O książce

W uproszczeniu, nowe funkcje w Javie 8, razem z (mniej oczywistymi) zmianami w Javie 9, są największą zmianą dokonaną w Javie w ciągu ostatnich 21 lat od momentu wydania Javy 1.0. Nic nie zostało usunięte, tak więc cały Twój istniejący kod Java będzie nadal działał – ale nowe funkcje dostarczają wiele nowych potężnych idiomów i wzorców projektowych, które pomogą Ci pisać bardziej przejrzysty i zwięzły kod. Na początku możesz pomyśleć (jak w przypadku każdej nowej funkcji), „Dlaczego oni znowu zmieniają mój język?". Ale potem, po odrobinie praktyki, nadchodzi objawienie, że oto dzięki tym nowym funkcjom jesteś w stanie pisać krótszy i bardziej przejrzysty kod, i to w zaledwie połowę czasu, jakiego oczekiwałeś – i uświadamiasz sobie, że już nigdy nie byłbyś w stanie wrócić z powrotem do „starej Javy".

Druga edycja tej książki, „Nowoczesna Java w działaniu: wyrażenia lambda, strumienie oraz programowanie funkcyjne i reaktywne", powstała po to, aby pomóc Ci przejść przez te pierwsze trudy typu „brzmi to dobrze w teorii, ale to wszystko jest zbyt nowe i mało znane" i umożliwić Ci biegłe kodowanie.

„Być może", możesz pomyśleć, „ale wyrażenia lambda i programowanie funkcyjne – czy nie są to te rzeczy, o których brodaci profesorowie w sandałach rozmawiają w swoich wieżach z kości słoniowej?". Może i to prawda, ale Java 8 wprowadza odpowiednio zbalansowaną ilość tych idei, tak aby ich zalety można było wykorzystać w sposób, który jest możliwy do przyswojenia przez zwykłego programistę Java. Książka ta opowiada historię z punktu widzenia zwykłego programisty, dodając niekiedy wzmianki „jak to powstało" w celu nakreślenia szerszej perspektywy.

„Wyrażenia lambda – to kojarzy mi się z Grecją!". Dokładnie tak, ale jest to naprawdę świetna koncepcja pozwalająca na pisanie zwięzłych programów w Javie. Wielu z Was jest zaznajomionych z metodami obsługi zdarzeń i wywołaniami zwrotnymi, gdzie rejestrujemy obiekt zawierający metodę, która ma zostać wykorzystana w przypadku wystąpienia określonego zdarzenia. Wyrażenia lambda sprawiają, że tego typu idee mogą być stosowane w Javie na dużo szerszą skalę. Krótko mówiąc, wyrażenia lambda i ich przyjaciele, referencje do metod, dostarczają możliwość zwięzłego przekazywania kodu lub metod jako argumentów do wykonania w środku innego kodu. W książce tej zobaczysz, że koncepcja ta używana jest znacznie częściej, niż Ci się wydaje: od prostego parametryzowania metody

sortowania za pomocą kodu porównującego, aż po wyrażanie złożonych zapytań na kolekcjach danych z użyciem nowego API strumieni.

„Strumienie – czym one są?". Są one świetnym dodatkiem wprowadzonym w Javie 8. Zachowują się jak kolekcje, ale mają w stosunku do nich kilka zalet, które pozwalają nam programować w nowych stylach. Po pierwsze, jeśli kiedykolwiek programowałeś z użyciem języka zapytań bazy danych, takiego jak SQL, to wiesz, że umożliwiają one pisanie zwięzłych zapytań, które w Javie zajęłyby wiele wierszy. Strumienie w Javie 8 obsługują ten zwięzły styl programowania zapytań wzorowany na bazach danych, ale za pomocą składni języka Java i bez konieczności posiadania jakiejkolwiek wiedzy w zakresie baz danych! Po drugie, strumienie zaprojektowane są w taki sposób, że nie wszystkie ich dane muszą znajdować się w pamięci (lub nawet być obliczone) w tym samym momencie. Z tego powodu możemy przetwarzać nawet takie strumienie, które są zbyt duże, aby zmieściły się w pamięci naszego komputera. Java 8 może optymalizować operacje na strumieniach w sposób, jaki nie jest możliwy dla kolekcji – na przykład, może ona pogrupować ze sobą kilka operacji do wykonania na tym samym strumieniu, dzięki czemu jego dane odwiedzane są tylko raz, zamiast kosztownego przechodzenia po nich po kilka razy. Co więcej, Java może automatycznie zrównoleglać operacje wykonywane na strumieniach (w przeciwieństwie do kolekcji).

„A programowanie funkcyjne, czym ono jest?". Jest to kolejny styl programowania, podobnie jak programowanie obiektowo zorientowane, skupiające się na wykorzystywaniu funkcji jako wartości, jak wspomnieliśmy wcześniej podczas omawiania wyrażeń lambda.

Świetne w Javie 8 jest to, że najlepsze idee programowania funkcyjnego implementuje ona w postaci znanej nam składni Java. Dobre wybory projektowe pozwalają nam postrzegać programowanie w stylu funkcyjnym w Javie 8 jako dodatkowy zestaw wzorców projektowych lub idiomów, umożliwiających nam pisanie bardziej przejrzystego i zwięzłego kodu w dużo krótszym czasie. Możesz to przyrównać do posiadania szerszego zakresu broni w swoim arsenale programowania.

O tak, poza tymi dodatkowymi funkcjami, które skupiają się na dużych konceptualnych dodatkach do Javy, omawiamy również wiele innych przydatnych funkcji i aktualizacji Javy 8, takich jak metody domyślne, nowa klasa `Optional`, obiekty `CompletableFuture` oraz nowe API daty i godziny.

Są jeszcze dodatki wprowadzone w Javie 9: nowy system modułów, wsparcie dla programowania reaktywnego poprzez API `Flow`, oraz różne inne rozszerzenia.

Ale zaraz, to tylko przegląd – czas teraz przystąpić do czytania tej książki.

Jak zorganizowana jest ta książka: mapa drogowa

Nowoczesna Java w działaniu podzielona jest na sześć części: „Podstawy", „Funkcyjne przetwarzanie danych z użyciem strumieni", „Efektywne programowanie z użyciem strumieni i wyrażeń lambda", „Java na co dzień", „Rozszerzona współbieżność Javy" oraz „Programowanie funkcyjne i dalszy rozwój języka Java". Choć gorąco zalecamy, abyś przeczytał najpierw rozdziały z dwóch pierwszych części (w kolejności, ponieważ wiele prezentowanych koncepcji opiera się na poprzednich rozdziałach), to pozostałe cztery części mogą być w zasadzie czytane niezależnie. Większość rozdziałów zawiera po kilka quizów, które pomogą Ci przećwiczyć zawarty w nich materiał.

Pierwsza część tej książki omawia podstawy, które pomogą Ci zapoznać się z nowymi ideami wprowadzonymi do Javy 8. Pod koniec tej pierwszej części będziesz mieć pełną wiedzę na temat tego, czym są wyrażenia lambda, i będziesz w stanie pisać kod, który jest na tyle zwięzły i elastyczny, aby mógł łatwo dostosowywać się do zmieniających się wymagań.

- W rozdziale 1 podsumowujemy główne zmiany wprowadzone do Javy (wyrażenia lambda, referencje do metod, strumienie i metody domyślne) oraz nakreślimy scenę dla tej książki.
- W rozdziale 2 zapoznasz się z parametryzacją zachowania – wzorcem rozwoju oprogramowania, na którym Java 8 w dużym stopniu bazuje, i który jest motywacją dla wyrażeń lambda.
- W rozdziale 3 w pełni wyjaśnimy koncepcję wyrażeń lambda i referencji do metod, podając przy tym przykładowy kod i quizy do rozwiązania.

Druga część tej książki stanowi głęboką eksplorację nowego API strumieni, które pozwala na pisanie potężnego kodu do przetwarzania kolekcji danych w sposób deklaratywny. Pod koniec tej drugiej części będziesz mieć pełną wiedzę na temat tego, czym są strumienie oraz w jaki sposób możesz używać ich w swoim kodzie do zwięzłego i efektywnego przetwarzania kolekcji danych.

- Rozdział 4 wprowadza koncepcję strumienia i wyjaśnia, czym różni się on od kolekcji.
- Rozdział 5 omawia szczegółowo operacje dostępne na strumieniach, takie jak filtrowanie, dzielenie, wyszukiwanie, dopasowywanie, mapowanie i redukowanie.
- Rozdział 6 omawia kolektory, będące funkcją API strumieni, która pozwala na wyrażanie nawet bardzo skomplikowanych zapytań przetwarzania danych.
- W rozdziale 7 dowiesz się, jak strumienie mogą automatycznie wykonywać się równolegle i wykorzystywać Twoją architekturę wielordzeniową. Dodatkowo zapoznasz się z różnymi pułapkami, na które możesz natknąć się nawet podczas poprawnego i efektywnego korzystania ze strumieni równoległych.

Trzecia część tej książki omawia różne tematy Javy 8 i Javy 9, które pozwolą Ci bardziej efektywnie korzystać z Javy i rozszerzą Twój kod o nowoczesne idiomy. Ponieważ zawarte w niej rozdziały ukierunkowane są na bardziej zaawansowane idee programistyczne, żaden z następujących po nich rozdziałów nie zależy od technik przedstawionych w tej części.

- Rozdział 8 jest nowym rozdziałem w drugiej edycji tej książki i wyjaśnia rozszerzenia wprowadzone do API kolekcji Javy 8 i Javy 9. Omawia on korzystanie z fabryk kolekcji oraz nowych idiomatycznych wzorców do pracy z kolejkami `List` i `Set`, wraz z idiomatycznymi wzorcami dotyczącymi kolekcji `Map`.
- Rozdział 9 pokazuje, w jaki sposób możesz usprawnić swój dotychczasowy kod za pomocą nowych funkcji Javy 8 i kilku przepisów. Dodatkowo omawia on pożyteczne techniki rozwoju oprogramowania, takie jak wzorce projektowe, refaktoryzacja, testowanie i debugowanie.
- Rozdział 10 również jest nowością w drugiej edycji tej książki. Omawia on ideę dotyczącą tworzenia nowego API w oparciu o język dziedzinowy. Jest to nie tylko potężny

sposób projektowania API, ale staje się on coraz bardziej popularny i zaczyna pojawiać się w klasach Javy, takich jak `Comparators`, `Stream` i `Collectors`.

Czwarta część tej książki omawia różne nowe funkcje w Javie 8 i Javie 9, dzięki którym będziesz w stanie pisać prostszy i bardziej niezawodny kod w swoich projektach. Rozpoczynamy od dwóch API wprowadzonych do Javy 8.

- Rozdział 11 omawia klasę `java.util.Optional`, która umożliwi Ci zarówno projektowanie lepszych API, jak i zredukowanie liczby wyjątków pustego wskaźnika.
- Rozdział 12 stanowi przegląd nowego API daty i godziny, które znacząco usprawnia poprzednie i podatne na błędy API przeznaczone do pracy z datami i godzinami.
- W rozdziale 13 dowiesz się, czym są metody domyślne i jak możesz za ich pomocą rozwijać swoje API z zachowaniem wstecznej kompatybilności, a także poznasz praktyczne wzorce użycia i reguły dotyczące efektywnego wykorzystywania metod domyślnych.
- Rozdział 14 jest nowością w tej edycji i omawia system modułów Javy, będący głównym rozszerzeniem w Javie 9. Pozwala on na modularyzowanie dużych systemów w udokumentowany i wykonalny sposób, zamiast konstruowania ich z „chaotycznej kolekcji pakietów".

Piąta część tej książki omawia bardziej zaawansowane sposoby strukturyzacji współbieżnych programów w Javie, wykraczając poza idee łatwego w użyciu przetwarzania równoległego dla strumieni z rozdziałów 6 i 7.

- Rozdział 15 jest nowością w tej edycji i stanowi wprowadzenie do asynchronicznych API – wliczając w to idee obiektów `Future` oraz protokołu publikacji i subskrypcji stojących za programowaniem reaktywnym, enkapsulowanym w API `Flow` Javy 9.
- Rozdział 16 omawia klasę `CompletableFuture`, która pozwala na wyrażanie złożonych asynchronicznych obliczeń w sposób deklaratywny – na wzór API strumieni.
- Rozdział 17 również jest nowością w drugiej edycji tej książki i szczegółowo omawia API `Flow` Javy 9, skupiając się na praktycznym kodzie programowania reaktywnego.

W szóstej i ostatniej części tej książki wykonujemy mały krok wstecz i pokazujemy, w jaki sposób możesz pisać w Javie efektywne programy w stylu funkcyjnym, porównując dodatkowo funkcje Javy 8 z funkcjami języka Scala.

- Rozdział 18 omawia szczegółowo koncepcję programowania funkcyjnego, wprowadza związaną z nim terminologię i wyjaśnia, jak pisać programy w stylu funkcyjnym w Javie.
- Rozdział 19 omawia bardziej zaawansowane techniki programowania funkcyjnego, wliczając w to funkcje wyższego rzędu, rozwijanie trwałych struktur danych, leniwe listy i dopasowywanie do wzorców. Rozdział ten możesz postrzegać jako miks praktycznych technik do zastosowania w swoim kodzie, jak również zestaw informacji akademickich, które uczynią Cię bardziej wszechstronnym programistą.
- Rozdział 20 stanowi porównanie funkcji Javy 8 z funkcjami dostępnymi w Scali – języku, który podobnie jak Java zaimplementowany jest na bazie wirtualnej maszyny Javy,

i który wyewoluował na tyle szybko, że dziś jest w stanie zagrozić pewnym aspektom niszy, w jakiej Java operuje w ekosystemie języków programowania.

- W rozdziale 21 podsumujemy naszą podróż po języku Java 8 w kierunku programowania w stylu funkcyjnym. Dodatkowo spekulujemy w nim odnośnie tego, jakie przyszłe usprawnienia i świetne nowe funkcje mogą trafić do przepływu Javy po wydaniach Javy 8 i 9, a także małych usprawnieniach w Javie 10.

Na końcu tej książki znajdują się cztery dodatki, które omawiają kilka innych tematów związanych z Javą 8. Dodatek A podsumowuje pomniejsze funkcje języka Java 8, które pominęliśmy w treści tej książki. Dodatek B stanowi przegląd głównych dodatków wprowadzonych do bibliotek Javy, które mogą być dla Ciebie przydatne. Dodatek C jest kontynuacją części 2 i skupia się na zaawansowanym wykorzystywaniu strumieni. Dodatek D omawia, w jaki sposób kompilator Javy implementuje za kulisami wyrażenia lambda.

O kodzie

Cały kod źródłowy zawarty w listingach lub tekście tej książki pisany jest za pomocą `czcionki o stałej szerokości znaków`, aby lepiej wyróżniał się on na tle zwykłego tekstu. Wielu listingom towarzyszą adnotacje kodu, które podkreślają istotne koncepcje.

Kod źródłowy dla wszystkich działających przykładów z tej książki oraz instrukcje do ich uruchamiania dostępne są w repozytorium GitHub oraz na stronie samej książki. Łącza prowadzące do kodu źródłowego dostępne są pod adresem *www.manning.com/books/modern-java-in-action*.

O autorach

Raoul-Gabriel Urma jest prezesem i współzałożycielem programu Cambridge Spark, wiodącej społeczności edukacyjnej dla naukowców programistów zajmujących się przetwarzaniem danych. Raoul został odznaczony tytułem Java Champion w 2017 roku. Pracował dla takich firm, jak Google, eBay, Oracle i Goldman Sachs. Raoul uzyskał tytuł doktora informatyki na Uniwersytecie w Cambridge. Z kolei na uniwersytecie Imperial College London uzyskał tytuł magistra inżyniera informatyki (z wyróżnieniem), zdobywając tam wiele nagród za innowacje techniczne. Raoul przeprowadził ponad 100 prezentacji technicznych na międzynarodowych konferencjach.

Mario Fusco jest starszym inżynierem oprogramowania w Red Hat, gdzie pracuje nad rozwojem narzędzia Drools – silnika reguł JBoss. Posiada ogromne doświadczenie jako programista Java – zaangażowany w liczne projekty klasy enterprise (którym często przewodzi) w kilku obszarach przemysłu, zaczynając od organizacji medialnych, a kończąc na sektorze finansowym. Do jego zainteresowań należą programowanie funkcyjne i języki dziedzinowe. Wykorzystując te dwie pasje stworzył on otwartoźródłową bibliotekę lambdaj, mając na celu dostarczenie w Javie wewnętrznego języka dziedzinowego do manipulowania kolekcjami i umożliwienie zastosowania programowania funkcyjnego w Javie.

Alan Mycroft jest profesorem informatyki na wydziale Computer Laboratory na Uniwersytecie w Cambridge, którego jest członkiem od 1984 roku. Jest pracownikiem uczelni Robinson College, współzałożycielem Europejskiego Stowarzyszenia dla Języków Programowania i Systemów, oraz współzałożycielem i członkiem zarządu organizacji Raspberry Pi Foundation. Posiada wykształcenie matematyczne (Cambridge) i informatyczne (Edynburg). Jest autorem ponad 100 publikacji naukowych i promotorem ponad 20 dysertacji. Jego badania koncentrują się na językach programowania oraz ich semantyce, optymalizacji i implementacji. Jest mocno związany z przemysłem, pracował dla AT&T Laboratories i Intel Research, a także dla rozwijającej się firmy Codemist Ltd., która zbudowała oryginalny kompilator ARM C pod nazwą Norcroft.

O ilustracji na okładce

Rysunek na okładce tego podręcznika podpisany jest jako „Habit mandaryna wojskowego w Chińskiej Tartarii w 1700 roku". Mandaryn nosi przy sobie miecz oraz łuk i kołczan na plecach, a jego habit jest odpowiednio przyozdobiony. Jeśli zwrócisz szczególną uwagę na jego pas, zauważysz tam klamrę w kształcie litery lambda (dodaną przez naszego projektanta jako ukłon w stronę jednego z tematów tej książki). Ilustracja ta pochodzi z książki Thomasa Jefferysa pt. „A Collection of the Dresses of Different Nations, Ancient and Modern" (Kolekcja szat różnych nacji, starożytnych i współczesnych), opublikowanej w Londynie między 1757 a 1772 rokiem. Strona tytułowa podaje, że są to ręcznie kolorowane miedzioryty, wykończone gumą arabską. Thomas Jefferys (1719-1771), nazywany „geografem króla Jerzego III", był angielskim kartografem i czołowym dostawcą map. Rytował i drukował on mapy dla rządu i innych oficjalnych instytucji, produkując przy tym szeroki zakres komercyjnych map i atlasów, zwłaszcza dla Ameryki Północnej. Jego praca jako twórcy map wzbudziła w nim zainteresowanie zwyczajami w zakresie lokalnego ubioru w krajach, w których sporządzał swoje mapy; szaty te zostały pięknie zaprezentowane w tej czteroczęściowej kolekcji.

Fascynacja odległymi krajami i podróżowanie dla przyjemności były w osiemnastym wieku względnie nowym fenomenem, a kolekcje, takie jak ta, były dosyć popularne, pozwalając zarówno prawdziwym, jak i kanapowym podróżnikom zapoznać się z mieszkańcami innych krajów. Różnorodność rysunków w woluminach Jefferysa świadczy o unikalności i indywidualności krajów świata w tamtym czasie. Od tamtej pory zasady ubioru zmieniły się, a różnorodność w poszczególnych regionach i krajach, w pewnym momencie tak bardzo bogata, niemal całkowicie wygasła. Dzisiaj trudno jest czasem rozróżnić mieszkańców jednego kontynentu od drugiego. Być może, patrząc na to nieco optymistycznie, zamieniliśmy kulturową i wizualną różnorodność na bardziej urozmaicone życie osobiste – lub bardziej urozmaicone i interesujące życie intelektualne i techniczne.

W czasach, gdy ciężko jest odróżnić jedną książkę informatyczną od drugiej, wydawnictwo Manning celebruje innowacyjność i inicjatywę biznesu komputerowego przy użyciu okładek dla książek, które oparte są bogatej różnorodności strojów ludowych noszonych trzy stulecia temu, przywróconych do życia za pomocą obrazów wykonanych przez Jefferysa.

Część 1

Podstawy

Pierwsza część tej książki omawia podstawy, które pomogą zapoznać się z nowymi koncepcjami języka Java wprowadzonymi w Javie 8. Po przyswojeniu zawartego tu materiału zdobędziesz wiedzę na temat wyrażeń lambda, a także nauczysz się pisać zwięzły i elastyczny kod, który łatwo będzie można dostosować pod nowe wymagania.

W rozdziale 1 podsumujemy główne zmiany wprowadzone do języka Java (wyrażenia lambda, referencje do metod, strumienie i metody domyślne) oraz nakreślimy scenę dla tej książki.

W rozdziale 2 porozmawiamy o parametryzacji zachowania, czyli o wzorcu w rozwoju oprogramowania, który stanowi solidną podstawę dla języka Java 8 i wyrażeń lambda.

Rozdział 3 wyczerpująco objaśnia koncepcje wyrażeń lambda i referencji do metod, dostarczając przy tym wiele przykładów kodu i quizów do rozwiązania.

1 Java 8, 9,10 i 11: co się dzieje?

W tym rozdziale:

- Dlaczego Java ciągle się zmienia?
- Nowe oblicze przetwarzania
- Nacisk na rozwój Javy
- Wprowadzenie do nowych funkcji Javy 8 i 9

Od czasu wydania pakietu JDK 1.0 (Java 1.0) w 1996 roku, Java zdobyła sobie uznanie dużej liczby studentów, menedżerów projektów i programistów będących jej czynnymi użytkownikami. Java jest niezwykle ekspresyjnym językiem, który nadal wykorzystywany jest przy małych i dużych projektach. Proces jej ewolucji (poprzez wzbogacanie jej o nowe funkcje) od Javy 1.1 (1997) do Javy 7 (2011) zarządzany był bardzo dobrze. Java w wersji 8 została wydana w marcu 2014 roku, w wersji 9 we wrześniu 2017 roku, w wersji 10 w marcu 2018 roku, a Java 11 planowana jest na wrzesień 2018. Nasuwa się pytanie: dlaczego powinniśmy w ogóle przejmować się tymi zmianami?

1.1 *Jak to wszystko wygląda?*

Uważamy, że zmiany wprowadzone do Javy 8 były pod wieloma względami dużo bardziej znaczące, niż jakiekolwiek inne zmiany poczynione w dotychczasowej historii tego języka (Java 9 dodaje ważne, ale jednak mniej znaczące zmiany w zakresie produktywności, o czym przekonamy się w dalszej części tego rozdziału, natomiast Java 10 wprowadza znacznie mniejsze poprawki w obszarze wnioskowania typów). Dobra wiadomość jest taka, że zmiany te pozwalają nam łatwiej pisać programy. Na przykład, zamiast pisać rozwlekły kod (sortujący jabłka w kolekcji `inventory` na podstawie ich wagi) jak w poniższym przykładzie

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

w języku Java 8 możemy napisać bardziej zwięzły kod, który będzie dużo bliższy słownej definicji naszego problemu:

```
inventory.sort(comparing(Apple::getWeight));                    (1)
```

(1) Pierwszy kod Java 8 w tej książce!

Kod ten możemy odczytać jako „posortuj inwentarz poprzez porównywanie wag jabłek". Nie przejmuj się na razie tym kodem. W dalszej części tej książki dowiesz się, co on robi, i w jaki sposób możesz samodzielnie napisać podobny kod.

Sprzęt komputerowy również ma tutaj pewien wpływ: oferowane na rynku procesory stały się wielordzeniowe – procesor w Twoim laptopie lub komputerze stacjonarnym zapewne zawiera w sobie cztery lub więcej rdzeni. Jednak zdecydowana większość istniejących programów pisanych w Javie wykorzystuje tylko jeden z tych rdzeni, pozostawiając pozostałe trzy w stanie bezczynności (ewentualnie wykorzystują mały ułamek ich mocy obliczeniowej do uruchamiania fragmentów systemu operacyjnego lub skanerów antywirusowych).

Przed wydaniem Javy 8 eksperci mówili nam, że do wykorzystania tych rdzeni powinniśmy używać wątków. Problem polega na tym, że praca z wątkami jest trudna i może prowadzić do powstania wielu błędów. W trakcie swojej ewolucji Java nieustannie starała się uczynić współbieżność znacznie łatwiejszą i mniej kłopotliwą. Java 1.0 oferowała wątki i blokady, a nawet model pamięci – będący w tamtym czasie najlepszą praktyką – ale te prymitywne konstrukcje okazały się być zbyt trudne, aby można je było niezawodnie stosować w niespecjalistycznych zespołach projektowych. Java 5 przyniosła nam wysokiej klasy elementy konstrukcyjne, takie jak pule wątków i współbieżne kolekcje. Java 7 dodała strukturę fork/join, co równoległość uczyniło bardziej praktyczną, ale nadal trudną w obsłudze. Z kolei Java 8 dała nam nowy, prostszy sposób myślenia o równoległości. Nadal jednak musimy stosować się do pewnych reguł, z którymi zapoznamy się w tej książce.

Jak zobaczymy w dalszej części tej książki, Java 9 przynosi nam dodatkowy sposób na strukturyzację współbieżności – programowanie reaktywne. Choć jego zastosowanie jest nieco bardziej specjalistyczne, standaryzuje ono sposób korzystania z narzędzi

strumieni reaktywnych RxJava i Akka, które są coraz częściej stosowane w systemach o wysokiej współbieżności.

Z powyższych dwóch dezyderatów (bardziej zwięzły kod i prostsze użycie procesorów wielordzeniowych) powstaje pełna i spójna konstrukcja wykorzystywana przez Javę 8. Rozpoczniemy od krótkiego przedstawienia poniższych idei (miejmy nadzieję wystarczająco długiego, by wzbudziły Twoje zainteresowanie, ale wystarczająco krótkiego, aby było to jedynie podsumowanie):

- Interfejs API strumieni
- Techniki przekazywania kodu do metod
- Metody domyślne w interfejsach

Java 8 dostarcza nowe API strumieni, które wspiera wiele równoległych operacji do przetwarzania danych i odzwierciedla sposób myślenia w językach zapytań baz danych – na wysokim poziomie abstrakcji wyrażamy to, co chcemy osiągnąć, a implementacja (w tym wypadku biblioteka Streams) wybiera najlepszy niskopoziomowy mechanizm wykonania. W rezultacie unikamy pisania kodu wykorzystującego słowo kluczowe `synchronized`, który nie tylko jest wysoce podatny na błędy, ale w przypadku wielordzeniowych procesorów może okazać się znacznie bardziej kosztowny niż się nam wydaje*.

Z nieco rewizjonistycznego punktu widzenia, pojawienie się strumieni w języku Java 8 może być postrzegane jako bezpośrednia przyczyna wprowadzenia do niej dwóch innych elementów: *zwięzłych technik przekazywania kodu do metod* (referencje do metod, wyrażenia lambda) i *metod domyślnych* w interfejsach.

Jednak myślenie o przekazywaniu kodu do metod jako po prostu konsekwencji wprowadzenia strumieni zaniża zakres ich użycia w Javie 8. Otrzymujemy tu nowy zwięzły sposób na wyrażanie parametryzacji zachowania. Załóżmy, że chcemy napisać dwie metody, które różnią się tylko kilkoma wierszami kodu. Teraz możemy po prostu przekazać kod różniących się fragmentów w formie argumentu (ta technika programowania jest krótsza, bardziej przejrzysta i mniej podatna na błędy niż powszechna tendencja do kopiowania i wklejania kodu). Eksperci zapewne zauważą tu, że przed wydaniem Javy 8 parametryzacja zachowania mogła zostać zakodowana z użyciem anonimowych klas – niechaj jednak przykład zamieszczony na pierwszej stronie tego rozdziału, który pokazuje zwiększoną zwięzłość kodu w Javie 8, mówi sam za siebie w kontekście oferowanej przejrzystości!

Funkcja przekazywania kodu do metod w Javie 8 (w tym również możliwość zwrócenia go i wcielenia do struktur danych) daje nam również dostęp do szerokiego zakresu dodatkowych technik, które powszechnie nazywane są *programowaniem w stylu funkcyjnym*. Mówiąc w skrócie, tego rodzaju kod, w społeczności programowania funkcyjnego nazywany funkcjami, może być przekazywany i łączony w taki sposób, aby produkował potężne idiomy programistyczne, które będziemy mieli okazję zobaczyć w tej książce w odmianie dopasowanej do Javy 8.

Główna część tego rozdziału rozpoczyna się od ogólnej dyskusji na temat tego, dlaczego języki ewoluują. Kolejne podrozdziały omawiają podstawowe funkcje języka Java 8, po czym

* Wielordzeniowe procesory mają osobne pamięci cache (szybka pamięć) przyłączone do każdego rdzenia procesora. Blokowanie wymaga zsynchronizowania tych pamięci, co sprowadza się do względnie powolnej komunikacji protokołu spójności pamięci między rdzeniami.

prezentują koncepcje programowania funkcyjnego, które są przez te funkcje upraszczane, i które faworyzowane są przez nowe architektury komputerów. Krótko mówiąc, podrozdział 1.2 omawia proces ewolucji języka Java oraz pojęcia, których do tej pory w nim brakowało, by można było w nim w prosty sposób wykorzystywać równoległość wielordzeniową. Podrozdział 1.3 wyjaśnia, dlaczego przekazywanie kodu do metod w Javie 8 jest takim ważnym i potężnym idiomem programowania. W podobny sposób podrozdział 1.4 omawia strumienie będące nowym sposobem reprezentowania danych sekwencyjnych, wskazując przy tym, czy mogą być one przetwarzane równolegle. Podrozdział 1.5 wyjaśnia, w jaki sposób nowa funkcja Javy 8 w postaci metod domyślnych pozwala na bezproblemową rozbudowę interfejsów i ich bibliotek, z mniejszą ilością rekompilacji. Podrozdział ten objaśnia również koncepcję „modułów" języka Java 9, które umożliwiają nam bardziej przejrzyste określanie komponentów dużych systemów, niż tylko „zwykły plik JAR zawierający pakiety". W końcu, w podrozdziale 1.6, przyjrzymy się koncepcjom programowania funkcyjnego w Javie oraz w innych językach działających na maszynie wirtualnej Javy. Podsumowując, rozdział ten stanowi wprowadzenie do pojęć, które będą sukcesywnie omawiane w dalszej części tej książki. Życzymy udanej zabawy!

1.2 *Dlaczego Java ciągle się zmienia?*

W latach 60-tych ubiegłego wieku rozpoczęła się pogoń za idealnym językiem programowania. Peter Landin, znany wówczas informatyk, w swoim przełomowym artykule* z 1966 roku zwrócił uwagę na fakt, że do tej pory powstało już 700 języków programowania i rozpoczął spekulacje na temat tego, jak wyglądać będzie kolejnych 700 – nie zabrakło tu również argumentów przemawiających za funkcyjnym stylem programowania, zbliżonym do tego z języka Java 8.

Wiele tysięcy języków programowania później, środowiska akademickie zgodnie przyznały, że języki programowania zachowują się jak ekosystem: nieustannie pojawiają się nowe języki, a stare języki, jeśli nie ewoluują, są zastępowane. Wszyscy wyczekujemy tego idealnego uniwersalnego języka, ale w rzeczywistości pewne języki dopasowane są do konkretnych nisz lepiej od innych. Na przykład, języki C i C++, ze względu na ich niewielkie wymagania uruchomieniowe, nadal pozostają popularne w obszarze budowy systemów operacyjnych i różnych systemów wbudowanych, nawet pomimo braku obecności w nich mechanizmów bezpieczeństwa. Ten brak zabezpieczeń może powodować nieprzewidywalne awarie programów i wystawiać luki zabezpieczeń na działanie wirusów i innych podobnych zagrożeń. Istotnie, języki z zaimplementowanym bezpieczeństwem typu, takie jak Java i C#, w wielu aplikacjach, gdzie dodatkowy narzut uruchomieniowy jest akceptowalny, zastąpiły języki C i C++.

Wcześniejsze zdobycie niszy przez dany język zdaje się zniechęcać jego konkurentów. Przesiadka na nowy język i zestaw narzędzi jest często zbyt bolesna dla tylko jednej funkcji, ale nowe języki prędzej czy później wyprą te istniejące, chyba że będą one ewoluować na tyle szybko, by dotrzymać im kroku (starsi czytelnicy zapewne są w stanie podać

* P.J. Landin, „The Next 700 Programming Languages", CACM 9(3):157-65, marzec 1966

przykłady języków, w których kiedyś pisali programy, a których popularność od tego czasu zmalała – należą do nich języki Ada, Algol, COBOL, Pascal, Delphi i SNOBOL).

My jednak jesteśmy programistami Java, a język Java przez niemal 20 lat z sukcesem kolonizował (wypierając przy tym języki konkurencyjne) duży ekosystem niszowych zadań programistycznych. Przyjrzyjmy się kilku powodom, dlaczego tak się dzieje.

1.2.1 *Miejsce języka Java w ekosystemie języków programowania*

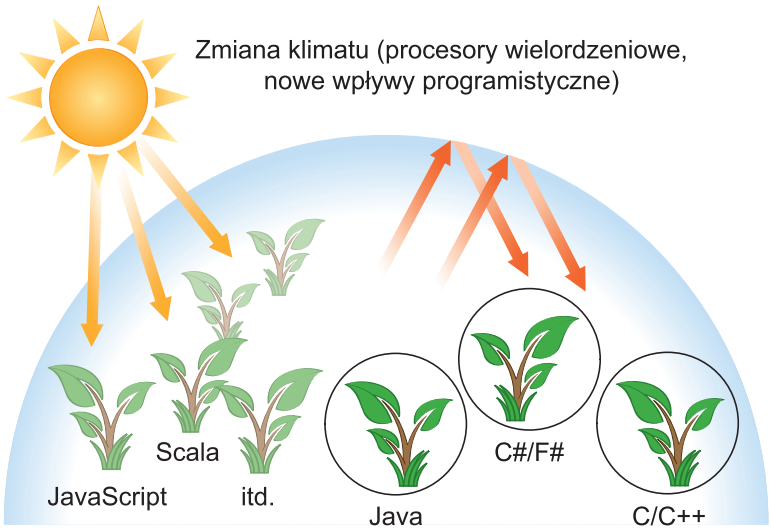
Java rozpoczęła swój żywot bardzo dobrze. Od samego początku był to dobrze zaprojektowany język zorientowany obiektowo z dużą liczbą przydatnych bibliotek. Od pierwszego dnia wspierał on współbieżność o niewielkiej skali, wykorzystując przy tym zintegrowane wsparcie dla wątków i blokad (mając już pełną świadomość tego, że współbieżne wątki na wielordzeniowych procesorach mogą mieć nieoczekiwane zachowania poza tymi, jakie występują na procesorach jednordzeniowych). Ponadto decyzja, aby kompilować kod Javy do kodu pośredniego JVM (kodu maszyny wirtualnej, który wkrótce rozumiała każda przeglądarka) sprawiła, że Java stała się językiem wyboru dla internetowych apletów (pamiętasz aplety?). Istotnie, istnieje ryzyko, że wirtualna maszyna Javy (JVM) i jej kod pośredni będzie postrzegana jako ważniejsza niż sam język Java, oraz że – przynajmniej w pewnych zastosowaniach – Java może zostać zastąpiona przez jednego z jej konkurentów, takich jak Scala, Groovy czy Kotlin, które również działają na maszynie wirtualnej Javy. Różne niedawne aktualizacje dedykowane maszynie JVM (na przykład nowy kod pośredni `invokedynamic` w JDK7) usprawniają działanie tych konkurencyjnych języków na maszynie wirtualnej Javy, jak również ich współdziałanie z językiem Java. Java zdołała również pomyślnie skolonizować różne aspekty przetwarzania w systemach wbudowanych (wliczając w to wszystko od inteligentnych kart, tosterów i dekoderów, aż po systemy hamowania w samochodach).

Jak Java trafiła do niszy programowania ogólnego?

Programowanie zorientowane obiektowo stało się modne w latach 90-tych ubiegłego wieku: jego zdyscyplinowana enkapsulacja skutkowała mniejszą ilością problemów inżynierii oprogramowania niż w przypadku języka C; zaś jako model mentalny z łatwością przechwycił on model programowania WIMP dla systemu Windows 95 i nowszych. Można to podsumować w następujący sposób: wszystko jest obiektem, zaś kliknięcie myszą wysyła komunikat zdarzenia do metody obsługującej (wywołuje metodę `clicked` w obiekcie `Mouse`). Model Javy „napisz raz, uruchamiaj wszędzie" oraz zdolność ówczesnych przeglądarek do (bezpiecznego) uruchamiania kodu apletów Java spopularyzował ją na uniwersytetach, których absolwenci zaczęli później wypełniać rynek programistów. Na początku istniała do niej pewna niechęć spowodowana dodatkowym kosztem uruchamiania kodu w stosunku do C/C++, ale z czasem komputery stawały się szybsze, a czas programisty coraz bardziej istotny. Język C# firmy Microsoft dodatkowo potwierdził słuszność wykorzystywania stylu zorientowanego obiektowo w języku Java.

Ale klimat ekosystemu języków programowania powoli się zmienia. Programiści coraz częściej mają do czynienia z pojęciem big data (zbiorami danych o rozmiarze sięgającym terabajtów) i coraz częściej pragną wykorzystywać wielordzeniowe komputery lub klastry komputerów do efektywnego przetwarzania tych danych. A to oznacza korzystanie z przetwarzania równoległego, dla którego Java nie była do tej pory zbyt przyjazna.

Być może natknąłeś się w swojej karierze na idee programowania pochodzące z języków programowania z innych nisz (np. map-reduce firmy Google lub względnie łatwe manipulowanie danymi za pomocą języków tworzenia zapytań baz danych, takich jak SQL), które pomagają Ci pracować z dużymi woluminami danych oraz wielordzeniowymi procesorami. Rysunek 1.1 podsumowuje ekosystem języków programowania: krajobraz należy postrzegać jako przestrzeń problemów programistycznych, zaś roślinność dominującą na danym fragmencie gleby jako ulubiony język dla tego problemu. Zmiana klimatu reprezentuje tu nowy sprzęt lub nowe wpływy programistyczne (np. „Dlaczego nie mogę programować w stylu SQL?"), które oznaczają, że różne języki stają się językami wyboru dla nowych projektów, tak jak zwiększanie regionalnych temperatur oznacza, że winogrona kwitną teraz na większych wysokościach. Nie brakuje oczywiście histerii – wielu starych farmerów nadal będzie uprawiać tradycyjne plony. Podsumowując, nowe języki pojawiają się i stają się coraz bardziej popularne, ponieważ szybko przystosowały się one do zmiany klimatu.



Rysunek 1.1 Ekosystem języków programowania i zmiana klimatu

Główną korzyścią dla programisty wynikającą z wprowadzonych do Javy 8 nowości jest to, że dostarczają one więcej narzędzi i koncepcji do szybszego rozwiązywania nowych i istniejących problemów programistycznych, i to niekiedy w bardziej zwięzły i łatwiejszy do utrzymania sposób. Choć koncepcje te są nowością w języku Java, to udowodniły one swą przydatność w językach niszowych, które często wykorzystywane są w badaniach. W książce tej nakreślamy i rozwijamy idee stojące za trzema takimi koncepcjami programistycznymi, które były motorem napędowym dla rozwoju funkcji Javy 8 mających na celu wykorzystanie współbieżności i zapewnienie możliwości pisania bardziej zwięzłego kodu. Zaprezentujemy je w nieco innej kolejności względem pozostałych rozdziałów tej książki,

aby umożliwić sobie przedstawienie analogii do systemu Unix oraz by wyjawić zależności typu „wymaga *tego* z powodu *tamtego*" nowej współbieżności języka Java 8 dla wielu rdzeni.

Kolejny czynnik zmiany klimatu dla Javy

Jeden z czynników zmiany klimatu dotyczy sposobu, w jaki duże systemy są projektowane. W dzisiejszych czasach powszechne jest, że duży system wykorzystuje duże podsystemy komponentów z innych projektów, a nie wykluczone, że i te zbudowane są na bazie innych komponentów od innych dostawców. Co gorsza, komponenty te i ich interfejsy również zwykle ewoluują. Java 8 i Java 9 rozwiązują ten problem oferując nam „metody domyślne" i „moduły", pozwalające na wykorzystanie takiego stylu projektowania.

1.2.2 *Przetwarzanie strumieni*

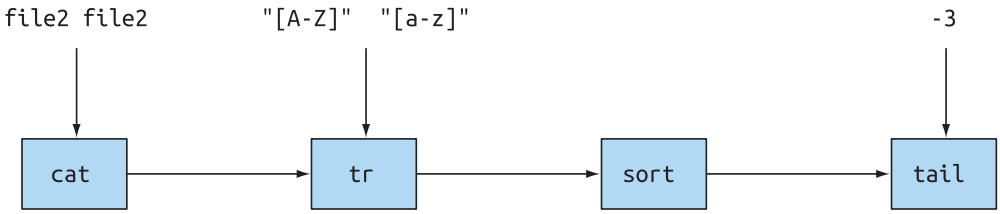
Pierwszą koncepcją programistyczną jest *przetwarzanie strumieni*. Na potrzeby tego wprowadzenia strumień jest sekwencją elementów danych, które produkowane są po jednym na raz – program może odczytywać elementy ze strumienia wejściowego jeden po drugim i w podobny sposób zapisywać elementy do strumienia wyjściowego. Strumień wyjściowy jednego programu może być z powodzeniem strumieniem wejściowym innego programu.

Praktycznym przykładem może tu być system uniksowy lub linuksowy, gdzie wiele programów odczytuje dane ze standardowego wejścia (`stdin` w Unix i C, `System.in` w Javie), wykonuje na nich operacje, a następnie zapisuje ich wyniki do standardowego wyjścia (`stdout` w Unix i C, `System.out` w Javie). Zacznijmy od nakreślenia tła. Polecenie uniksowe `cat` tworzy strumień poprzez skonkatenowanie dwóch plików, `tr` tłumaczy znaki w strumieniu, `sort` sortuje wiersze w strumieniu, zaś `tail -3` zwraca ostatnie trzy wiersze w strumieniu. Wiersz poleceń systemu Unix umożliwia łączenie ze sobą takich programów za pomocą operatora potoku (ang. *pipe, (*|)) przykładowo w poniższy sposób:

```
cat file1 file2  |  tr "[A-Z]"  "[a-z]"  |  sort  |  tail -3
```

co (przy założeniu, że pliki `file1` i `file2` zawierają pojedyncze słowo w każdym wierszu) wypisze trzy słowa z plików, które w porządku alfabetycznym znajdują się na końcu, po tym jak zostaną one najpierw przekształcone na małe znaki. Mówimy, że komenda `sort` przyjmuje strumień wierszy* jako wejście i produkuje jako wyjście kolejny strumień wierszy (tym razem posortowany), jak zostało to przedstawione na rysunku 1.2. Zwróćmy uwagę, że w systemie Unix polecenia (`cat`, `tr`, `sort` i `tail`) wykonywane są współbieżnie, dzięki czemu `sort` może zacząć przetwarzać pierwszych kilka wierszy, zanim jeszcze `cat` lub `tr` zakończą swoje działanie. Bardziej mechaniczną analogią może tu być linia produkcyjna w fabryce samochodów, gdzie strumień samochodów ustawiany jest przed każdą stacją przetwarzania. Każda taka stacja przyjmuje samochód, modyfikuje go i przekazuje do następnej stacji w celu dalszej obróbki; przetwarzanie w oddzielnych stacjach jest zazwyczaj współbieżne, mimo że linia produkcyjna fizycznie jest sekwencją.

* Puryści powiedzą „strumień znaków", ale koncepcyjnie łatwiej jest postrzegać metodę `sort` jako zmieniającą kolejność wierszy.



Rysunek 1.2 Polecenia systemu Unix operujące na strumieniu

Bazując na tej idei, język Java 8 wprowadza interfejs programowania aplikacji (ang. *application programming interface, API*) o nazwie Streams (zwróć uwagę na dużą literę S), dostępny w pakiecie `java.util.stream`. Strumień `Stream<T>` jest sekwencją elementów typu `T`. Na tę chwilę możesz myśleć o nim jak o wymyślnym iteratorze. API strumieni zawiera wiele metod, które mogą być ze sobą łączone w celu uformowania pojedynczego złożonego przepływu operacji, dokładnie tak, jak polecenia systemu Unix zostały połączone w poprzednim przykładzie.

Dzięki takiemu podejściu możemy teraz programować w Javie 8 na wyższym poziomie abstrakcji, strukturyzując nasze myśli na temat zamiany strumienia jednego rodzaju na strumień innego rodzaju (podobnie jak myślimy podczas tworzenia zapytań bazy danych) zamiast operować na elementach jeden po drugim. Kolejną zaletą jest to, że Java 8 może transparentnie wykonywać takie przepływy operacji w kilku rdzeniach procesora na rozłącznych fragmentach wejścia – w ten sposób, zamiast ręcznie posługiwać się wątkami, otrzymujemy współbieżność *niemal za darmo*. API strumieni Javy 8 omówimy szczegółowo w rozdziałach 4-7.

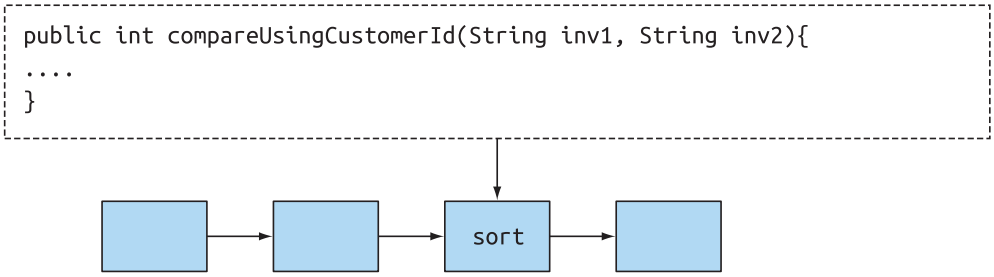
1.2.3 *Przekazywanie kodu do metod za pomocą parametryzacji zachowania*

Drugą koncepcją programistyczną wprowadzoną do języka Java 8 jest możliwość przekazania fragmentu kodu do interfejsu programowania aplikacji. Brzmi to nazbyt abstrakcyjnie. Wracając do naszego przykładu z systemem Unix, możemy poinformować polecenie `sort`, że chcemy skorzystać z niestandardowej metody porządkowania. Choć polecenie `sort` obsługuje parametry wiersza poleceń do wykonywania różnych predefiniowanych sortowań, takich jak sortowanie odwrócone, to jednak są one dosyć ograniczone.

Załóżmy przykładowo, że mamy kolekcję identyfikatorów faktur o formacie zbliżonym do 2013UK0001, 2014US0002, itd. Pierwsze cztery cyfry reprezentują rok, kolejne dwie litery reprezentują kod kraju, zaś ostatnie cztery cyfry to identyfikator klienta. Możemy zechcieć posortować te identyfikatory według roku, identyfikatora klienta czy kodu kraju. Chcemy więc przekazać do polecenia `sort` w formie argumentu porządkowanie zdefiniowane przez użytkownika: oddzielny fragment kodu przekazywany do polecenia `sort`.

Teraz, przekładając to bezpośrednio na język Java, chcemy poprosić metodę `sort` o porównywanie za pomocą niestandardowego porządku. Moglibyśmy napisać metodę `compareUsingCustomerId` do porównywania dwóch identyfikatorów faktur, ale przed wydaniem Javy 8 nie moglibyśmy przekazać jej do innej metody! Moglibyśmy utworzyć obiekt `Comparator` do przekazania do metody `sort`, jak to pokazaliśmy na początku tego rozdziału, ale ten sposób jest dosyć rozwlekły i utrudnia ponowne wykorzystanie istniejącego fragmentu zachowania. Java 8 wprowadza możliwość przekazywania metod (naszego kodu)

jako argumentów do innych metod. Ideę tę ilustruje rysunek 1.3, bazujący na rysunku 1.2. Koncepcję tę określamy mianem *parametryzacji zachowania*. Dlaczego jest to istotne? API strumieni oparte jest na idei przekazywania kodu w celu sparametryzowania zachowania jego operacji, jak w przypadku przekazania metody `compareUsingCustomerId` w celu sparametryzowania zachowania metody `sort`.



Rysunek 1.3 Przekazanie metody `compareUsingCustomerId` jako argumentu do metody `sort`

Sposób działania tego mechanizmu podsumujemy w podrozdziale 1.3, ale w pełni szczegółowo zajmiemy się nim w rozdziałach 2 i 3. Rozdziały 18 i 19 omawiają bardziej zaawansowane rzeczy, jakie możemy osiągnąć za pomocą tej funkcji, w tym również techniki oferowane przez społeczność *programowania funkcyjnego*.

1.2.4 *Równoległość i współdzielone dane modyfikowalne*

Trzecia koncepcja programistyczna jest nieco mniej jawna i wywodzi się z frazy „współbieżność niemal za darmo" z naszej poprzedniej dyskusji na temat przetwarzania strumieni. Co musimy poświęcić? Może zajść potrzeba dokonania pewnych małych zmian w sposobie, w jaki kodujemy zachowanie przekazywane do metod strumienia. Na pierwszy rzut oka zmiany te mogą wydawać się nieco niekomfortowe, ale gdy się już do nich przyzwyczaimy, pokochamy je. Musimy dostarczyć zachowanie, które jest bezpieczne pod względem wykonywania współbieżnego na różnych fragmentach wejścia. Zwykle oznacza to pisanie kodu, który nie uzyskuje dostępu do współdzielonych danych modyfikowalnych w celu wykonania swojej pracy. Pisany w ten sposób kod określamy mianem czystych funkcji, funkcji bezstanowych lub funkcji pozbawionych efektów ubocznych – omówimy je szczegółowo w rozdziałach 18 i 19. Poprzednia współbieżność możliwa jest tylko dzięki założeniu, że wiele kopii naszego fragmentu kodu może pracować niezależnie. Jeśli istnieje jakaś współdzielona zmienna lub obiekt, do którego zapisujemy, wówczas rzeczy przestają działać: co, jeśli dwa procesy chcą zmodyfikować współdzieloną zmienną w tym samym czasie? (podrozdział 1.4 dostarcza na ten temat bardziej szczegółowe wyjaśnienie z diagramem). Więcej na temat tego stylu dowiesz się w dalszej części tej książki.

Strumienie Javy 8 wykorzystują równoległość w prostszy sposób niż istniejące API wątków języka Java, tak więc choć *możliwe jest* wykorzystanie słowa kluczowego `synchronized` do złamania reguły „braku współdzielonych danych modyfikowalnych", będzie to po prostu nadużycie abstrakcji zoptymalizowanej wokół tej reguły. Posługiwanie się konstrukcją `synchronized` w obrębie wielu przetwarzających rdzeni jest zwykle znacznie bardziej kosztowne niż się tego spodziewamy, ponieważ synchronizacja zmusza kod do wykonywania sekwencyjnego, co stoi w sprzeczności z celem wykonywania równoległego.

Dwa z tych punktów (brak współdzielonych danych modyfikowalnych oraz zdolność do przekazywania metod i funkcji – kodu – do innych metod) stanowią podstawę tego, co nazywamy *paradygmatem programowania funkcyjnego*, z którym zapoznamy się szczegółowo w rozdziałach 18 i 19. Z kolei w paradygmacie programowania imperatywnego zazwyczaj opisujemy program w kontekście sekwencji instrukcji, które modyfikują stan programu. Wymaganie braku współdzielonych danych modyfikowalnych oznacza, że metoda jest perfekcyjnie opisana wyłącznie przez sposób, w jaki przekształca swoje argumenty na wyniki. Innymi słowy, zachowuje się ona jak funkcja matematyczna i nie ma (widocznych) efektów ubocznych.

1.2.5 *Java musi ewoluować*

Z pewnością zaobserwowałeś już wcześniej proces ewolucji w Javie. Na przykład, wprowadzenie generyków i stosowanie konstrukcji `List<String>` zamiast po prostu `List` mogło być początkowo irytujące. Ale teraz jesteś już zaznajomiony z tym stylem i oferowanymi przez niego korzyściami (wyłapywanie większej liczby błędów na etapie kompilacji i tworzenie bardziej czytelnego kodu, ponieważ wiadomo teraz, jakiego rodzaju jest to lista).

Inne zmiany pozwoliły na łatwiejsze wyrażanie niektórych rzeczy, np. korzystanie z pętli `foreach` zamiast eksponowania zbędnego kodu w przypadku użycia iteratora. Główne zmiany dokonane w języku Java 8 odzwierciedlają odejście od klasycznej orientacji obiektowej, która często skupia się na modyfikowaniu istniejących wartości, i przesunięcie w kierunku spektrum programowania funkcyjnego, w którym to, co chcemy osiągnąć (np. utworzyć wartość reprezentującą wszystkie drogi transportowe z punktu A do punktu B o koszcie mniejszym od podanego) jest dużo ważniejsze i odseparowane od tego, w jaki sposób możemy to osiągnąć (np. przeskanować strukturę danych modyfikując pewne komponenty). Zwróćmy uwagę, że klasyczne programowanie zorientowane obiektowo i programowanie funkcyjne, jako dwie różne skrajności, mogą wydawać się ze sobą sprzeczne. Idea polega na tym, by z obu tych paradygmatów programowania wyciągnąć to, co w nich najlepsze, abyśmy mieli większą szansę na uzyskanie właściwego narzędzia pracy! Omówimy to bardziej szczegółowo w podrozdziałach 1.3 i 1.4.

Najważniejszym wnioskiem może być to: języki muszą ewoluować, by podążać za zmieniającym się sprzętem lub oczekiwaniami programistów (jeśli potrzebujesz dowodu, to weź pod uwagę, że COBOL był kiedyś jednym z najważniejszych komercyjnych języków programowania). Aby przetrwać, Java musi ewoluować poprzez wprowadzanie nowych funkcji. Ewolucja ta będzie jednak bezwartościowa, jeśli funkcje te nie będą wykorzystywane, tak więc używając języka Java 8 chronisz swoje życie programisty Java. Na dodatek mamy przeczucie, że pokochasz korzystanie z nowych funkcji Javy 8. Zapytaj dowolną osobę, która korzystała z Javy 8, czy chce wrócić do poprzedniej wersji! Co więcej, nowe funkcje Javy 8 mogą, w analogii do naszego ekosystemu, umożliwić Javie podbój terytoriów zadań programistycznych okupowanych obecnie przez inne języki, a wtedy programiści Javy 8 będą jeszcze bardziej rozchwytywani.

Jeden po drugim zaprezentujemy teraz nowe pojęcia wprowadzone do języka Java 8, podając przy tym rozdziały, które omawiają je bardziej szczegółowo.

1.3 *Funkcje w Javie*

Słowa funkcja w językach programowania używa się powszechnie jako synonimu dla metody, w szczególności dla metody statycznej; ponadto używa się go do określania funkcji matematycznych, czyli funkcji pozbawionych efektów ubocznych. Na szczęście, jak za chwilę zobaczymy, gdy Java 8 odnosi się do funkcji, ich znaczenie i sposób wykorzystania są bardzo podobne.

Java 8 wprowadza funkcje jako nową formę wartości. Wykorzystują one strumienie, omówione w podrozdziale 1.4, które Java 8 dostarcza w celu wykorzystania programowania równoległego na procesorach wielordzeniowych. Rozpoczniemy od pokazania, że funkcje jako wartości same w sobie mogą być bardzo przydatne.

Rozważmy możliwe wartości, jakimi możemy manipulować w programach pisanych w Javie. Po pierwsze, dostępne są wartości prymitywne, takie jak 42 (typu `int`) i 3.14 (typu `double`). Po drugie, wartości mogą być obiektami (ściślej mówiąc, referencjami do obiektów). Jedynym sposobem uzyskania takiego obiektu jest posłużenie się słowem kluczowym `new`, przykładowo za pośrednictwem metody wytwórczej lub funkcji bibliotecznej; referencje do obiektów wskazują na instancje klasy. Przykładami mogą być `"abc"` (typu `String`), `new Integer(1111)` (typu `Integer`) oraz rezultat `new HashMap<Integer, String>(100)` jawnego wywołania konstruktora klasy `HashMap`. Nawet tablice są obiektami. Na czym więc polega problem?

Aby odpowiedzieć na to pytanie, trzeba najpierw podkreślić, że głównym zadaniem języka programowania jest manipulowanie wartościami, które – podążając za historyczną tradycją języków programowania – nazywane są z tego powodu wartościami pierwszej kategorii (lub obywatelami pierwszej kategorii, w terminologii zapożyczonej z ruchu praw obywatelskich w USA z lat 60-tych dwudziestego wieku). Inne struktury w naszych językach programowania, które przykładowo pomagają nam wyrazić strukturę wartości, ale które nie mogą być przekazywane w czasie wykonywania programu, są obywatelami drugiej kategorii. Wartości wylistowane powyżej są w języku Java obywatelami pierwszej kategorii, ale różne inne koncepcje tego języka, takie jak metody i klasy, stanowią przykład obywateli drugiej kategorii. Metody przydają się do definiowania klas, na podstawie których możemy z kolei tworzyć instancje w celu wyprodukowania konkretnych wartości, ale ani metody, ani klasy same w sobie nie są wartościami. Czy ma to jakieś znaczenie? Tak, okazuje się, że możliwość przekazywania metod w czasie wykonywania programu, a tym samym uznawanie ich za obywateli pierwszej kategorii, jest w programowaniu bardzo przydatne, tak więc projektanci języka Java 8 wprowadzili możliwość wyrażania tego bezpośrednio w Javie. Być może zastanawiasz się, czy traktowanie innych obywateli drugiej kategorii (takich jak klasy) jako wartości pierwszej kategorii byłoby dobrym pomysłem. Różne języki, takie jak Smalltalk i JavaScript, poszły właśnie tą drogą.

1.3.1 *Metody i wyrażenia lambda jako obywatele pierwszej kategorii*

Eksperymenty w językach, takich jak Scala czy Groovy, pokazały, że umożliwienie wykorzystywania metod jako wartości pierwszej kategorii ułatwia programowanie poprzez rozszerzenie zestawu narzędzi dostępnych dla programistów. Jak tylko programiści zaznajomią się z tak potężną funkcją, nie chcą oni używać języka bez niej! Tak więc projektanci Javy 8

zdecydowali się zezwolić na traktowanie metod jako wartości, by w ten sposób ułatwić nam programowanie. Co więcej, funkcjonalność Javy 8 traktująca metody jak wartości tworzy podstawę dla różnych innych funkcji tego języka (takich jak strumienie).

Pierwszą z nowych funkcji Javy 8, które zaprezentujemy, będą *referencje do metod*. Załóżmy, że chcemy przefiltrować wszystkie ukryte pliki w katalogu. Musimy rozpocząć od napisania metody, która po podaniu pliku (`File`) poinformuje nas, czy plik ten jest ukryty, czy też nie. Na szczęście istnieje już taka metoda wewnątrz klasy `File` o nazwie `isHidden`. Metodę tę możemy postrzegać jako funkcję, która przyjmuje plik i zwraca wartość logiczną `boolean`. Aby jednak móc wykorzystać ją do filtrowania, musimy opakować ją w obiekt `FileFilter`, który następnie przekazujemy do metody `File.listFiles`, jak w poniższym przykładzie:

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
     public boolean accept(File file) {
         return file.isHidden();                                    (1)
     }
});
```

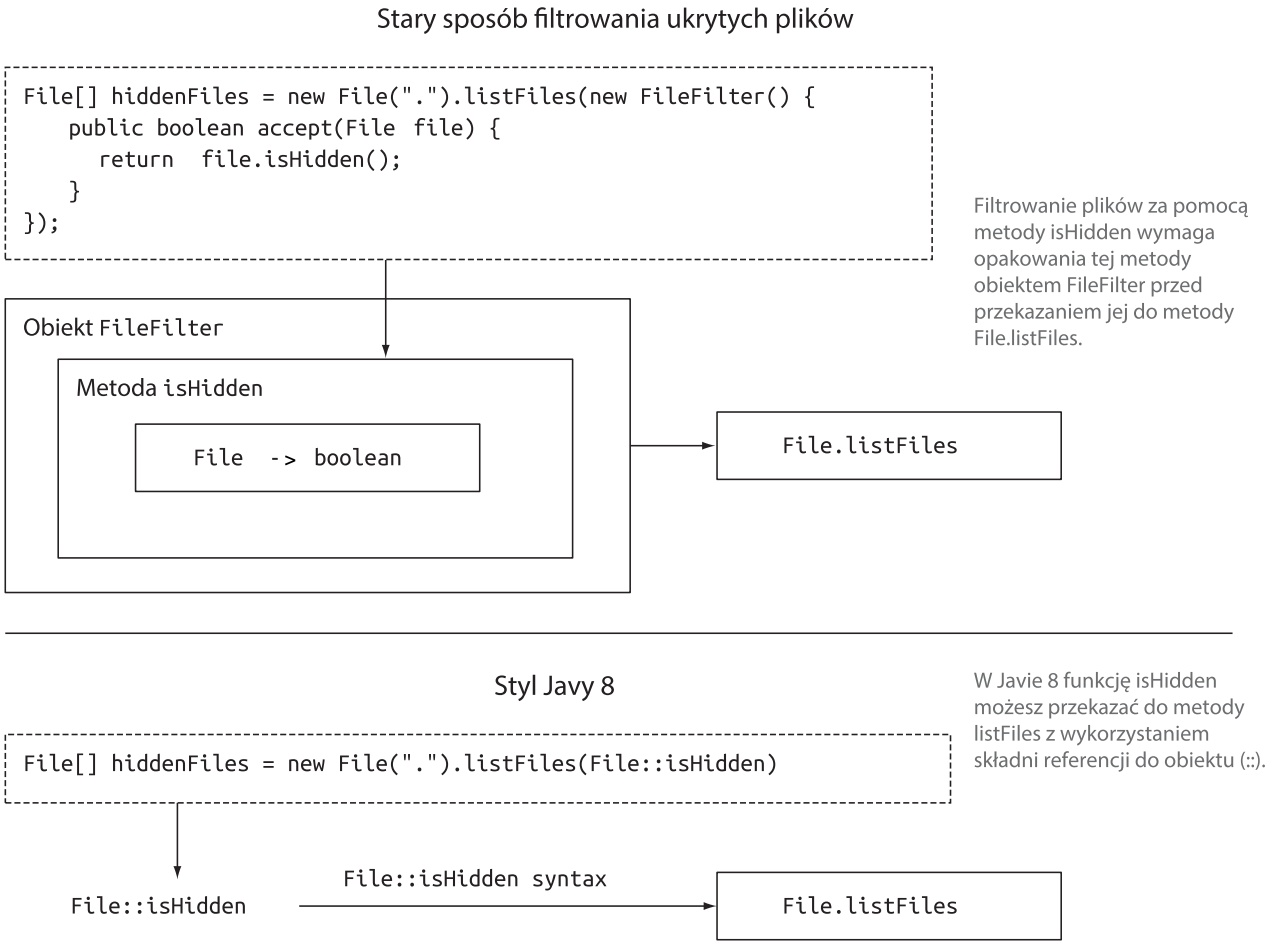
(1) Filtrowanie ukrytych plików!

Fuj! To wygląda okropnie! Choć to tylko trzy wiersze kodu, to są to trzy nieprzejrzyste wiersze – wszyscy przy pierwszym podejściu zadajemy sobie pytanie „Czy naprawdę muszę to robić w ten sposób?”. Mamy już metodę `isHidden`, którą moglibyśmy wykorzystać. Dlaczego musimy opakowywać ją dodatkową klasą `FileFilter`, a następnie tworzyć jej instancję? Ponieważ to właśnie musielibyśmy zrobić w starszych wersjach języka Java!

Teraz, w Javie 8, możemy przepisać ten kod w następujący sposób:

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

Wow! Czy to nie wspaniałe? Mamy już dostępną funkcję `isHidden`, więc po prostu przekazujemy ją do metody `listFiles` przy wykorzystaniu składni *referencji do metody* `::` (co oznacza „użyj tej metody jako wartości”); zwróćmy uwagę, że zaczęliśmy również wykorzystywać słowo funkcja w stosunku do metod. Później wyjaśnimy, w jaki sposób to działa. Niewątpliwą zaletą takiego podejścia jest to, że nasz kod dużo bardziej odpowiada teraz definicji rozważanego problemu. Oto przedsmak tego, co nas czeka: metody nie są już dłużej wartościami drugiej kategorii. Analogicznie do wykorzystywania referencji do obiektu podczas przekazywania obiektów (a referencje do obiektów tworzone są konstrukcją `new`), w Javie 8, gdy napiszemy `File::isHidden`, tworzymy referencję do metody, która może być przekazywana w podobny sposób. Koncepcja ta omówiona zostanie szczegółowo w rozdziale 3. Biorąc pod uwagę, że metody zawierają kod (wykonywalne ciało metody), to wykorzystanie referencji do metod umożliwia nam przekazywanie kodu jak na rysunku 1.3. Rysunek 1.4 ilustruje tę koncepcję. W kolejnym podrozdziale zobaczymy również konkretny przykład (dotyczący wybierania jabłek z inwentarza).



Rysunek 1.4 Przekazywanie do metody listFiles referencji do metody File::isHidden

Wyrażenia lambda – funkcje anonimowe

Java 8 nie tylko zezwala na traktowanie (nazwanych) metod jak wartości pierwszej kategorii, ale też wprowadza szerszą ideę *funkcji jako wartości*, wliczając w to *wyrażenia lambda** (lub funkcje anonimowe). Na przykład, możemy teraz napisać `(int x) -> x + 1`, aby zdefiniować „funkcję, która w przypadku wywołania z argumentem x zwróci wartość x + 1". Zapewne zastanawiasz się, dlaczego jest to konieczne. W końcu moglibyśmy zdefiniować metodę `add1` wewnątrz klasy `MyMathUtils`, a potem napisać `MyMathUtils::add1`! Tak, moglibyśmy, ale nowa składnia wyrażeń lambda jest bardziej zwięzła dla przypadków, w których nie dysponujemy tak wygodną metodą i klasą. Wyrażenia lambda omówione zostaną szczegółowo w rozdziale 3. Programy wykorzystujące te koncepcje określane są mianem programów pisanych funkcyjnym stylem programowania. Fraza ta oznacza „pisanie programów, które przekazują funkcje jako wartości pierwszej kategorii".

* Nazwa pochodzi od greckiej litery λ (lambda). Choć symbol ten nie jest używany w Javie, to jego nazwa jest powszechnie stosowana.

1.3.2 *Przekazywanie kodu: przykład*

Spójrzmy na przykład (omówiony bardziej szczegółowo w rozdziale 2, „Przekazywanie kodu z parametryzacją zachowania") dotyczący tego, w jaki sposób pomoże nam to pisać programy. Cały kod dla prezentowanych tu przykładów dostępny jest na stronie GitHub tej książki (*https://github.com/java-manning/modern-java*). Załóżmy, że mamy klasę `Apple` z metodą `getColor` oraz zmienną `inventory` przechowującą listę jabłek (listę obiektów `Apple`). W takim wypadku możemy zechcieć wybrać z listy wszystkie zielone jabłka (wykorzystując typ wyliczeniowy `Color` zawierający wartości `GREEN` i `RED`) i zwrócić je w postaci listy. Zwykle do wyrażenia tej koncepcji używamy czasownika *przefiltrować* (ang. *filter*). Aby to zrobić, przed wydaniem języka Java 8 moglibyśmy przykładowo zdefiniować metodę `filterGreenApples`:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
     List<Apple> result = new ArrayList<>();                          (1)
     for (Apple apple: inventory){
          if (GREEN.equals(apple.getColor())) {                       (2)
               result.add(apple);
          }
     }
     return result;
}
```

(1) Lista result akumuluje w sobie wynik; na początku lista ta jest pusta, po czym zielone jabłka dodawane są do niej jedno po drugim

(2) Wytłuszczony tekst wybiera wyłącznie zielone jabłka

Później jednak ktoś potrzebowałby listy ciężkich jabłek (powiedzmy o wadze większej niż 150 g), tak więc, z ciężkim sercem, napisalibyśmy poniższą metodę (chociażby z wykorzystaniem metody kopiuj-wklej):

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {                                (1)
            result.add(apple);
        }
    }
    return result;
}
```

(1) Tutaj wytłuszczony tekst wybiera jedynie ciężkie jabłka

Wszyscy wiemy, jakie zagrożenia w inżynierii oprogramowania niesie ze sobą kopiowanie i wklejanie (np. aktualizacje i poprawki dla jednego wariantu, ale dla drugiego już nie), a do tego te dwie metody różnią się wyłącznie jednym wierszem: wytłuszczonym warunkiem wewnątrz konstrukcji `if`. Gdyby różnica pomiędzy tymi dwoma wywołaniami metod

sprowadzała się po prostu do tego, jaki zakres wag jest dopuszczalny, moglibyśmy wówczas zwyczajnie przekazać dolną i górną granicę wagi jako argumenty do metody `filter` – przykładowo (`150`, `1000`) w celu wybrania ciężkich jabłek (ponad 150 g), lub (`0`, `80`) w celu wybrania lekkich jabłek (poniżej 80 g).

Ale jak już wspomnieliśmy wcześniej, Java 8 umożliwia nam przekazywanie kodu warunku w formie argumentu, co pozwala nam uniknąć zdublowania kodu metody `filter`. Teraz możemy napisać:

```
public static boolean isGreenApple(Apple apple) {
     return GREEN.equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
     return apple.getWeight() > 150;
}
public interface Predicate<T>{                                          (1)
       boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {                   (2)
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {                                            (3)
            result.add(apple);
        }
    }
    return result;
}
```

(1) Zawarte w celu zwiększenia czytelności (normalnie importowane z pakietu java.util.function)

(2) Metoda przekazywana jest jako parametr Predicate o nazwie p (zobacz wzmiankę „Czym jest predykat?")

(3) Czy jabłko pasuje do warunku reprezentowanego przez p?

Aby z tego skorzystać, wywołujemy albo

```
filterApples(inventory, Apple::isGreenApple);
```

albo

```
filterApples(inventory, Apple::isHeavyApple);
```

W kolejnych dwóch rozdziałach wyjaśnimy szczegółowo, w jaki sposób to działa. Najważniejszym wnioskiem do wyciągnięcia z tego jest to, że w języku Java 8 możemy przekazywać metody!

Czym jest predykat?

Poprzedni kod przekazał metodę `Apple::isGreenApple` (która przyjmuje jako argument jabłko `Apple` i zwraca wartość logiczną `boolean`) do metody `filterApples`, która oczekiwała parametru `Predicate<Apple>`. Słowo predykat jest często używane w matematyce do oznaczenia czegoś na wzór funkcji, która przyjmuje pewną wartość dla argumentu i zwraca prawdę (`true`) lub fałsz (`false`). Jak zobaczysz później, Java 8 pozwoliłaby również napisać `Function<Apple, Boolean>`, co będzie bardziej zrozumiałe dla czytelników, którzy w szkole poznali funkcje, ale nie predykaty – ale korzystanie z `Predicate<Apple>` jest bardziej standardowe (i odrobinę bardziej wydajne, ponieważ rozwiązanie to unika opakowywania (ang. *boxing*) typu `boolean` w typ `Boolean`).

1.3.3 *Od przekazywania metod do wyrażeń lambda*

Przekazywanie metod jako wartości jest oczywiście przydatne, ale konieczność pisania definicji dla krótkich metod, takich jak `isHeavyApple` i `isGreenApple`, gdy są one wykorzystywane co najwyżej dwa razy, może być irytujące. Na szczęście Java 8 rozwiązuje również i ten problem. Wprowadza ona nową notację (anonimowe funkcje lub wyrażenia lambda), dzięki której możemy napisać po prostu

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
```

lub

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

lub nawet

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     RED.equals(a.getColor()) );
```

Tak więc dla metody, która wykorzystywana jest tylko jeden raz, nie musimy nawet pisać definicji; taki kod jest znacznie bardziej przejrzysty, ponieważ nie musimy wyszukiwać fragmentu kodu, który chcemy przekazać. Jeśli jednak długość takiego wyrażenia lambda przekracza kilka wierszy (przez co jego zachowanie nie jest natychmiastowo rozpoznawalne), wówczas powinniśmy użyć referencji do metody z jakąś opisową nazwą zamiast anonimowego wyrażenia lambda. Przejrzystość kodu powinna być na pierwszym miejscu.

Projektanci Javy 8 mogli się tu w zasadzie zatrzymać, i być może zrobiliby to, gdyby nie wielordzeniowe procesory! Programowanie funkcyjne, jak je przedstawiliśmy do tej pory, okazuje się być bardzo potężne, o czym się wkrótce przekonamy. Java mogłaby wówczas zostać wyposażona w metodę `filter` oraz w kilku przyjaciół w postaci generycznych metod bibliotecznych, takich jak

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

Wtedy nie musielibyśmy nawet tworzyć takich metod, jak `filterApples`, ponieważ przykładowo poprzednie wywołanie

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

moglibyśmy wówczas zapisać jako wywołanie metody bibliotecznej `filter`:

```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

Jednak z powodów związanych z chęcią lepszego wykorzystania równoległego przetwarzania projektanci postąpili inaczej. Zamiast tego Java 8 zawiera rozbudowany i przypominający kolekcje interfejs programowania aplikacji o nazwie Streams (Strumienie). API strumieni zawiera wyczerpujący zbiór operacji zbliżonych do metody `filter`, z jakimi programiści funkcyjni mogą być już zaznajomieni (np. `map` lub `reduce`), wraz z metodami do konwertowania pomiędzy kolekcjami i strumieniami, o których teraz powiemy sobie więcej.

1.4 *Strumienie*

Prawie każda aplikacja napisana w języku Java tworzy i przetwarza kolekcje. Jednak praca z kolekcjami nie zawsze jest bezproblemowa. Załóżmy przykładowo, że chcemy przefiltrować z listy tylko drogie transakcje, a następnie pogrupować je według walut. W celu zaimplementowania takiego zapytania przetwarzania danych będziemy musieli napisać sporo zbędnego kodu, jak w poniższym przykładzie:

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();                                                    (1)
for (Transaction transaction : transactions) {                          (2)
    if(transaction.getPrice() > 1000){                                  (3)
        Currency currency = transaction.getCurrency();                  (4)
        List<Transaction> transactionsForCurrency =
            transactionsByCurrencies.get(currency);
        if (transactionsForCurrency == null) {                          (5)
            transactionsForCurrency = new ArrayList<>();
            transactionsByCurrencies.put(currency,
                                         transactionsForCurrency);
        }
        transactionsForCurrency.add(transaction);                       (6)
    }
}
```

(1) Tworzy mapę, w której zbierane będą pogrupowane transakcje
(2) Przechodzi po elementach listy transakcji
(3) Filtruje drogie transakcje
(4) Wyodrębnia walutę transakcji
(5) Jeśli w mapie grupującej nie ma wpisu dla tej waluty, utwórz go
(6) Dodaj aktualnie przetwarzaną transakcję do listy transakcji z tą samą walutą

Dodatkowo ciężko jest na początku zrozumieć działanie takiego kodu, ponieważ zawiera on wiele zagnieżdżonych instrukcji kontroli przepływu. Korzystając z API strumieni możemy rozwiązać ten problem w następujący sposób:

```
import static java.util.stream.Collectors.groupingBy;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
                .filter((Transaction t) -> t.getPrice() > 1000)          (1)
                .collect(groupingBy(Transaction::getCurrency));          (2)
```

(1) Filtruje drogie transakcje
(2) Grupuje je według walut

Nie martw się na razie tym kodem, gdyż może on dla Ciebie wyglądać dosyć magicznie. Korzystanie z API strumieni zostanie omówione w rozdziałach 4-7. Na tym etapie warto zauważyć, że API strumieni dostarcza kompletnie inny sposób przetwarzania danych w porównaniu do API kolekcji. Korzystając z kolekcji sami zarządzamy procesem iteracji. Po kolejnych elementach musimy iterować jeden po drugim za pomocą pętli `foreach`, a następnie przetwarzać te elementy. Ten sposób iterowania po danych nazywamy *iteracją zewnętrzną*. Z kolei korzystając z API strumieni nie musimy wcale przejmować się pętlami. Przetwarzanie danych odbywa się wewnątrz biblioteki, zaś ten sposób iterowania nazywamy *iterowaniem wewnętrznym*. Do pojęć tych powrócimy w rozdziale 4.

Aby zrozumieć kolejny problem, jaki niesie ze sobą praca z kolekcjami, wyobraźmy sobie, w jaki sposób moglibyśmy przetworzyć listę transakcji w przypadku, gdyby transakcji tych było bardzo wiele. W jaki sposób należałoby przetworzyć tak dużą listę? Pojedynczy procesor nie byłby w stanie przetworzyć tak dużej ilości danych, ale każdy z nas dysponuje zapewne wielordzeniowym komputerem. W idealnym przypadku staralibyśmy się podzielić tę pracę na poszczególne rdzenie procesora dostępne w naszym komputerze, by zredukować czas potrzebny na przetworzenie wszystkich danych. W teorii, jeśli mamy osiem rdzeni, powinny być one w stanie przetworzyć nasze dane osiem razy szybciej niż w przypadku pojedynczego rdzenia*, ponieważ będą one pracować równolegle.

Wielordzeniowe komputery

Wszystkie nowe komputery stacjonarne i laptopy są komputerami wielordzeniowymi. Zamiast pojedynczego procesora są one wyposażone w cztery, osiem lub więcej jednostek obliczeniowych (nazywanych zwykle rdzeniami). Problem polega na tym, że klasyczne programy Java wykorzystują tylko jeden z tych rdzeni, zaś moc pozostałych rdzeni jest marnowana. W podobny sposób wiele przedsiębiorstw do wydajnego przetwarzania dużych ilości danych wykorzystuje klastry obliczeniowe (komputery

* Nazewnictwo jest w pewnym stopniu niefortunne. Każdy z rdzeni w układzie wielordzeniowym jest tak naprawdę pełnoprawnym procesorem. Jednak fraza „wielordzeniowy procesor" upowszechniła się na tyle, że terminu rdzeń używamy dziś do określenia poszczególnych procesorów.

połączone ze sobą w szybkie sieci). Java 8 oferuje nowe style programowania, aby móc lepiej wykorzystać takie komputery.

Przykładem kodu, który jest zbyt duży, by mógł działać na pojedynczym komputerze jest silnik wyszukiwarki Google. Silnik ten skanuje każdą stronę w Internecie i tworzy w ten sposób indeks, w ramach którego każde słowo pojawiające się na dowolnej stronie internetowej mapowane jest do każdego adresu URL zawierającego to słowo. Później, gdy dokonujemy wyszukiwania w Google zawierającego kilka słów, oprogramowanie może skorzystać z tego indeksu, aby dać nam zbiór witryn internetowych, które zawierają to słowo. Spróbuj sobie wyobrazić, jak mógłbyś zakodować ten algorytm w języku Java (nawet dla znacznie mniejszego indeksu musiałbyś wykorzystać moc wszystkich rdzeni swojego komputera).

1.4.1 *Wielowątkowość jest trudna*

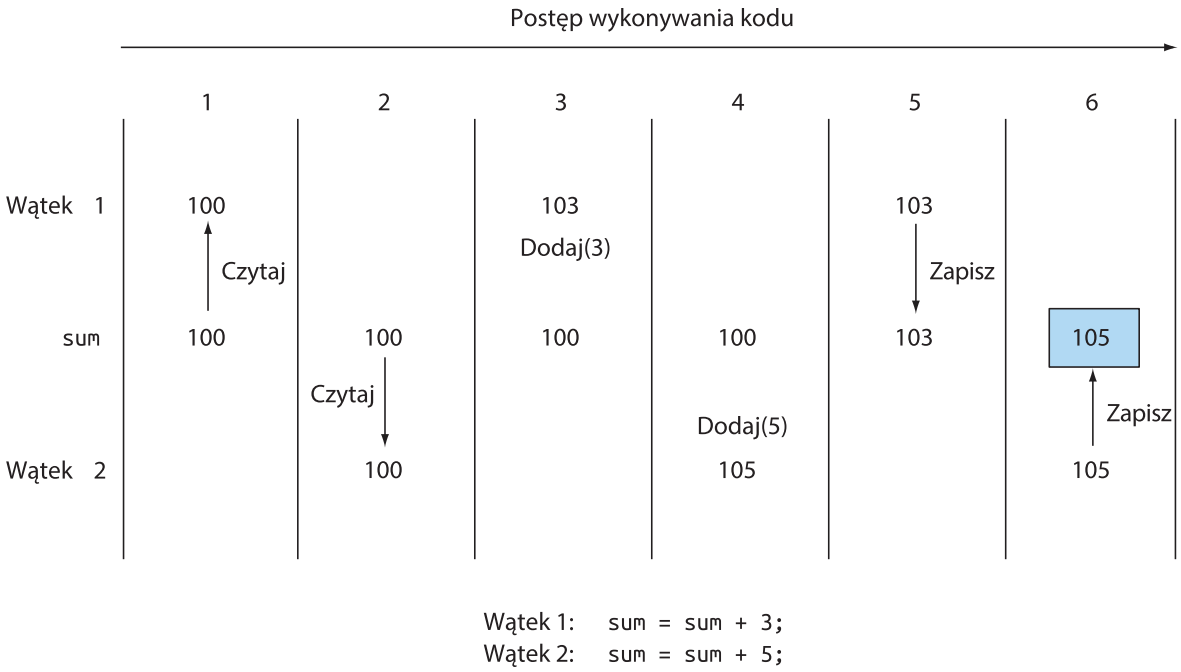
Problem polega na tym, że korzystanie z dobrodziejstw współbieżności poprzez pisanie wielowątkowego kodu (przy wykorzystaniu interfejsu programowania `Threads` z poprzednich wersji języka Java) jest dosyć trudne. Konieczna jest zmiana sposobu myślenia: wątki mogą uzyskiwać dostęp i aktualizować współdzielone zmienne w tym samym czasie, w wyniku czego dane mogą się nieoczekiwanie zmieniać, jeśli nie będą właściwie koordynowane*. Taki model przetwarzania danych jest znacznie trudniejszy w pojęciu niż model sekwencyjny**, wykonywany krok po kroku. Na przykład, rysunek 1.5 pokazuje możliwe wystąpienie problemu z dwoma wątkami (obiektami `Threads`) próbującymi dodać liczbę do współdzielonej zmiennej `sum`, jeśli nie zostaną one poprawnie zsynchronizowane.

Java 8 adresuje oba te problemy (zbędny kod przy przetwarzaniu kolekcji oraz trudności w wykorzystywaniu wielu rdzeni) za pośrednictwem API strumieni (`java.util.stream`). Pierwszym motywatorem projektowym jest to, że istnieje wiele wzorców przetwarzania danych (podobnie jak w przypadku metody `filterApples` w poprzednim podrozdziale, lub operacji zbliżonych do języków zapytań bazy danych, takich jak SQL), które wciąż pojawiają się w wielu miejscach kodu, a które skorzystałyby na uformowaniu fragmentu biblioteki: *filtrowanie* danych na podstawie kryteriów (np. ciężkie jabłka), *wyodrębnianie* danych (np. wyodrębnianie pola wagi z każdego jabłka na liście) lub *grupowanie* danych (np. grupowanie listy liczb na oddzielne listy z liczbami parzystymi i nieparzystymi), i tak dalej. Drugim motywatorem jest to, że operacje takie mogą być często wykonywane równolegle. Na przykład, jak ilustruje to rysunek 1.6, filtrowanie listy na dwóch procesorach może zostać wykonane poprzez poproszenie jednego procesora o przetworzenie pierwszej połowy listy, a drugiego procesora o przetworzenie drugiej połowy listy. Jest to tzw. *rozwidlenie* (ang. *forking; na rysunku krok 1*). Następnie procesory filtrują przydzielone im połówki listy

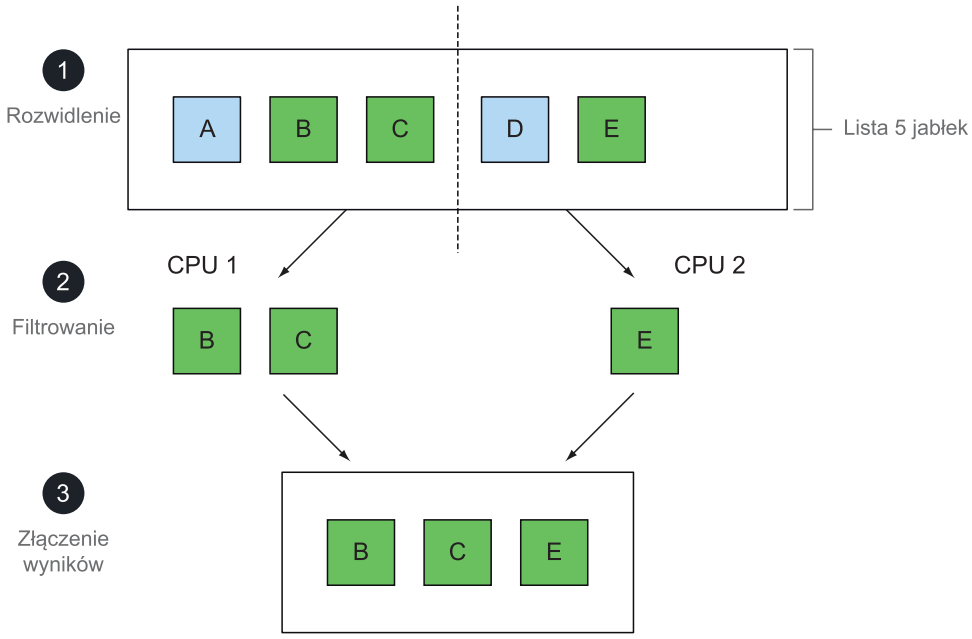
* Tradycyjnie poprzez słowo kluczowe `synchronized`, ale w wyniku jego nieprawidłowego użycia powstaje wiele drobnych błędów. Oparta na strumieniach równoległość Javy 8 zachęca do programowania w stylu funkcyjnym, w którym słowo `synchronized` jest rzadko używane; styl ten skupia się na partycjonowaniu danych zamiast na koordynowaniu dostępu do nich.

** Aha! Źródło nacisku na język, aby ten się rozwijał!

(krok 2). Na koniec (krok 3) jeden procesor dokonuje złączenia dwóch wyników (ma to ścisły związek z tym, dlaczego wyszukiwanie Google działa tak szybko, oczywiście przy użyciu znacznie większej liczby procesorów).



Rysunek 1.5 Możliwy problem z dwoma wątkami próbującymi dodać liczbę do współdzielonej zmiennej sum. Wynikiem jest 105 zamiast oczekiwanego 108.



Rysunek 1.6 Rozwidlenie operacji `filter` na dwa procesory i łączenie wyników

Na ten moment powiemy jedynie, że nowe API strumieni zachowuje się podobnie do istniejącego API kolekcji Javy: oba oferują dostęp do sekwencji elementów danych. Warto

jednak pamiętać, że API kolekcji dotyczy głównie przechowywania i uzyskiwania dostępu do danych, podczas gdy strumienie służą głównie do opisywania obliczeń wykonywanych na danych. Kluczowe jest tutaj to, że strumienie pozwalają i zachęcają do równoległego przetwarzania elementów wewnątrz strumienia. Choć na początku może się to wydawać dziwne, to często najszybszym sposobem na przefiltrowanie kolekcji (np. z użyciem metody `filterApples` na liście w poprzednim podrozdziale) jest skonwertowanie jej na strumień, przetworzenie tego strumienia w sposób równoległy, a następnie skonwertowanie go z powrotem na listę, jak na poniższych przykładach ilustrujących przetwarzanie sekwencyjne i równoległe. Na koniec raz jeszcze powiemy tylko, że „równoległość ta jest niemal darmowa" i podamy przykład tego, jak możemy przefiltrować ciężkie jabłka na liście w sposób sekwencyjny i w sposób równoległy z użyciem strumienia i wyrażenia lambda:

Oto przykład przetwarzania sekwencyjnego:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
```

A oto ten sam kod wykorzystujący przetwarzanie równoległe:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

Równoległość w Javie i brak współdzielonego modyfikowalnego stanu

Ludzie zawsze mówili, że równoległość w Javie jest trudna, a wszystko co związane ze słowem kluczowym `synchronized` jest podatne na błędy. Gdzie jest zatem ten cudowny lek w Javie 8?

Istnieją tak naprawdę dwa cudowne leki. Po pierwsze, biblioteka strumieni obsługuje partycjonowanie, czyli dzielenie dużego strumienia na kilka mniejszych strumieni, które przetwarzane będą w sposób równoległy. Po drugie, ta prawie darmowa równoległość w strumieniach działa tylko wtedy, gdy metody przekazywane do metod bibliotek, takich jak `filter`, nie oddziaływują ze sobą, przykładowo poprzez modyfikowanie współdzielonych obiektów. Ale okazuje się, że ograniczenie to jest dosyć naturalne dla kodera (spójrz chociażby na nasz przykład `Apple::isGreenApple`). Mimo że słowo *funkcyjne* w terminie *programowanie funkcyjne* oznacza przede wszystkim „korzystanie z funkcji jak z wartości pierwszej kategorii", ma ono często również drugie znaczenie w postaci „braku interakcji pomiędzy komponentami w czasie wykonywania".

Równoległe przetwarzanie danych w języku Java 8 i jego wydajność zostaną omówione bardziej szczegółowo w rozdziale 7. Jednym z praktycznych problemów, na jakie programiści natknęli się w ewoluującym języku Java po wprowadzeniu tych wszystkich nowości jest ewolucja istniejących interfejsów. Przykładowo, metoda `Collections.sort` należy

tak naprawdę do interfejsu `List`, ale nigdy nie została dołączona. W idealnym przypadku chcielibyśmy napisać `list.sort(comparator)` zamiast `Collections.sort(list, comparator)`. Może się to wydawać trywialne, ale w wersjach języka Java starszych niż 8 możemy zaktualizować interfejs tylko wtedy, gdy zaktualizujemy wszystkie implementujące go klasy, co jest prawdziwym logistycznym koszmarem. W Javie 8 problem ten rozwiązany został za pomocą metod domyślnych.

1.5 *Metody domyślne i moduły Java*

Jak wspomnieliśmy wcześniej, nowoczesne systemy budowane są przeważnie z komponentów (przykładowo nabywanych z innych źródeł). Historycznie Java nie oferowała zbyt szerokiego wsparcia w tym zakresie, nie licząc plików JAR zawierających zestawy pakietów Java bez żadnej konkretnej struktury. Co więcej, ewoluowanie interfejsów do takich pakietów było trudne, ponieważ zmiana interfejsu w Javie oznaczała zmianę każdej klasy, która go implementuje. Java 8 i Java 9 jako pierwsze adresują ten problem.

Po pierwsze, Java 9 wprowadza system modułów, który dostarcza nam składnię do definiowania *modułów* zawierających kolekcje pakietów, a do tego utrzymuje lepszą kontrolę w zakresie widoczności i przestrzeni nazw. Moduły wzbogacają strukturę prostego komponentu, takiego jak JAR, o strukturę, zarówno w celu dokumentacji, jak i w celu sprawdzania maszynowego; moduły zostały omówione szczegółowo w rozdziale 14. Po drugie, Java 8 wprowadziła metody domyślne w celu wsparcia *ewoluowanych* interfejsów. Zostały one omówione szczegółowo w rozdziale 13. Metody domyślne są istotne, gdyż coraz częściej napotykamy je w interfejsach, ale ponieważ relatywnie niewielu programistów będzie musiało pisać je samodzielnie, a do tego odpowiadają one za ewolucję programów zamiast wspomagać tworzenie jakiegokolwiek konkretnego programu, nasze omówienie tego tematu będzie krótkie i oparte na przykładach:

W podrozdziale 1.4 zaprezentowaliśmy następujący przykład kodu Java 8:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

Ale jest tu pewien problem: interfejs `List<T>` w wersjach starszych niż Java 8 nie zawiera metod `stream` lub `parallelStream` – podobnie z resztą jak interfejs `Collection<T>`, który go implementuje – ponieważ metody te jeszcze nie istniały! A bez tych metod kod ten nie skompiluje się. Najprostszym rozwiązaniem, które możemy zaimplementować w naszych własnych interfejsach, byłoby, gdyby projektanci języka Java 8 po prostu dodali metodę `stream` do interfejsu `Collection` oraz jej implementację w klasie `ArrayList`.

Takie rozwiązanie byłoby jednak koszmarem dla użytkowników. Istnieje wiele alternatywnych zestawów kolekcji, które implementują interfejsy pochodzące z API kolekcji Javy. Dodanie nowej metody do interfejsu oznacza, że wszystkie skonkretyzowane klasy muszą dostarczyć dla niej implementację. Projektanci języka nie mają kontroli nad wszystkimi

istniejącymi implementacjami API kolekcji, tak więc mamy tu pewien dylemat: w jaki sposób możemy ewoluować opublikowane interfejsy bez wywierania wpływu na istniejące implementacje?

Rozwiązaniem oferowanym przez język Java 8 jest zerwanie ostatniego ogniwa – interfejs może teraz zawierać sygnatury metod, dla których klasa implementująca nie dostarcza implementacji! Kto więc implementuje te metody? Brakujące ciała metod dostarczane są jako część interfejsu (stąd też są to implementacje domyślne) zamiast implementacji w klasie implementującej interfejs.

Dzięki temu projektant interfejsu może rozszerzyć go poza metody, które były pierwotnie planowane, jednak bez psucia istniejącego kodu. Aby to osiągnąć, Java 8 pozwala nam wykorzystywać w specyfikacjach interfejsów istniejące słowo kluczowe `default`.

Na przykład, w Javie 8 możemy teraz wywołać metodę `sort` bezpośrednio na liście `List`. Stało się to możliwe dzięki poniższej metodzie domyślnej w interfejsie `List`, który wywołuje statyczną metodę `Collections.sort`:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Oznacza to, że dowolne skonkretyzowane klasy interfejsu `List` nie muszą jawnie implementować metody `sort`, podczas gdy w poprzednich wersjach języka Java takie klasy przestałyby się kompilować, chyba że dostarczałyby implementację dla `sort`.

Ale zaraz, pojedyncza klasa może implementować wiele interfejsów, prawda? Jeśli więc mamy wiele domyślnych implementacji w kilku interfejsach, to czy oznacza to, że mamy tu pewną formę wielokrotnego dziedziczenia w Javie? W pewnym stopniu tak! W rozdziale 13 pokażemy, że istnieją pewne reguły, które pozwalają nam unikać problemów, takich jak niesławny problem diamentowego dziedziczenia w C++.

1.6 *Inne dobre pomysły z programowania funkcyjnego*

Poprzednie podrozdziały wprowadziły dwie podstawowe idee z programowania funkcyjnego, które są teraz częścią języka Java: wykorzystywanie metod i wyrażeń lambda jak wartości pierwszej kategorii oraz koncepcja, że wywołania funkcji lub metod mogą być wydajnie i bezpiecznie wykonywane równolegle przy braku modyfikowalnego stanu współdzielonego. Obie te idee wykorzystywane są przez nowe API strumieni, które omówiliśmy wcześniej.

Powszechne języki funkcyjne (SML, Ocaml czy Haskell) dostarczają ponadto dodatkowe konstrukcje wspomagające pracę programistów. Jedną z nich jest unikanie wartości `null` poprzez jawne użycie bardziej opisowych typów danych. Istotnie, Tony Hoare, jeden z gigantów informatyki, podczas swojej prezentacji na QCon London 2009 powiedział:

> *Nazywam to pomyłką za miliard dolarów. Było nią stworzenie pustej referencji w 1965 roku… Nie mogłem się powstrzymać przed dodaniem pustej referencji, ponieważ była ona tak łatwa do zaimplementowania.*

W języku Java 8 dostępna jest klasa `Optional<T>`, która – jeśli używana w sposób spójny – może nam pomóc ustrzec się przed wyjątkami pustego wskaźnika. Jest to obiekt kontenera,

który może, ale nie musi przechowywać wartości. `Optional<T>` zawiera metody do jawnej obsługi przypadku, kiedy to wartość jest nieobecna, w wyniku czego możemy uniknąć wyjątków pustego wskaźnika. Innymi słowy, dzięki wykorzystaniu systemu typów umożliwia nam ona zasygnalizowanie, że dana zmienna może nie otrzymać wartości. Klasa `Optional<T>` została omówiona w rozdziale 11.

Drugą ideą jest (strukturalne) dopasowywanie do wzorców (ang. *pattern matching*)*, wykorzystywane w matematyce, np.:

```
f(0) = 1
f(n) = n*f(n-1) w przeciwnym wypadku
```

W Javie zastosowalibyśmy tu instrukcję `if-then-else` lub `switch`. Inne języki pokazały jednak, że dla bardziej złożonych typów danych dopasowywanie do wzorców jest w stanie wyrazić koncepcje programistyczne bardziej zwięźle niż konstrukcja `if-then-else`. Dla takich typów danych jako alternatywę dla `if-then-else` możemy również wykorzystywać polimorfizm i przeciążenia metod, ale wśród projektantów nadal toczy się dyskusja co do tego, które z tych rozwiązań jest bardziej odpowiednie**. Naszym zdaniem oba są bardzo przydatne, tak więc powinieneś mieć je oba w swoim arsenale. Niestety, Java 8 nie oferuje pełnego wsparcia dla dopasowywania do wzorców, ale mimo tego w rozdziale 19 pokażemy, w jaki sposób możemy wyrazić je sami. Obecnie przedyskutowywana jest propozycja rozszerzenia języka Java, mająca na celu zaimplementowanie wsparcia dla dopasowywania do wzorców w przyszłej wersji Javy (patrz *http://openjdk.java.net/jeps/305*). W międzyczasie zilustrujmy przykład wyrażony w języku programowania Scala (kolejnym języku zbliżonym do Javy i działającym na maszynie wirtualnej Javy, który zainspirował pewne aspekty ewolucji języka Java; patrz rozdział 20). Załóżmy, że chcemy napisać program, który dokonuje pewnych uproszczeń na drzewie reprezentującym wyrażenie arytmetyczne. Mając typ danych `Expr` reprezentujący takie wyrażenia, w języku Scala możemy napisać poniższy kod, który zdekomponuje wyrażenie `Expr` na jego części składowe, a następnie zwróci kolejne wyrażenie `Expr`:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e                    (1)
    case BinOp("-", e, Number(0)) => e                    (2)
    case BinOp("*", e, Number(1)) => e                    (3)
    case BinOp("/", e, Number(1)) => e                    (4)
    case _ => expr                                        (5)
}
```

(1) Dodaje 0
(2) Odejmuje 0

* Fraza ta ma więcej niż jedno znaczenie. Tutaj mamy na myśli dopasowywanie znane z matematyki i programowania funkcyjnego, w którym to funkcja definiowana jest przez wylistowanie przypadków zamiast przy użyciu instrukcji `if-the-else`. To drugie znaczenie dotyczy fraz typu „znajdź w podanym katalogu wszystkie pliki w formacie 'IMG*.JPG'", powiązanych z tak zwanymi wyrażeniami regularnymi.

** Wstęp do tej dyskusji znajdziesz w angielskiej Wikipedii pod hasłem „expression problem" (termin zaproponowany przez Phila Wadlera).

3. Mnoży przez 1
4. Dzieli przez 1
5. Nie można uprościć za pomocą tych przypadków, więc zostaw bez zmian

W języku Scala składnia `expr match` odpowiada instrukcji `switch(expr)`w Javie; nie przejmuj się na razie tym kodem – więcej na temat dopasowywania do wzorców dowiesz się w rozdziale 19. Na ten moment o dopasowywaniu do wzorców możesz myśleć jak o rozszerzonej instrukcji `switch`, która dodatkowo rozkłada typ danych na jego komponenty składowe.

Dlaczego instrukcja `switch` języka Java powinna być ograniczona do wartości prymitywnych i łańcuchów `String`? Języki funkcyjne pozwalają zwykle na wykorzystywanie instrukcji `switch` na wielu dodatkowych typach danych, wliczając w to operacje dopasowywania do wzorca (w kodzie języka Scala realizujemy to w ramach operacji `match`). W projekcie zorientowanym obiektowo wzorzec odwiedzającego (ang. *visitor*) jest powszechnie stosowanym wzorcem do przechodzenia po rodzinach klas (takich jak różnych komponentach samochodu: kole, silniku, podwoziu, itd.) i aplikowania operacji do każdego odwiedzonego obiektu. Zaletą dopasowywania do wzorców jest to, że kompilator może zgłosić nam niektóre powszechne błędy, takie jak „Klasa Hamulce jest częścią rodziny klas wykorzystywanych do reprezentowania komponentów klasy Samochód. Zapomniałeś jawnie ją obsłużyć".

Rozdziały 18 i 19 stanowią pełne wprowadzenie do programowania funkcyjnego i pokazują, w jaki sposób należy pisać programy w języku Java 8 z użyciem stylu funkcyjnego – wliczając w to zestaw funkcji dostarczanych w bibliotece Javy. Rozdział 20 wyjaśnia, w jaki sposób funkcje języka Java 8 odpowiadają funkcjom języka Scala – języka, który podobnie jak Java zaimplementowany jest na bazie maszyny wirtualnej Javy, i który ewoluował na tyle szybko, że jest teraz w stanie zagrozić pewnym aspektom niszy zajmowanej przez Javę w ekosystemie języków programowania. Zagadnienia te zostały poruszone w końcowej części książki, aby dać czytelnikowi dodatkowe informacje odnośnie tego, dlaczego zostały dodane nowe funkcje języków Java 8 i Java 9.

Funkcje Javy 8, 9, 10 i 11: gdzie zacząć?

Java 8 i Java 9 wprowadziły do języka Java znaczące usprawnienia. Prawdopodobnie jednak to właśnie dodatki języka Java 8 będą mieć na programistów Java największy wpływ podczas realizacji wykonywanych na co dzień zadań – możliwość przekazywania metod lub wyrażeń lambda coraz częściej staje się wymaganą wiedzą. Z kolei usprawnienia języka Java 9 poszerzają naszą zdolność do definiowania i korzystania z komponentów o większej skali, jak na przykład strukturyzowanie systemu za pomocą modułów czy importowanie zestawu narzędzi do programowania reaktywnego. W porównaniu do wspomnianych aktualizacji, Java 10 jest znacznie mniejszym uaktualnieniem, na które składa się głównie umożliwienie nam korzystania z funkcji wnioskowania typów w zmiennych lokalnych, o czym powiemy sobie krótko w rozdziale 21, gdzie wspomnimy także o bogatszej składni dla argumentów wyrażeń lambda, jaka ma zostać wprowadzona w Javie 11.

ciąg dalszy na stronie następnej

ciąg dalszy ze strony poprzedniej

W czasie pisania tej książki wydanie Javy 11 planowane jest na wrzesień 2018. Java 11 przynosi również nową asynchroniczną bibliotekę klienta HTTP (*http://openjdk.java.net/jeps/321*), która wykorzystuje obiekty `CompletableFuture` i programowanie reaktywne, wprowadzone w Javie 8 i Javie 9 (szczegóły na ich temat znajdują się w rozdziałach 15, 16 i 17).

Podsumowanie

- Miej na uwadze ideę ekosystemu języka i wynikający z niego nacisk „ewoluuj albo przemiń" na języki programowania. Choć Java może obecnie uchodzić za zdrowy i silny język, warto przypomnieć sobie inne zdrowe języki, takie jak COBOL, którym nie udało się ewoluować.
- Podstawowe dodatki do języka Java 8 dostarczają szereg nowych koncepcji i funkcjonalności, dzięki którym łatwiej możemy pisać zwięzłe i efektywne programy.
- Procesory wielordzeniowe nie są w pełni obsługiwane przez istniejące praktyki programistyczne stosowane w Javie.
- Funkcje są wartościami pierwszej kategorii; pamiętaj o tym, w jaki sposób metody mogą być przekazywane jako wartości funkcyjne i jak zapisywane są funkcje anonimowe (wyrażenia lambda).
- Koncepcja strumieni w języku Java 8 generalizuje wiele aspektów kolekcji, ale zapewnia bardziej czytelny kod i pozwala na równoległe przetwarzanie elementów strumienia.
- Oparte na komponentach programowanie na dużą skalę oraz ewoluowanie interfejsów systemu nie były do tej pory dobrze obsługiwane przez Javę. Jednak w Javie 9 możemy teraz określać moduły w celu ustrukturyzowania pisanych systemów i wykorzystywać metody domyślne, aby umożliwić rozszerzenie interfejsu bez konieczności modyfikowania wszystkich klas, które go implementują.
- Do pozostałych interesujących koncepcji programowania funkcyjnego możemy zaliczyć pracę z wartościami `null` oraz korzystanie z dopasowywania do wzorców.

2 Przekazywanie kodu z parametryzacją zachowania

W tym rozdziale:

- Radzenie sobie ze zmieniającymi się wymaganiami
- Parametryzacja zachowania
- Klasy anonimowe
- Pierwsze spojrzenie na wyrażenia lambda
- Praktyczne przykłady: `Comparator`, `Runnable` i graficzny interfejs użytkownika

Dobrze znanym problemem w inżynierii oprogramowania jest to, że czego byśmy nie robili, wymagania użytkowników i tak będą się zmieniać. Na przykład, wyobraźmy sobie aplikację, która pomaga farmerowi lepiej poznać jego inwentarz. Farmer może zażyczyć sobie funkcjonalności, która pozwoli mu wyszukać w jego inwentarzu wszystkie zielone jabłka. Ale już następnego dnia może nam on powiedzieć: „Właściwie to chciałbym jeszcze móc wyszukiwać wszystkie jabłka o wadze większej niż 150 g". Dwa dni później farmer wraca do nas i dodaje: „Byłoby fajnie, gdybym mógł znaleźć wszystkie jabłka, które są jednocześnie zielone i cięższe niż 150 g". W jaki sposób możemy poradzić sobie z tak zmieniającymi się wymaganiami? Byłoby idealnie, gdyby udało nam się zminimalizować nasze wysiłki inżynierskie. Dodatkowo, podobne nowe funkcjonalności powinny być łatwe do zaimplementowania i utrzymania przez długi czas.

Parametryzacja zachowania jest wzorcem projektowania oprogramowania, który pozwala nam poradzić sobie z częstymi zmianami wymagań. Mówiąc w skrócie, oznacza to udostępnianie wybranego bloku kodu bez jego uruchamiania. Taki blok kodu może być wywoływany przez inne fragmenty naszych programów, co oznacza, że możemy opóźnić wykonanie tego bloku kodu. Na przykład, wybrany blok kodu moglibyśmy przekazać jako argument do innej metody, która wykona go później. W rezultacie zachowanie metody jest sparametryzowane w oparciu o ten blok kodu. Jeśli przykładowo przetwarzamy jakąś kolekcję, możemy zechcieć napisać metodę, która

- Może zrobić „coś" dla każdego elementu listy
- Może zrobić „coś innego" po zakończeniu przetwarzania listy
- Może zrobić „coś jeszcze innego", jeśli napotkamy błąd

Do tego właśnie odnosi się parametryzacja zachowania. Oto pewna analogia: nasz współlokator wie, jak dojechać samochodem do supermarketu i wrócić do domu. Możemy więc powiedzieć mu, by kupił chleb, ser i wino. Odpowiednikiem tego w kodzie jest wywołanie metody `goAndBuy` z argumentem w postaci listy produktów. Ale pewnego dnia jesteśmy w biurze i chcemy poprosić go, aby zrobił dla nas coś, czego nigdy jeszcze nie robił, tj. pojechał na pocztę, użył podanego przez nas numeru referencyjnego, porozmawiał z menedżerem i odebrał paczkę. Listę instrukcji moglibyśmy przekazać mu poprzez e-mail, a gdy tylko ją otrzyma, będzie mógł przystąpić do ich wykonywania. Zrobiliśmy teraz coś odrobinę bardziej zaawansowanego, czego odpowiednikiem jest metoda `go`, która może przyjmować w formie argumentów różne nowe zachowania i wykonywać je.

Rozdział ten rozpoczniemy od przestudiowania przykładu, który pokaże nam, w jaki sposób możemy rozwijać nasz kod, aby uczynić go bardziej elastycznym w kontekście zmieniających się wymagań. Bazując na tej wiedzy pokażemy również sposób wykorzystywania parametryzacji zachowania na kilku praktycznych przykładach. Nie wykluczone, że wykorzystywałeś już wcześniej wzorzec parametryzacji zachowania przy użyciu istniejących klas i interfejsów z API Javy podczas sortowania listy, filtrowania nazw plików lub nawet informowania wątku, by ten wykonał określony blok kodu lub obsłużył zdarzenie graficznego interfejsu użytkownika. Wkrótce przekonasz się, że wykorzystywanie tego wzorca w Javie generuje dosyć rozwlekły kod. Na szczęście wyrażenia lambda w Javie 8 i nowszych rozwiązują ten problem. W rozdziale 3 wyjaśnimy, jak konstruować wyrażenia lambda, gdzie ich używać oraz w jaki sposób możemy za ich pomocą uczynić nasz kod bardziej zwięzłym.

2.1 *Radzenie sobie ze zmieniającymi się wymaganiami*

Pisanie kodu, który jest w stanie poradzić sobie ze zmieniającymi się wymaganiami, nie jest wcale łatwe. Spójrzmy teraz na przykład, który będziemy stopniowo usprawniać, prezentując przy tym najlepsze praktyki mające na celu zwiększenie elastyczności naszego kodu. W kontekście wspomnianej aplikacji inwentarza na farmie, musimy zaimplementować funkcjonalność do filtrowania na liście zielonych jabłek. Nie brzmi to skomplikowanie, prawda?

2.1.1 *Pierwsza próba: filtrowanie zielonych jabłek*

Załóżmy, jak w rozdziale 1, że mamy typ wyliczeniowy `Color` do reprezentowania różnych kolorów jabłek:

```
enum Color { RED, GREEN }
```

Nasze pierwsze rozwiązanie może być następujące:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
     List<Apple> result = new ArrayList<>();                          (1)
     for(Apple apple: inventory){
         if( GREEN.equals(apple.getColor() ) {                        (2)
               result.add(apple);
         }
     }
     return result;
}
```

(1) Lista akumulująca dla jabłek
(2) Wybierz tylko zielone jabłka

Wytłuszczony wiersz wskazuje warunek wymagany do wyboru zielonych jabłek. Zakładamy tu istnienie typu wyliczeniowego `Color` z predefiniowanym zestawem kolorów, takich jak `GREEN`. Teraz jednak farmer zmienia swoje zdanie i chce być również w stanie filtrować jabłka czerwone. Co możemy zrobić? Rozwiązaniem naiwnym byłoby zdublowanie tej metody, zmiana jej nazwy na `filterRedApples` oraz zmodyfikowanie warunku `if` w celu dopasowania do czerwonych jabłek. Tak czy inaczej, podejście to nie poradzi sobie zbyt dobrze ze zmianami, jeśli farmer będzie chciał filtrować również jabłka w innych kolorach: jasnozielonym, ciemnoczerwonym, żółtym, itd. Dobra zasada brzmi następująco: jeśli napisany przez Ciebie kod jest podobny do już istniejącego, spróbuj go wyabstrahować.

2.1.2 *Druga próba: parametryzacja koloru*

W jaki sposób możemy uniknąć zdublowania większości kodu z metody `filterGreenApples` podczas tworzenia metody `filterRedApples`? Do naszej metody moglibyśmy dodać nowy argument w celu sparametryzowania koloru, co pozwoli nam lepiej reagować na zmiany:

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
                                               Color color) {
     List<Apple> result = new ArrayList<>();
     for (Apple apple: inventory) {
         if ( apple.getColor().equals(color) ) {
             result.add(apple);
         }
     }
```

```
    return result;
}
```

Możemy teraz uszczęśliwić naszego farmera i wywoływać naszą metodę w poniższy sposób:

```
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
…
```

Zbyt łatwe, co? Spróbujmy teraz nieco utrudnić nasz przykład. Farmer powraca i mówi, „Byłoby naprawdę fajnie, gdybym mógł odróżnić jabłka lekkie od ciężkich. Jabłka ciężkie ważą zwykle więcej niż 150 g".

Mając doświadczenie w inżynierii oprogramowania uświadamiamy sobie zawczasu, że farmer może też zechcieć rozróżniać inne wagi, tak więc tworzymy poniższą metodę, która poradzi sobie z różnymi wagami jabłek dzięki zastosowaniu dodatkowego parametru:

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                               int weight) {
    List<Apple> result = new ArrayList<>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ) {
            result.add(apple);
        }
    }
    return result;
}
```

Jest to dobre rozwiązanie, ale zwróćmy uwagę, że musimy zdublować większość implementacji odpowiedzialnej za przejście po inwentarzu i zastosowanie kryteriów filtrowania do każdego jabłka. Jest to w pewnym stopniu rozczarowujące, ponieważ łamie to zasadę inżynierii oprogramowania DRY (ang. *Don't Repeat Yourself* – nie powtarzaj się). Co jeśli chcielibyśmy zmienić sposób filtrowania, aby zwiększyć wydajność? Musielibyśmy teraz zmodyfikować implementację wszystkich naszych metod zamiast jednej. Z perspektywy wysiłku inżynierskiego będzie to bardzo kosztowne.

Kolor i wagę moglibyśmy połączyć do postaci jednej metody o nazwie `filter`, ale wtedy nadal musielibyśmy jakoś rozróżniać, po którym atrybucie chcemy filtrować. W tym celu moglibyśmy wstawić odpowiednią flagę, która rozróżniałaby od siebie obydwa te zapytania (ale nigdy nie powinniśmy tego robić! Za moment wyjaśnimy dlaczego).

2.1.3 *Trzecia próba: filtrowanie z użyciem dowolnego atrybutu, jaki przyjdzie nam na myśl*

Nasza brzydka próba scalenia wszystkich atrybutów może wyglądać tak:

```
public static List<Apple> filterApples(List<Apple> inventory, Color color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( (flag && apple.getColor().equals(color)) ||
```

```
            (!flag && apple.getWeight() > weight) ){                    ❶
            result.add(apple);
        }
    }
    return result;
}
```

❶ Naprawdę brzydki sposób wybierania koloru lub wagi

Moglibyśmy skorzystać z tej metody w poniższy (i naprawdę brzydki) sposób:

```
List<Apple> greenApples = filterApples(inventory, GREEN, 0, true);
List<Apple> heavyApples = filterApples(inventory, null, 150, false);
…
```

Rozwiązanie to jest ekstremalnie złe. Przede wszystkim kod klienta wygląda okropnie. Co oznaczają tu wartości `true` i `false`? Mało tego, rozwiązanie to nie radzi sobie dobrze ze zmieniającymi się wymaganiami. Co będzie, jeśli farmer poprosi nas o przefiltrowanie jabłek na postawie innych atrybutów, takich jak rozmiar, kształt czy pochodzenie? A co, jeśli poprosi nas o bardziej zaawansowane zapytania, które łączą poszczególne atrybuty, jak np. zielone jabłka, które są jednocześnie ciężkie? Musielibyśmy wówczas albo napisać wiele zdublowanych metod `filter`, albo jedną dużą i bardzo złożoną metodę. Jak do tej pory parametryzowaliśmy metodę `filterApples` przy użyciu *wartości*, takich jak `String`, `Integer`, typ wyliczeniowy czy `boolean`. Dla pewnych dobrze zdefiniowanych problemów może to być wystarczające, ale w tym przypadku potrzebujemy lepszego sposobu na wskazanie metodzie `filterApples` kryteriów wybierania jabłek. W kolejnym podrozdziale omówimy sposób wykorzystania parametryzacji zachowania w celu zyskania tej elastyczności.

2.2 *Parametryzacja zachowania*

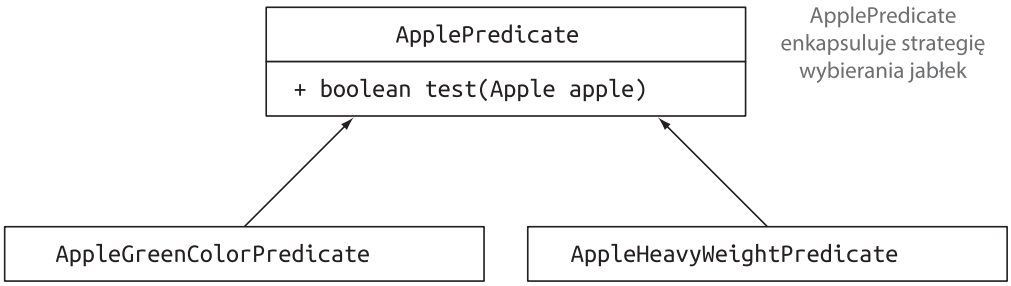
W poprzednim podrozdziale zobaczyliśmy, że potrzebny nam jest lepszy sposób niż dodawanie dużej liczby parametrów, abyśmy mogli poradzić sobie ze zmieniającymi się wymaganiami. Zróbmy teraz krok wstecz i poszukajmy lepszego poziomu abstrakcji. Jednym z możliwych rozwiązań jest wymodelowanie naszych kryteriów wyszukiwania: pracujemy z jabłkami i zwracamy wartość `boolean` na podstawie pewnych atrybutów jabłek (na przykład, czy jest zielone? Czy jest cięższe niż 150 g?). Nazywamy to predykatem (tj. funkcją, która zwraca wartość logiczną `boolean`). Zdefiniujmy więc interfejs do modelowania kryteriów wyszukiwania:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

Możemy teraz zadeklarować wiele implementacji interfejsu `ApplePredicate` do reprezentowania różnych kryteriów wyboru, na przykład (zobacz również rysunek 2.1):

```
public class AppleHeavyWeightPredicate implements ApplePredicate {          (1)
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {           (2)
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
```

(1) Wybierz tylko ciężkie jabłka
(2) Wybierz tylko zielone jabłka



Rysunek 2.1 Różne strategie wybierania jabłek

Kryteria te możemy postrzegać jako różne zachowania dla metody `filter`. To, co właśnie zrobiliśmy, związane jest ze wzorcem projektowym Strategia*, który pozwala nam zdefiniować pewną rodzinę algorytmów, enkapsulować każdy z nich (nazywany strategią) i wybrać odpowiedni algorytm w czasie działania programu. W tym przypadku rodziną algorytmów jest ApplePredicate, zaś różnymi strategiami są `AppleHeavyWeightPredicate` i `AppleGreenColorPredicate`.

W jaki sposób możemy jednak wykorzystać te różne implementacje `ApplePredicate`? Nasza metoda `filterApples` musi przyjmować obiekty `ApplePredicate` do przetestowania warunku na jabłku. W taki właśnie sposób definiujemy *parametryzację zachowania*: jest to zdolność do poinformowania metody, by przyjęła ona różne zachowania (lub strategie) w formie parametrów i wykorzystała je wewnątrz do osiągnięcia różnych zachowań.

Aby zrealizować to w naszym przykładzie, dodajemy parametr do metody `filterApples` w celu przyjęcia obiektu `ApplePredicate`. Przynosi to znaczące korzyści w zakresie inżynierii oprogramowania: możemy teraz oddzielić logikę iterowania po kolekcji wewnątrz metody `filterApples` od zachowania, które chcemy zastosować do każdego elementu tej kolekcji (w tym wypadku predykatu).

* Patrz *http://en.wikipedia.org/wiki/Strategy_pattern*

2.2.1 *Czwarta próba: filtrowanie na podstawie kryteriów abstrakcyjnych*

Nasza zmodyfikowana metoda `filter` wykorzystująca `ApplePredicate` wygląda następująco:

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if(p.test(apple)) {                                    ❶
            result.add(apple);
        }
    }
    return result;
}
```

❶ Predykat p enkapsuluje warunek do przetestowania na jabłku

Przekazywanie kodu/zachowania

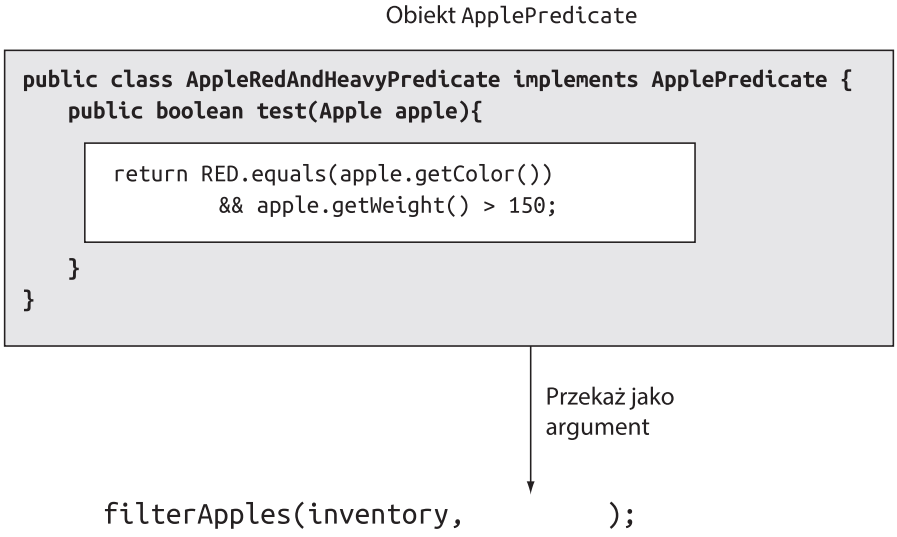
Zatrzymajmy się teraz na chwilę i trochę poświętujmy. Ten kod jest znacznie bardziej elastyczny niż nasza pierwsza próba, a jednocześnie jest on łatwiejszy w użyciu i bardziej czytelny! Możemy teraz utworzyć różne obiekty `ApplePredicate` i przekazać je do metody `filterApples`. Darmowa elastyczność! Jeśli teraz farmer poprosi nas przykładowo o znalezienie wszystkich czerwonych jabłek, które są cięższe niż 150g, to wszystko co musimy zrobić, to utworzyć klasę, która implementuje odpowiedni predykat `ApplePredicate`. Nasz kod jest wystarczająco elastyczny, aby mógł zaakceptować dowolną zmianę wymagań dotyczącą atrybutów jabłka:

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
        public boolean test(Apple apple){
                return RED.equals(apple.getColor())
                       && apple.getWeight() > 150;
        }
}
List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

Osiągnęliśmy coś naprawdę wyjątkowego: zachowanie metody `filterApples` zależy od kodu, który do niej przekazujemy za pośrednictwem obiektu `ApplePredicate`. Innymi słowy, sparametryzowaliśmy zachowanie metody `filterApples`!

Zwróćmy uwagę, że w poprzednim przykładzie znaczenie ma jedynie kod będący implementacją metody `test`, jak to pokazano na rysunku 2.2. To właśnie on definiuje nowe zachowania dla metody `filterApples`. Niestety, ponieważ metoda `filterApples` może przyjmować wyłącznie obiekty, musieliśmy opakować ten kod w obiekt `ApplePredicate`. Jest to podobne do „przekazywania fragmentu kodu" bezpośrednio w wierszu kodu, ponieważ przekazujemy wyrażenie `boolean` przez obiekt, który implementuje metodę `test`. W podrozdziale 2.3 (a bardziej szczegółowo w rozdziale 3) zobaczymy, że dzięki wykorzystaniu

wyrażeń lambda wyrażenie `RED.equals(apple.getColor()) && apple.getWeight() > 150` możemy przekazać do metody `filterApples` bezpośrednio, bez konieczności definiowania wielu klas `ApplePredicate`, usuwając przy tym niepotrzebną rozwlekłość w kodzie.



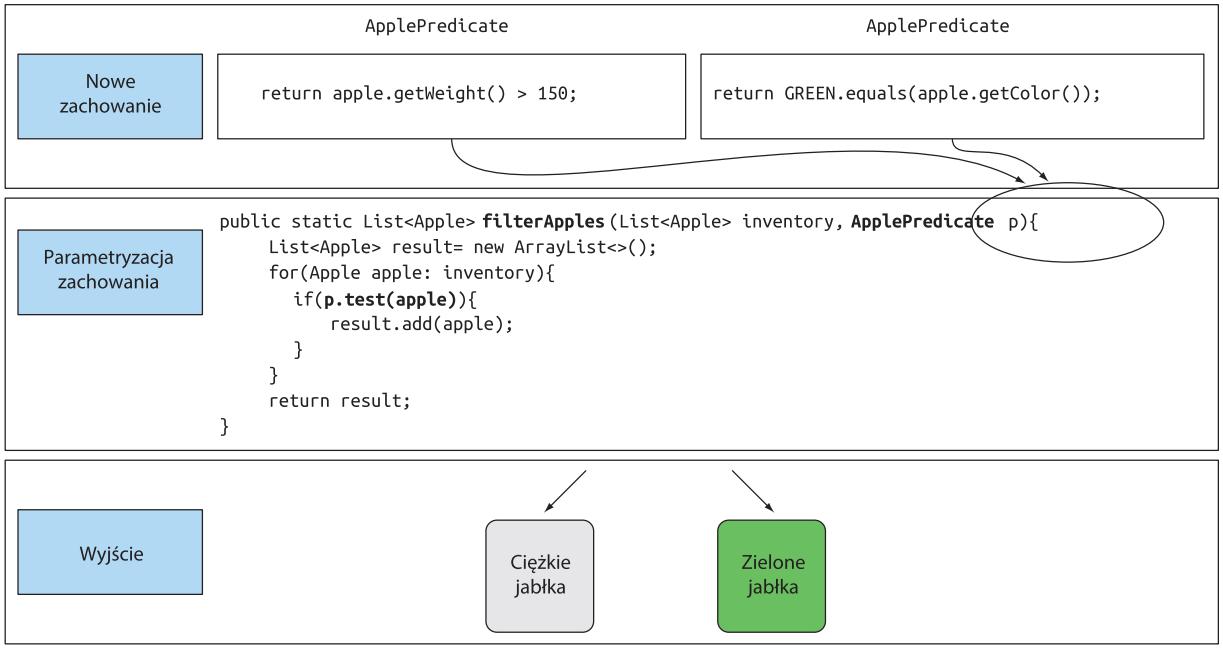
Przekaż strategię do metody filtrującej: przefiltruj jabłka za pomocą wyrażenia boolean enkapsulowanego wewnątrz obiektu ApplePredicate. W celu enkapsulacji tego fragmentu kodu został on opakowany znaczną ilością niepotrzebnego kodu (wytłuszczony fragment).

Rysunek 2.2 Parametryzowanie zachowania metody `filterApples` i przekazywanie różnych strategii filtrowania

Wiele zachowań, jeden parametr

Jak wyjaśniliśmy wcześniej, parametryzacja zachowania jest świetna, ponieważ umożliwia nam ona oddzielenie logiki iterowania po kolekcji, która ma zostać przefiltrowana, od zachowania, które ma być aplikowane na każdym elemencie tej kolekcji. W konsekwencji możemy ponownie użyć tej samej metody i nadać jej inne zachowania w celu osiągnięcia różnych rzeczy, jak to przedstawiono na rysunku 2.3. To właśnie dlatego *parametryzacja zachowania* jest przydatną koncepcją, z której powinniśmy korzystać podczas tworzenia elastycznych interfejsów programowania aplikacji.

Aby upewnić się, że dobrze pojmujesz ideę parametryzacji zachowania, spróbuj teraz rozwiązać quiz 2.1!



Rysunek 2.3 Parametryzowanie zachowania metody filterApples i przekazywanie różnych strategii filtrowania

Quiz 2.1: Napisz elastyczną metodę `prettyPrintApple`

Napisz metodę `prettyPrintApple` przyjmującą listę (`List`) jabłek (`Apple`), która w ramach parametryzacji zachowania może produkować łańcuchy `String` z jabłek (trochę na wzór wielu niestandardowych metod `toString`). Przykładowo, możesz poprosić metodę `prettyPrintApple` o wypisanie wyłącznie wagi każdego z jabłek. Dodatkowo możesz poinstruować swoją metodę `prettyPrintApple`, aby wypisała każde jabłko z osobna i oceniła, czy jest ono ciężkie, czy lekkie. Rozwiązanie podobne jest do przykładu filtrowania, które omówiliśmy wcześniej. Aby ułatwić rozpoczęcie tego zadania, przygotowaliśmy następujący szkielet metody `prettyPrintApple`:

```
public static void prettyPrintApple(List<Apple> inventory, ???) {
    for(Apple apple: inventory) {
        String output = ???.???(apple);
        System.out.println(output);
    }
}
```

Odpowiedź:

Przede wszystkim potrzebujesz jakiegoś sposobu na reprezentowanie zachowania, które przyjmuje jabłko i zwraca sformatowany wynik typu `String`. Podobnej rzeczy dokonaliśmy podczas tworzenia interfejsu `ApplePredicate`:

```
public interface AppleFormatter {
    String accept(Apple a);
}
```

ciąg dalszy na stronie następnej

ciąg dalszy ze strony poprzedniej

Teraz poprzez zaimplementowanie interfejsu `AppleFormatter` będziesz mógł przedstawić wiele zachowań formatowania:

```
public class AppleFancyFormatter implements AppleFormatter {
     public String accept(Apple apple) {
         String characteristic = apple.getWeight() > 150 ? "heavy" : "light";
         return "A " + characteristic +
                "  " + apple.getColor() +" apple";
     }
}
public class AppleSimpleFormatter implements AppleFormatter {
     public String accept(Apple apple) {
         return "An apple of " + apple.getWeight() + "g";
     }
}
```

Na koniec musisz poinformować swoją metodę `prettyPrintApple`, by przyjmowała obiekty typu `AppleFormatter` i wykorzystywała je w swoim ciele. Możesz to osiągnąć poprzez dodanie parametru do metody `prettyPrintApple`:

```
public static void prettyPrintApple(List<Apple> inventory,
                                     AppleFormatter formatter) {
   for(Apple apple: inventory) {
     String output = formatter.accept(apple);
     System.out.println(output);
   }
}
```

Bingo! Jesteś teraz w stanie przekazywać wiele zachowań do swojej metody `prettyPrintApple`. Możesz to osiągnąć poprzez tworzenie instancji z implementacji interfejsu `AppleFormatter` i podanie ich jako argumentów do metody `prettyPrintApple`:

```
prettyPrintApple(inventory, new AppleFancyFormatter());
```

Spowoduje to wyprodukowanie następującego wyjścia:

```
A light green apple
A heavy red apple
…
```

Z kolei kod:

```
prettyPrintApple(inventory, new AppleSimpleFormatter());
```

Wygeneruje nam poniższy rezultat:

```
An apple of 80g
An apple of 155g
…
```

Zobaczyliśmy, że poprzez wyabstrahowanie zachowania możemy sprawić, że nasz kod dostosuje się do zmienionych wymagań, ale jest dosyć rozwlekły proces, ponieważ musimy zadeklarować wiele klas, które instancjonowane będą tylko raz. Zobaczmy więc teraz, w jaki sposób możemy ten proces usprawnić.

2.3 *Zwalczanie rozwlekłości*

Wszyscy wiemy, że nowa funkcja lub koncepcja, która jest trudna w wykorzystywaniu, będzie unikana. Na ten moment, jeśli chcemy przekazać nowe zachowanie do naszej metody `filterApples`, jesteśmy zmuszeni zadeklarować kilka klas implementujących interfejs `ApplePredicate`, a następnie utworzyć kilka instancji obiektów `ApplePredicate`, które alokowane będą tylko raz. Zostało to zilustrowane na poniższym listingu, który podsumowuje wszystko, co do tej pory widzieliśmy. Mamy tu do czynienia z dość rozwlekłym kodem, który pochłania także nasz czas.

Listing 2.1 Parametryzacja zachowania: filtrowanie jabłek z użyciem predykatów

```
public class AppleHeavyWeightPredicate implements ApplePredicate {                ❶
     public boolean test(Apple apple) {
         return apple.getWeight() > 150;
     }
}
public class AppleGreenColorPredicate implements ApplePredicate {                 ❷
     public boolean test(Apple apple) {
         return GREEN.equals(apple.getColor());
     }
}
public class FilteringApples {
     public static void main(String…args) {
         List<Apple> inventory = Arrays.asList(new Apple(80, GREEN),
                                               new Apple(155, GREEN),
                                              new Apple(120, RED));
         List<Apple> heavyApples =
              filterApples(inventory, new AppleHeavyWeightPredicate());            ❸
         List<Apple> greenApples =
              filterApples(inventory, new AppleGreenColorPredicate());             ❹
     }
     public static List<Apple> filterApples(List<Apple> inventory,
                                            ApplePredicate p) {
         List<Apple> result = new ArrayList<>();
         for (Apple apple : inventory) {
             if (p.test(apple)){
                 result.add(apple);
             }
```

```
            }
            return result;
        }
    }
```

1. Wybiera ciężkie jabłka
2. Wybiera zielone jabłka
3. Rezultat będzie listą zawierającą jedno jabłko o wadze 155 g
4. Rezultat będzie listą zawierającą dwa zielone jabłka

Jest to tylko zbędny narzut, ale czy możemy zrobić to lepiej? Java oferuje mechanizm anonimowych klas, które pozwalają nam jednocześnie deklarować i tworzyć instancje klas. Klasy anonimowe pozwalają nam w pewnym stopniu usprawnić nasz kod, czyniąc go nieco bardziej zwięzłym. Nie są one jednak w pełni satysfakcjonujące. W podrozdziale 2.3.3 pokażemy krótki przykład tego, jak wyrażenia lambda mogą poprawić czytelność naszego kodu, zanim omówimy je szczegółowo w kolejnym rozdziale.

2.3.1 *Klasy anonimowe*

Klasy anonimowe zachowują się jak klasy lokalne Javy (klasy zdefiniowane w ramach bloku), z którymi jesteś już zaznajomiony. Ale klasy anonimowe nie mają zdefiniowanej nazwy. Pozwalają nam one deklarować ciała klas i tworzyć ich instancje w ramach tego samego wyrażenia. Innymi słowy, pozwalają nam one tworzyć implementacje ad hoc.

2.3.2 *Piąta próba: korzystanie z anonimowych klas*

Poniższy kod pokazuje, w jaki sposób możemy przepisać nasz przykład filtrowania poprzez utworzenie obiektu, który implementuje `ApplePredicate` z użyciem klasy anonimowej:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {          (1)
    public boolean test(Apple apple){
        return RED.equals(apple.getColor());
    }
});
```

1. Parametryzuje zachowanie metody filterApples za pomocą anonimowej klasy!

Klasy anonimowe są często wykorzystywane w aplikacjach z graficznym interfejsem użytkownika do tworzenia obiektów obsługi zdarzeń. Nie chcemy tu przywoływać bolesnych wspomnień z biblioteki Swing, ale jest to bardzo powszechny wzorzec, który stosowany jest w praktyce (tutaj przy wykorzystaniu API JavaFX, nowoczesnej platformy interfejsu użytkownika dla języka Java):

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Woooo a click!!");
```

```
        }
});
```

Ale klasy anonimowe nie są wystarczająco dobre. Po pierwsze, są one często masywne i zajmują dużo miejsca, co widać po poniższym wytłuszczonym kodzie wykorzystującym te same dwa przykłady, z których korzystaliśmy wcześniej:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
      public boolean test(Apple a){
         return RED.equals(a.getColor());
      }
});
button.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent event) {
         System.out.println("Woooo a click!!");
      }
});
```

❶ Sporo niepotrzebnego kodu!

A po drugie wielu programistów nie do końca wie, jak należy z nich korzystać. Przykładowo quiz 2.2 pokazuje klasyczną zagadkę w Javie, na której wykłada się większość programistów! Sprawdź, czy sobie z nią poradzisz.

W ogólnym przypadku rozwlekłość jest zła, ponieważ zniechęca ona do korzystania z funkcji języka, gdyż pisanie i utrzymywanie rozwlekłego kodu jest długotrwałe, a ponadto kodu takiego nie czyta się zbyt przyjemnie! Dobry kod powinien być łatwy do zrozumienia już po chwili od rozpoczęcia jego czytania. Mimo że klasy anonimowe w pewnym stopniu zwalczają rozwlekłość związaną z deklarowaniem wielu skonkretyzowanych klas dla interfejsu, to nadal nie są one wystarczająco satysfakcjonujące. W kontekście przekazywania prostego fragmentu kodu (np. wyrażenia `boolean` reprezentującego kryterium wyboru), nadal musimy utworzyć obiekt i jawnie zaimplementować metodę do zdefiniowania nowego zachowania (np. metodę `test` dla interfejsu `Predicate` lub metodę `handle` dla interfejsu `EventHandler`).

Zachęcamy programistów do korzystania ze wzorca parametryzacji zachowania, ponieważ jak widzieliśmy, sprawia ona, że nasz kod lepiej przystosowuje się do zmieniających się wymagań. W rozdziale 3 zobaczymy, że projektanci języka Java 8 rozwiązali problem rozwlekłości poprzez wprowadzenie wyrażeń lambda, które zapewniają bardziej zwięzły sposób na przekazywanie kodu. Dosyć już tej niepewności, oto krótki przegląd tego, jak wyrażenia lambda mogą pomóc nam w poszukiwaniu czystego kodu.

Quiz 2.2: Zagadka anonimowych klas

Co będzie wynikiem po wykonaniu poniższego kodu? 4, 5, 6, a może 42?

```
public class MeaningOfThis {
     public final int value = 4;
     public void doIt() {
         int value = 6;
         Runnable r = new Runnable() {
                 public final int value = 5;
                 public void run(){
                     int value = 10;
                     System.out.println(this.value);
                 }
             };
             r.run();
         }
         public static void main(String…args) {
                    MeaningOfThis m = new MeaningOfThis();
             m.doIt();                                                    (1)
         }
}
```

(1) Co będzie wynikiem działania tego wiersza?

Odpowiedź:

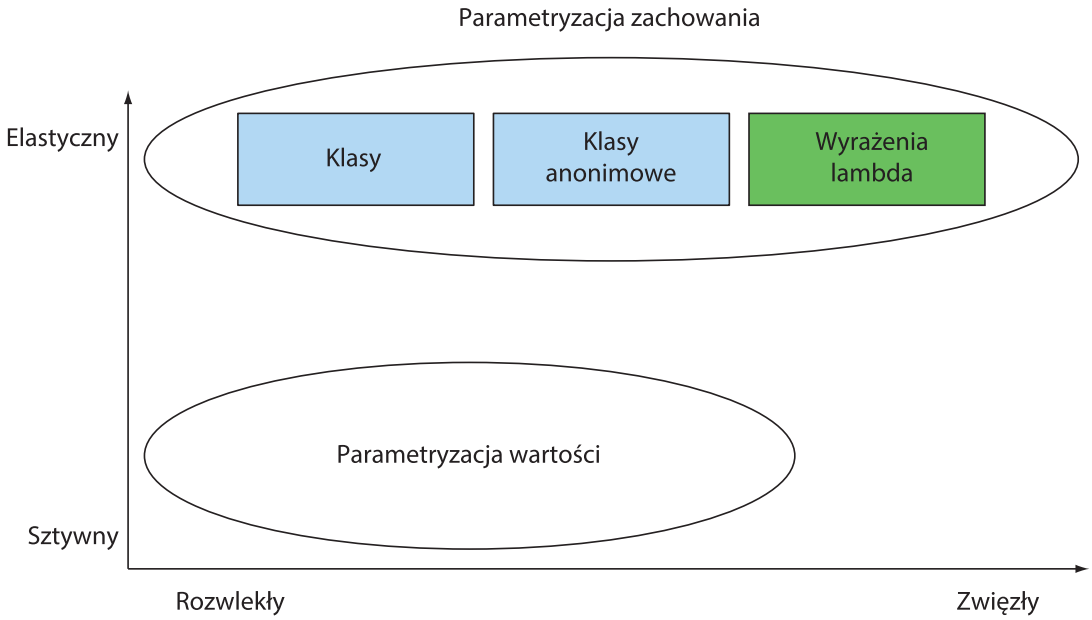
Odpowiedzią jest 5, ponieważ obiekt `this` odnosi się do otaczającej klasy anonimowej `Runnable`, a nie otaczającej ją klasy `MeaningOfThis`.

2.3.3 *Szósta próba: korzystanie z wyrażeń lambda*

Poprzedni kod możemy przepisać w Javie 8 w poniższy sposób, wykorzystując do tego wyrażenie lambda:

```
List<Apple> result =
  filterApples(inventory, (Apple apple) -> RED.equals(apple.getColor()));
```

Musisz przyznać, że powyższy kod wygląda dużo bardziej przejrzyście niż nasze poprzednie próby! To świetnie, ponieważ zaczyna on coraz bardziej przypominać definicję naszego problemu. Rozwiązaliśmy teraz problem rozwlekłości. Rysunek 2.3 podsumowuje naszą dotychczasową podróż.



Rysunek 2.4 Parametryzacja zachowania kontra parametryzacja wartości

2.3.4 *Siódma próba: wyabstrahowany typ List*

Jest jeszcze jeden krok, który możemy wykonać w naszej podróży w kierunku abstrakcji. Na chwilę obecną metoda `filterApples` działa tylko na obiektach `Apple`. Możemy jednak wyabstrahować typ `List`, aby wyjść w ten sposób poza dziedzinę naszego problemu:

```
public interface Predicate<T> {
     boolean test(T t);
}
public static <T> List<T> filter(List<T> list, Predicate<T> p) {          (1)
    List<T> result = new ArrayList<>();
    for(T e: list) {
         if(p.test(e)) {
             result.add(e);
         }
    }
    return result;
}
```

(1) Wprowadza parametr typu T

Teraz możemy używać naszej metody `filter` z listą bananów, pomarańczy, liczb całkowitych lub łańcuchów znakowych! Oto przykład wykorzystujący wyrażenia lambda:

```
List<Apple> redApples =
    filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers =
    filter(numbers, (Integer i) -> i % 2 == 0);
```

Czy to nie wspaniałe? Udało nam się znaleźć optymalny punkt pomiędzy elastycznością i zwięzłością, co było nieosiągalne przed wydaniem Javy 8!

2.4 *Przykłady praktyczne*

Teraz wiemy już, że parametryzacja zachowania jest przydatnym wzorcem pozwalającym na łatwe przystosowanie kodu do zmieniających się wymagań. Wzorzec ten pozwala nam enkapsulować konkretne zachowanie (fragment kodu) i sparametryzować działanie metod poprzez przekazanie i wykorzystanie tych zachowań (np. różnych predykatów dla obiektu `Apple`). Wcześniej wspomnieliśmy, że to podejście jest podobne do wzorca projektowego Strategia. Być może korzystałeś już z tego wzorca. Wiele metod dostępnych w API języka Java może być parametryzowanych z wykorzystaniem różnych zachowań. Metody te są często wykorzystywane z anonimowymi klasami. Zaprezentujemy teraz cztery przykłady, które pomogą Ci utrwalić ideę przekazywania kodu: sortowanie z użyciem komparatora, wykonywanie bloku kodu z użyciem interfejsu `Runnable`, zwracanie wyniku z zadania z użyciem `Callable` oraz obsługa zdarzeń graficznego interfejsu użytkownika.

2.4.1 *Sortowanie z użyciem komparatora*

Sortowanie istniejącej kolekcji jest stale powracającym zadaniem programistycznym. Załóżmy przykładowo, że nasz farmer chce, abyśmy posortowali inwentarz jabłek według ich wagi. Albo też zmienia on swoje zdanie i chce, żebyśmy posortowali jabłka według ich koloru. Brzmi znajomo? Tak, musimy w jakiś sposób zaprezentować i użyć różnych zachowań sortowania, aby dostosować się do zmieniających się wymagań.

Od czasu wydania Javy 8 każda lista (`List`) dostarczana jest z metodą `sort` (moglibyśmy również użyć tutaj metody `Collections.sort`). Zachowanie metody `sort` może zostać sparametryzowane za pomocą obiektu `java.util.Comparator`, którego interfejs jest następujący:

```
// java.util.Comparator
public interface Comparator<T> {
      int compare(T o1, T o2);
}
```

Możemy więc utworzyć różne zachowania dla metody `sort` poprzez utworzenie implementacji ad hoc interfejsu `Comparator`. Na przykład, możemy go użyć do posortowania inwentarza rosnąco według wag przy użyciu klasy anonimowej:

```
inventory.sort(new Comparator<Apple>() {
     public int compare(Apple a1, Apple a2) {
         return a1.getWeight().compareTo(a2.getWeight());
     }
});
```

Jeśli farmer nagle zmieni zdanie co do sposobu sortowania jabłek, będziemy mogli na szybko utworzyć taki komparator, który pasował będzie do nowych wymagań, a następnie

przekazać go do metody `sort`! Szczegóły dotyczące implementacji sortowania są więc wyabstrahowane. W przypadku użycia wyrażenia lambda może to wyglądać następująco:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

Nie przejmuj się na razie tą nową składnią. Kolejny rozdział szczegółowo wyjaśnia sposób tworzenia i wykorzystywania wyrażeń lambda.

2.4.2 *Wykonywanie bloku kodu z użyciem interfejsu Runnable*

Wątki w Javie sprawiają, że blok kodu może być wykonywany równolegle względem reszty programu. Ale w jaki sposób możemy wskazać wątkowi blok kodu, który chcemy uruchomić? Każdy z kilku wątków może wykonywać inny kod. Potrzebny jest więc jakiś sposób na reprezentowanie fragmentu kodu do późniejszego wykonania. Do czasu wydania języka Java 8 tylko obiekty mogły być przekazywane do konstruktora obiektu `Thread`, tak więc typowym nieporadnym wzorcem użycia było przekazanie anonimowej klasy zawierającej metodę `run`, która zwraca `void` (tj. nie zwraca żadnego rezultatu). Takie anonimowe klasy implementują interfejs `Runnable`:

W Javie interfejs `Runnable` możemy wykorzystać do reprezentowania bloku kodu do wykonania. Zwróćmy uwagę, że kod nie zwraca żadnego rezultatu (stąd też typ zwracany to `void`):

```
// java.lang.Runnable
public interface Runnable {
      void run();
}
```

Możemy użyć tego interfejsu do tworzenia wątków z wybranym przez nas zachowaniem, jak w poniższym przykładzie:

```
Thread t = new Thread(new Runnable() {
     public void run() {
         System.out.println("Hello world");
     }
});
```

Jednak od momentu wydania Javy 8 możemy po prostu użyć wyrażenia lambda, tak więc odwołanie do obiektu `Thread` może teraz wyglądać tak:

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

2.4.3 *Zwracanie rezultatu z użyciem interfejsu Callable*

Być może jesteś zaznajomiony z abstrakcją `ExecutorService`, wprowadzoną do języka Java 5. Interfejs `ExecutorService` rozdziela sposób, w jaki zadania są podsyłane i wykonywane. Dzięki wykorzystaniu interfejsu `ExecutorService` możemy przesłać określone zadanie do puli wątków, dzięki czemu jego rezultat zostanie zachowany w rezultacie `Future`. Nie