tak naprawdę nie jest to żaden test. Musielibyśmy go zmienić na coś w rodzaju: „Tak długo jak *Ferdek leci w TV*, oglądamy”, aby stwierdzenie miało sens.

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

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [ ] **Aktualizacja funkcji**
- [ ] Zastosowanie REPL

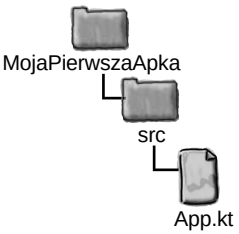
Zaktualizujemy kod w pliku *App.kt*, umieszczając w nim nową wersję funkcji `main`. Funkcję tę zmodyfikujemy w taki sposób, by wyświetlała komunikat przed uruchomieniem pętli, podczas każdej iteracji pętli oraz po jej zakończeniu.

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

```
fun main(args: Array<String>) {
    println("Hau!")
    var x = 1
    println("Przed pętlą. x = $x.")
    while (x < 4) {
        println("Wewnątrz pętli. x = $x.")
        x = x + 1
    }
    println("Za pętlą. x = $x.")
}
```

Usuń ten wiersz — nie jest już nam potrzebny.

Ten zapis wyświetla wartość zmiennej x.



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` — wyświetl 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.

Używanie if do zwracania wartości

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [ ] **Aktualizacja funkcji**
- [ ] Zastosowanie REPL

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:

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

```
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")
```

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”.

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 zwięzły.

Na następnej stronie przedstawimy kompletny kod naszej funkcji `main`.

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.

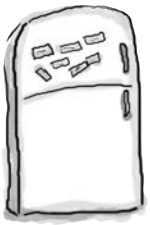
Weźmy teraz ten kod na jazdę próbną.

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.



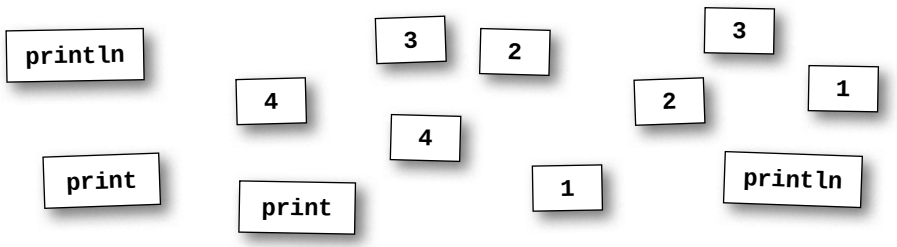
Magnesiki z kodem

Ktoś użył magnesików, by ułożyć na lodówce nową, przydatną wersję funkcji **main**, która wyświetla tekst „YabbaDabbaDo". Niestety niewyjaśnione kuchenne tornado poprzestawiało 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.

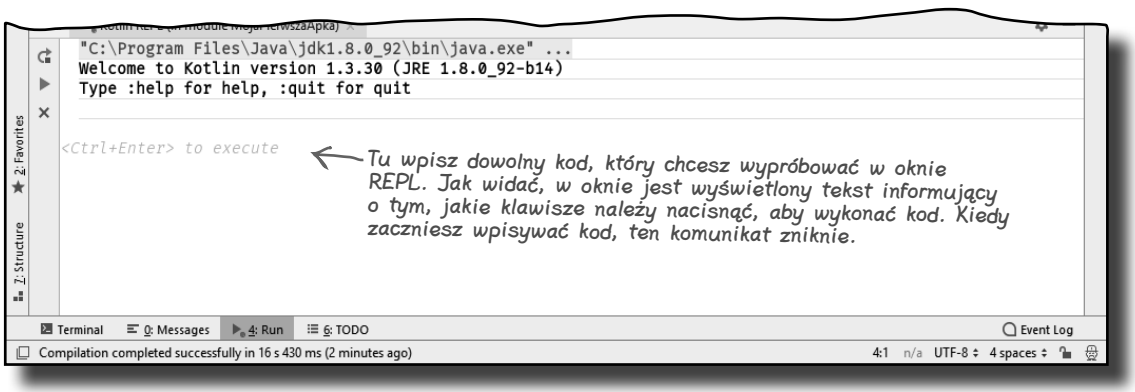
Stosowanie interaktywnej powłoki Kotlina

- [x] Zbudowanie aplikacji
- [x] Dodanie funkcji
- [x] Aktualizacja funkcji
- [ ] Zastosowanie REPL

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.

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

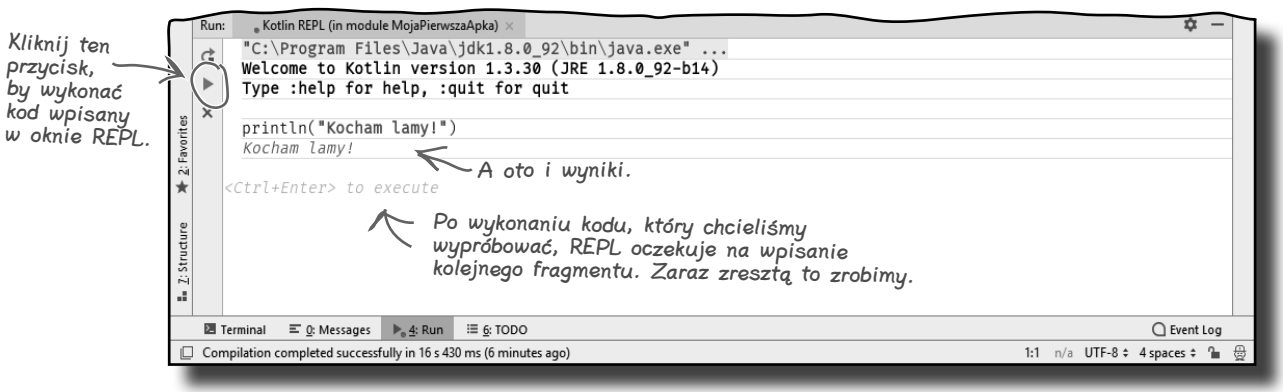
Narzędzie REPL można otworzyć, wyświetlając w menu głównym opcję *Tools*, a następnie wybierając opcję *Kotlin/Kotlin REPL*. Po jej wybraniu w dolnej części IntelliJ IDEA pojawi się okno przedstawione poniżej.



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

- ☑ Zbudowanie aplikacji
- ☑ Dodanie funkcji
- ☑ Aktualizacja funkcji
- → ☑ Zastosowanie REPL

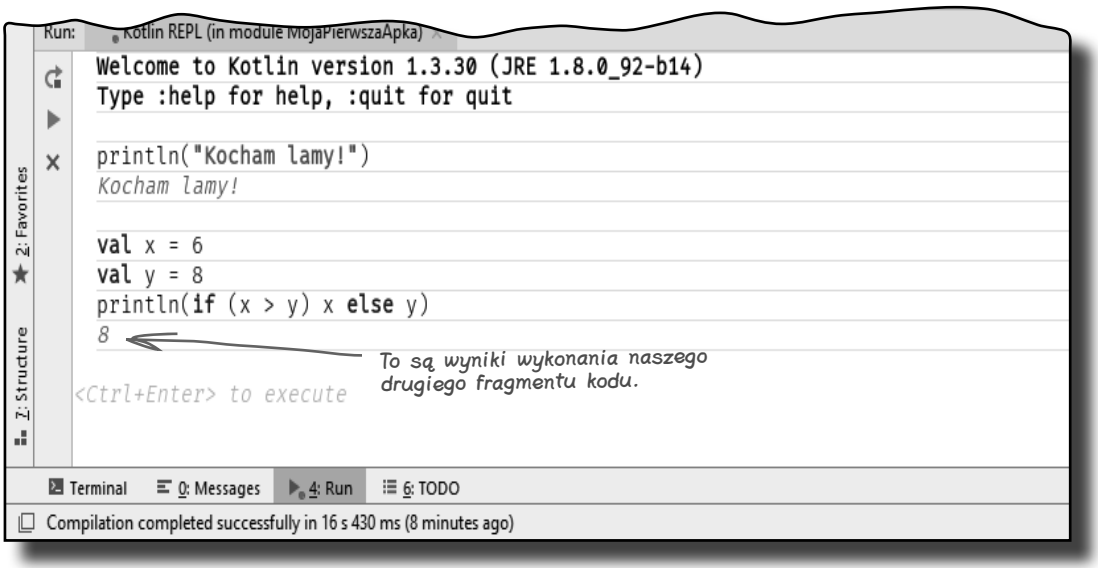
Wykonaliśmy już wszystkie czynności przewidziane w tym rozdziale.

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.

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?

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>) {
    val 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) {
        [                                ]   <- Proponowany blok kodu
        print("$x$y ")
        x = x + 1
    }
}
```

Dopasuj każdy z proponowanych bloków kodu z możliwymi wynikami.

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
```

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
```

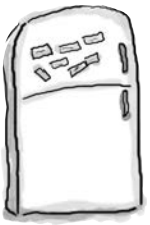
```
00 11 21 32 42
```

```
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 poprzestawiało 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")
}
```

println

4

2

4

2

1

println

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 i else`, można dodawać do kodu rozgałęzienia warunkowe.
- `if` można także używać jako wyrażenia, które zwraca wartość. W takim przypadku zastosowanie klauzuli `else` jest obowiązkowe.

O'REILLY®

Rusz głową!

Kotlin

Kariera Kotlina 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 Kotlina, 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

helion.pl

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

ISBN 978-83-283-5869-0

9 788328 358690

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł