Helion

Java™ Przewodnik dla początkujących

Twórz, kompiluj i uruchamiaj nowoczesne programy pisane w Javie

Wydanie IX

WYDANIE ZAKTUALIZOWANE I ROZSZERZONE

Herbert Schildt

McGraw Hill

Tytuł oryginału: Java: A Beginner's Guide, Ninth Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-0479-8

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/javpp9
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/javpp9.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ

1

Podstawy Javy

W tym rozdziale poznasz:

- historię i filozofię języka Java,
- znaczenie języka Java dla Internetu,
- istotę kodu bajtowego,
- terminologię związaną z Javą,
- podstawowe zasady programowania obiektowego,
- sposoby tworzenia, kompilowania i wykonywania prostych programów w języku Java,
- zmienne,
- zastosowanie instrukcji sterujących `if` i `for`,
- tworzenie bloków kodu,
- zasady tworzenia tekstu programu (pozycje instrukcji, wcięcia),
- słowa kluczowe języka Java,
- zasady tworzenia identyfikatorów Java.

Niewiele technologii wywarło na świat komputerów tak wielki wpływ jak język Java. Jego powstanie w początkowym okresie ery Internetu ukształtowało jego nowoczesną postać, i to zarówno po stronie klienta, jak i serwera. Nowoczesne cechy Javy zapewniły postęp zarówno pod względem sztuki, jak i nauki programowania, wyznaczając także nowe standardy projektowania języków programowania. Kultura myślenia z wyprzedzeniem, jaka wykształciła się wokół tego języka, gwarantuje, że będzie on aktywny i żywy oraz że będzie się dostosowywał do błyskawicznych zmian, jakie często występują w świecie komputerów.

Choć jako język programowania Java jest często kojarzona z programowaniem aplikacji internetowych, jej możliwości w żadnym razie nie ograniczają się do tego obszaru zastosowań. Java jest potężnym, kompletnym językiem programowania ogólnego przeznaczenia. Zatem dla wszystkich zaczynających przygodę z programowaniem będzie stanowić doskonały wybór. Co więcej, w dzisiejszych czasach bycie profesjonalnym programistą wiąże się z umiejętnością programowania w Javie, język ten bowiem jest wyjątkowo ważny. Dzięki tej książce zdobędziesz podstawowe umiejętności, które pozwolą Ci opanować Javę.

Zadaniem niniejszego rozdziału jest wprowadzenie czytelników w świat Javy, przedstawienie jej historii, filozofii projektowania programów oraz kilku najważniejszych cech języka. Największą trudnością związaną z nauką każdego języka programowania jest to, że żaden jego element nie pozostaje w izolacji. Każdy element języka działa w połączeniu z innymi. Powiązania te są szczególnie widoczne w Javie i trudno jest omawiać poszczególne aspekty tego języka, nie nawiązując do innych. Aby poradzić

sobie z tym problemem, w tym rozdziale omówię zwięźle kilka podstawowych cech języka, takich jak ogólna postać programów, niektóre z podstawowych struktur sterujących oraz operatorów. Nie będę zagłębiać się w szczegóły, lecz raczej skoncentruję się na ogólnych koncepcjach wspólnych wszystkim programom w Javie.

Historia i filozofia Javy

Zanim będzie można w pełni docenić unikalne cechy języka Java, trzeba zrozumieć przyczyny, które doprowadziły do jego powstania, filozofię programowania, jaką reprezentuje, oraz kluczowe aspekty jego projektu. Podczas lektury kolejnych rozdziałów tej książki będzie można zauważyć, że wiele aspektów Javy jest bezpośrednim lub pośrednim efektem historycznych sił, które ukształtowały ten język. Dlatego też zasadnym jest rozpoczęcie naszej prezentacji Javy od pokazania jej związków z szerszym uniwersum programowania.

Pochodzenie Javy

Autorami języka Java są James Gosling, Patrick Naughton, Chris Warth, Ed Frank i Mike Sheridan, którzy opracowali go w Sun Microsystems w 1991 roku. Początkowo język ten nosił roboczą nazwę „Oak", ale w 1995 roku zmieniono ją na „Java". Co nieco zaskakujące, początkowo przeznaczeniem języka nie był wcale Internet! Główną motywacją dla Javy była potrzeba istnienia języka programowania umożliwiającego tworzenie aplikacji przenośnych na różne platformy, głównie konsumenckie urządzenia elektroniczne takie jak tostery, kuchenki mikrofalowe czy piloty zdalnego sterowania. Jak łatwo się domyślić, w urządzeniach tych stosowane są procesory najróżniejszych typów. W owym czasie jednak większość języków programowania zaprojektowano z myślą o kompilacji kodu źródłowego programów do kodu maszynowego wykonywanego na określonych rodzajach procesorów. Weźmy na przykład język C++.

Chociaż teoretycznie programy napisane w tym języku można skompilować dla każdego typu procesora, to w praktyce wymaga to posiadania kompilatora specjalnie stworzonego dla konkretnego typu procesora. Jednak tworzenie kompilatorów to zajęcie wymagające wiele czasu i pracy. Poszukując lepszego rozwiązania, Gosling i jego koledzy zaprojektowali przenośny język umożliwiający tworzenie kodu wykonywanego na różnych procesorach w różnych środowiskach. W ten sposób narodziła się Java.

Mniej więcej w tym samym czasie, gdy dopracowywano szczegóły Javy, pojawił się kolejny czynnik, który miał zadecydować o jej powodzeniu. Oczywiście chodzi tutaj o rozwój sieci World Wide Web. Gdyby nie on, Java mogłaby pozostać wprawdzie użytecznym, ale mało znanym narzędziem tworzenia oprogramowania elektroniki konsumenckiej. Rozwój World Wide Web wysunął ją jednak na pierwszy plan języków programowania, ponieważ również wymagał tworzenia przenośnych programów.

Większość programistów wcześnie przekonuje się, że przenośne programy są równie rzadkie co pożądane. Chociaż poszukiwania sposobu tworzenia efektywnych i przenośnych (działających na różnych platformach) programów są prawie tak stare jak historia samego programowania, to jednak z czasem ustąpiły one pola innym, bardziej palącym problemom. Dopiero rozwój Internetu i WWW spowodował, że zagadnienie przenośności powróciło ze zdwojoną siłą. Pamiętajmy, że Internet jest w swej istocie rozproszonym środowiskiem przetwarzania, na które składają się komputery najróżniejszych typów, działające pod kontrolą różnych systemów operacyjnych na różnych procesorach.

Problem przenośności odzyskał utracony priorytet. W 1993 roku dla członków zespołu pracującego nad Javą stało się jasne, że problemy przenośności związane z tworzeniem oprogramowania elektroniki konsumenckiej będą w równej mierze trapić twórców kodu działającego w Internecie. Dlatego też zdecydowano się skoncentrować dalszy rozwój języka na tym drugim zagadnieniu. Zatem chociaż u początków Javy legła potrzeba stworzenia języka umożliwiającego programowanie niezależne od konkretnej platformy, to o jej ostatecznym sukcesie zadecydował rozwój Internetu.

Java a języki C i C++

Historia języków programowania nie składa się z odizolowanych zdarzeń. To raczej nieprzerwane kontinuum, w którym każdy z nowych języków został w mniejszym lub większym stopniu oparty na swoich

poprzednikach. Java nie jest pod tym względem wyjątkiem. Dlatego nim przejdziemy dalej, warto zrozumieć, w którym miejscu rodzinnego drzewa języków programowania jest miejsce Javy.

Dwoma językami stanowiącymi bezpośrednich przodków Javy są C i C++. Jak zapewne wiesz, C oraz C++ są jedynymi z najważniejszych języków programowania, które są powszechnie używane od początków swojego istnienia. Z języka C Java dziedziczy składnię, a z języka C++ zaadaptowano model programowania obiektowego. Związki Javy z tymi językami są ważne z wielu powodów. Po pierwsze, w czasie kiedy powstawała Java, wielu programistów znało składnię języka C/C++. Dzięki znajomej składni dla programistów, którzy wcześniej używali C/C++, opanowanie Javy nie stanowiło większego problemu. Oznaczało to, że Java mogła zostać błyskawicznie wykorzystana przez rzeszę istniejących programistów, co z kolei znacząco ułatwiło powszechne zaakceptowanie jej w programistycznej społeczności.

Po drugie, projektanci Javy nie musieli ponownie „wymyślać koła". Zamiast tego udoskonalili jedynie istniejący, udany sposób programowania. Era nowoczesnego programowania zaczęła się właśnie od języka C. Po nim nastąpił C++, a później Java. Korzystając z dorobku tej tradycji, Java dostarcza potężnego, spójnego logicznie środowiska programowania, które wykorzystuje najlepsze osiągnięcia poprzedników, dodając do nich nowe możliwości związane z wykorzystaniem środowisk internetowych i zapewniając tym samym postęp w sztuce programowania. Co najważniejsze, ze względu na podobieństwa C, C++ i Java definiują wspólny szkielet koncepcyjny używany przez profesjonalnych programistów. Dzięki temu przechodząc do programowania w kolejnym języku, nie natrafiają oni na poważniejsze przeszkody.

Z językami C i C++ łączy Javę jeszcze jedna wspólna cecha: została ona zaprojektowana, przetestowana i ulepszona przez prawdziwych programistów-praktyków. U podstaw tego języka leżą potrzeby i oczekiwania jego twórców. Nie ma lepszego sposobu na stworzenie doskonałego praktycznego języka programowania.

I jeszcze jedna, ostatnia sprawa: choć języki C++ i Java są ze sobą powiązane, zwłaszcza pod względem wsparcia dla programowania obiektowego, to jednak *nie można* uznać Javy jedynie za „internetową wersję języka C++". Javę wiele różni od C++, tak na płaszczyźnie praktycznej, jak i filozoficznej. Co więcej, Java *nie jest* także rozszerzoną wersją C++. Przykładowo: nie jest zgodna z językiem C++ ani w dół, ani w górę. Zadaniem Javy jest rozwiązanie pewnego zbioru problemów, a języka C++ innego. Dlatego też oba języki będą jeszcze długo istnieć obok siebie.

Wpływ Javy na Internet

Internet umożliwił Javie wysunięcie się na czoło języków programowania. Z kolei i Java wywarła wpływ na rozwój Internetu. Nie tylko uprościła tworzenie programów działających w sieci, ale również doprowadziła do powstania nowej kategorii programów zwanych apletami, które zmieniły sposób postrzegania treści dostępnych w Internecie. Java stanowi również odpowiedź na najpoważniejsze problemy związane z Internetem: kwestie przenośności i bezpieczeństwa. Przyjrzyjmy się zatem bliżej tym zagadnieniom.

Java uprościła programowanie aplikacji internetowych

Java ułatwiła programowanie aplikacji internetowych na wiele sposobów. Prawdopodobnie najważniejszym z nich było zapewnienie możliwości tworzenia przenośnych, wieloplatformowych programów. Niemal równie ważne było wsparcie dla programowania sieciowego, jakie oferuje ten język. Jego biblioteka, udostępniająca gotowe do użycia możliwości funkcjonalne, zapewniła programistom możliwość tworzenia programów korzystających z Internetu na wiele różnych sposobów. Oprócz tego Java udostępnia możliwości rozpowszechniania programów przez Internet. Choć szczegółowe przedstawienie tych zagadnień i rozwiązań wykracza poza ramy tematyczne tej książki, trzeba wiedzieć, że wsparcie Javy dla wykorzystania Internetu stało się jednym z kluczowych czynników błyskawicznego wzrostu jej popularności.

Aplety Javy

W początkowym okresie po udostępnieniu Javy jedną z najbardziej fascynujących możliwości były aplety. Aplet jest specjalnym rodzajem programu w języku Java zaprojektowanym do przesyłania w sieci Internet i automatycznego wykonania w przeglądarce WWW. Jeśli użytkownik kliknie hiperłącze zawierające aplet, zostanie on automatycznie załadowany i uruchomiony przez przeglądarkę. Aplety były pomyślane jako niewielkie programy, wykorzystywane przeważnie do wyświetlania danych udostępnianych przez serwer, obsługi danych wprowadzanych przez użytkownika lub lokalnego wykonywania prostych funkcji, takich jak kalkulator kredytowy. Zastosowanie apletów umożliwia zatem przeniesienie określonej funkcjonalności z serwera na klienta.

Powstanie apletów zrewolucjonizowało programowanie w Internecie, ponieważ w tamtym czasie doprowadziło do poszerzenia spektrum obiektów, które mogą przemieszczać się w cyberprzestrzeni. Obiekty transmitowane pomiędzy serwerem i klientem możemy podzielić na dwie podstawowe kategorie: pasywne informacje i dynamiczne, aktywne programy. Na przykład czytając pocztę elektroniczną, przeglądamy pasywne dane. Gdy ładujemy tradycyjny program z Internetu, to jego kod również stanowi pasywne dane, aż do momentu, w którym zostanie uruchomiony. Natomiast aplet stanowi dynamiczny, automatycznie uruchamiany program. Pełni on funkcję aktywnego agenta na komputerze klienta, choć jest dostarczany przez serwer.

W początkowym okresie wykorzystywania języka Java aplety stanowiły jedno z jego kluczowych zastosowań. Stanowiły ilustrację jego możliwości i korzyści, jakie zapewnia, znacząco uatrakcyjniały także strony WWW i pozwalały programistom na poznawanie pełnego zakresu możliwości zapewnianych przez Javę. Choć jest całkiem możliwe, że aplety wciąż są używane, to jednak wraz z upływem lat ich znaczenie zmalało, a jak już niebawem wyjaśnię, w JDK 9 zaczęto je wycofywać, a w JDK 11 wsparcie dla apletów zostało całkowicie usunięte.

Ekspert odpowiada

Pytanie: Czym jest język C# i jak się on ma do Javy?

Odpowiedź: Kilka lat po udostępnieniu Javy firma Microsoft opracowała i stworzyła język C#. To ważne, gdyż jest on blisko związany z Javą. W rzeczywistości wiele cech C# to bezpośrednie odpowiedniki cech Javy. Zarówno Java, jak i C# stosują tę samą ogólną składnię bazującą na języku C++, dysponują wsparciem dla programowania rozproszonego i używają podobnego modelu obiektowego. Oczywiście pomiędzy tymi dwoma językami występują także pewne różnice, niemniej jednak ogólna postać i charakter obu są bardzo podobne. Oznacza to, że jeśli już znasz C#, to nauka Javy przyjdzie Ci z dużą łatwością. I odwrotnie: jeśli planujesz w przyszłości poznać C#, to znajomość Javy okaże się bardzo przydatna.

Bezpieczeństwo

Niezależnie od tego, jak pożądane i dynamiczne są programy sieciowe, stwarzają one poważne problemy związane z bezpieczeństwem i przenośnością. Oczywiście nie można dopuścić, by programy, które są automatycznie pobierane i uruchamiane na komputerze klienta, wyrządziły na nim jakieś szkody. Oprócz tego taki program musi mieć możliwość działania w wielu różnych środowiskach i systemach operacyjnych. Jak się przekonasz, Java rozwiązuje te problemy w efektywny i wydajny sposób. Przyjrzyjmy się każdemu z tych zagadnień, zaczynając od bezpieczeństwa.

Pobierając z sieci program, narażamy się na niebezpieczeństwo, ponieważ jego kod może zawierać wirusa, konia trojańskiego lub inny szkodliwy kod. U źródła problemu leży fakt, że program wirusa ma możliwość wykonania szkodliwych działań, ponieważ uzyskał nieautoryzowany dostęp do zasobów systemu. Przykładowo: program wirusa może zbierać prywatne informacje, takie jak numery kart kredytowych, stany kont bankowych czy hasła, przeszukując zawartość lokalnego systemu plików Twojego komputera. Aby programy mogły być bezpiecznie ładowane z sieci i wykonywane na komputerze klienta, należało uniemożliwić im przeprowadzanie wspomnianych ataków.

Java umożliwia taką ochronę poprzez ograniczenie aplikacji do środowiska wykonywania Javy i uniemożliwienie jej dostępu do innych elementów komputera. (Wkrótce zobaczysz, w jaki sposób się to odbywa). Możliwość bezpiecznego pobierania aplikacji bez obaw o potencjalne wyrządzenie

szkód czy naruszenie bezpieczeństwa była czynnikiem, który znacząco przyczynił się do początkowego sukcesu Javy.

Przenośność

Przenośność jest podstawowym aspektem działania programów w Internecie, ponieważ do sieci tej przyłączone są najróżniejsze typy komputerów działających pod kontrolą najróżniejszych systemów operacyjnych. Jeśli program w języku Java ma działać na dowolnym komputerze połączonym z Internetem, to musi istnieć sposób umożliwiający jego działanie w systemach operacyjnych różnych typów. Innymi słowy, konieczny jest mechanizm pozwalający na wczytywanie i wykonywanie aplikacji na przeróżnych procesorach, w różnych systemach operacyjnych oraz przeglądarkach. Mało praktycznym rozwiązaniem tego problemu byłoby stworzenie różnych wersji aplikacji, przeznaczonych dla różnych komputerów. Aby aplikacja miała sens, ten *sam* kod musi działać na *wszystkich* komputerach. Dlatego też należało stworzyć mechanizm generowania przenośnego kodu wykonywalnego. Jak wkrótce zobaczysz, ten sam mechanizm pozwala również zapewnić bezpieczeństwo wykonywania takiego kodu.

Magiczny kod bajtowy

Kluczem do rozwiązania opisanych problemów bezpieczeństwa i przenośności programów Java jest to, że kompilator tego języka nie generuje kodu wykonywalnego. Zamiast niego tworzony jest kod bajtowy. Kod bajtowy jest zoptymalizowanym zbiorem instrukcji przeznaczonych do wykonania w środowisku Java, zwanym **maszyną wirtualną Java** (*Java Virtual Machine*, w skrócie *JVM*), która wchodzi w skład środowiska uruchomieniowego Javy (*Java Runtime Environment*, w skrócie *JRE*). Oryginalną maszynę JVM zaprojektowano w zasadzie jako **interpreter kodu bajtowego**. Stanowi to pewną niespodziankę, ponieważ wiele nowoczesnych języków programowania jest wyłącznie kompilowanych do kodu wykonywalnego i przeznaczonych dla konkretnych rodzajów procesorów, głównie ze względów efektywnościowych. Jednak w przypadku programów w języku Java fakt ich wykonywania przez maszynę wirtualną umożliwia rozwiązanie opisanych problemów związanych z pobieraniem i uruchamianiem kodu z Internetu. Oto dlaczego.

Kompilowanie programów w Javie do kodu bajtowego ułatwia ich wykonywanie w różnych środowiskach, ponieważ jedynie JRE (zawierające wirtualną maszynę Javy) trzeba zaimplementować osobno dla każdej platformy. Gdy środowisko uruchomieniowe jest już dostępne w danym systemie, mogą w nim działać dowolne programy napisane w języku Java. Chociaż JRE różnią się szczegółami implementacji w różnych systemach, to wszystkie wykonują ten sam kod bajtowy Java. Gdyby programy Java były kompilowane do kodu wykonywalnego, to dla różnych komputerów w Internecie musiałyby istnieć różne wersje każdego programu. Takie rozwiązanie nie miałoby większego sensu. Dlatego wykonywanie kodu bajtowego przez maszynę wirtualną jest najprostszym sposobem osiągnięcia rzeczywistej przenośności programów.

Wykonywanie programów Java przez maszynę wirtualną pomaga również zapewnić bezpieczeństwo. Ponieważ wirtualna maszyna Javy dysponuje pełną kontrolą, może zarządzać realizacją programu. JVM mogła zatem utworzyć środowisko wykonawcze o ograniczonych możliwościach, nazywane **piaskownicą** (ang. *sandbox*), zawierające wykonywany program i uniemożliwiające mu uzyskanie nieograniczonego dostępu do komputera. Dodatkowo bezpieczeństwo zapewniają również pewne restrykcje istniejące w samym języku Java.

W ogólnym przypadku program interpretowany działa wolniej niż ten sam program skompilowany do kodu wykonywalnego. Jednak w przypadku programów Java różnica ta jest niewielka. Ponieważ kod bajtowy został zoptymalizowany, maszyna wirtualna wykonuje go szybciej, niż moglibyśmy się spodziewać.

Chociaż język Java zaprojektowano z myślą o interpretowanym wykonywaniu programów przez maszynę wirtualną, nic nie stoi na przeszkodzie, aby dokonać kompilacji kodu bajtowego do kodu wykonywalnego dla poprawy efektywności wykonywania programu. Dlatego wkrótce po udostępnieniu pierwszych wersji Javy pojawiła się technologia HotSpot JVM. HotSpot zawiera kompilator JIT (*just-in-time*) kodu bajtowego. Kompilator JIT stanowi część maszyny wirtualnej i umożliwia w czasie rzeczywistym kompilację wybranych fragmentów kodu bajtowego do kodu wykonywalnego jeden po drugim i na żądanie.

Należy zauważyć, że kompilator JIT kompiluje kod bajtowy dopiero, gdy jest to konieczne, tuż przed jego wykonaniem. Co więcej, nie wszystkie sekwencje kodu bajtowego są kompilowane, a jedynie te, w przypadku których kompilacja poprawi efektywność działania. Pozostały kod jest jedynie interpretowany. Mimo to zastosowanie kompilatora JIT daje znaczącą poprawę efektywności działania programów. Zachowuje przy tym wszystkie zalety związane z bezpieczeństwem i przenośnością, ponieważ programy nadal wykonywane są pod kontrolą maszyny wirtualnej.

I jeszcze jedna informacja: w Javie prowadzone były eksperymenty z kompilatorem **ahead-of-time** — z wyprzedzeniem. Miał on zapewniać możliwość kompilacji kodów bajtowych do kodu natywnego jeszcze *przed* jego wykonaniem przez JVM, a nie w trakcie tego wykonywania. Niektóre poprzednie wersje JDK zawierały eksperymentalny kompilator tego rodzaju, jednak w JDK 17 został on usunięty. Ta kompilacja z wyprzedzeniem jest wyspecjalizowaną możliwością, która nie zastępuje opisanego wcześniej, tradycyjnego modelu wykonywania kodu stosowanego w Javie. Ze względu na wysoce wyspecjalizowany charakter nie jest to rozwiązanie stosowane podczas nauki Javy i dlatego nie będę go opisywał w tej książce.

Ekspert odpowiada

Pytanie: Słyszałem, że istnieje specjalny rodzaj programów Java zwanych serwletami. Czym się one charakteryzują?

Odpowiedź: Serwlet jest niewielkim programem wykonywanym na serwerze. Serwlet dynamicznie poszerza możliwości funkcjonalne serwera. Mimo że aplikacje klienckie są tak użyteczne, stanowią one jedynie połowę równania klient-serwer. Wkrótce po udostępnieniu Javy stało się jasne, że technologia ta przyda się również na serwerach. W efekcie powstały serwlety. W ten sposób Java rozszerzyła swój zasięg na oba końce połączenia klient-serwer. Zagadnienie tworzenia i stosowania serwletów oraz ogólnie programowania po stronie serwera wykracza poza zakres niniejszej książki, ale z pewnością warto zainteresować się nim, kiedy już lepiej poznasz język Java.

Coś więcej niż aplety

Minęły już dwie dekady od momentu wprowadzenia pierwszej wersji Javy. W ciągu tych lat w języku tym wprowadzono wiele zmian. W czasie kiedy Java została udostępniona, Internet był nowym i ekscytującym wynalazkiem, przeglądarki WWW były dynamicznie rozwijane i usprawniane, smartfony, w ich obecnej, nowoczesnej postaci jeszcze nie istniały, a na wszechobecność komputerów trzeba było poczekać jeszcze kilka lat. Jak można się spodziewać, także język Java ulegał zmianom, a wraz z nim sposoby jego stosowania.

Jak już wyjaśniłem, w początkowym okresie istnienia Javy aplety stanowiły jedno z jej kluczowych zastosowań. Nie tylko podnosiły atrakcyjność stron WWW, lecz urosły do roli wizytówki zwiększającej prestiż i popularność języka. Niemniej jednak działanie apletów zależało od wtyczek rozszerzających możliwości przeglądarek WWW. A zatem by aplet mógł działać, przeglądarka musiała go obsługiwać. W ostatnich kilku latach wsparcie dla wtyczek Javy przeznaczonych dla przeglądarek WWW staje się coraz mniejsze. Mówiąc bez ogródek: bez wsparcia dla tych wtyczek aplety tracą rację bytu. Z tego względu po wprowadzeniu JDK 9 technologię apletów uznano za niezalecaną i zaczęto wycofywać się ze wsparcia apletów w przeglądarkach. W terminologii języka Java oznacza to, że dane rozwiązanie wciąż jest dostępne, lecz uważa się je za przestarzałe. Ten proces wycofywania zakończył się wraz z wprowadzeniem JDK 11, gdzie wsparcie dla apletów zostało całkowicie usunięte. Począwszy od JDK 17, cały Applet API jest uznawany za *przestarzały i przeznaczony do usunięcia*, co oznacza, że w przyszłości zostanie usunięty z JDK.

W ramach ciekawostki warto wspomnieć, że kilka lat po pojawieniu się Javy udostępniono rozwiązanie stanowiące alternatywę dla apletów. Java Web Start, bo tak nazywa się to rozwiązanie, umożliwia dynamiczne pobieranie aplikacji ze stron WWW. Java Web Start był mechanizmem uruchomieniowym przydatnym w szczególności w przypadku dużych aplikacji, które nie nadawały się do zaimplementowania w formie apletów. Różnica pomiędzy apletami a aplikacjami Java Web Start polega na tym, że aplikacje te działają samodzielnie, a nie wewnątrz przeglądarek WWW. Dlatego wyglądają niemal jak

„normalne" aplikacje. Niemniej jednak do ich działania konieczne jest zainstalowanie na komputerze niezależnego środowiska uruchomieniowego Javy (JRE) wspierającego mechanizm Java Web Start. W JDK 11 wsparcie dla Java Web Start zostało usunięte.

Zważywszy na to, że w najnowszej wersji Javy nie są dostępne ani aplety, ani Java Web Start, można się zastanawiać, jakiego innego mechanizmu używać do wdrażania aplikacji pisanych w tym języku. W czasie kiedy przygotowywałem tę książkę, jedną z opcji było użycie narzędzia **jlink** dodanego w JDK 9. Pozwala ono tworzyć kompletne obrazy uruchomieniowe, zawierające wszystkie składniki niezbędne do uruchomienia programu, w tym także JRE. Kolejną częścią odpowiedzi jest narzędzie **jpackage**. Zostało ono dodane w JDK 16 i może być używane do tworzenia aplikacji gotowych do zainstalowania. Jak można się domyślić, wdrażanie jest zaawansowanym zagadnieniem wykraczającym poza ramy tematyczne niniejszej książki. Na szczęście, aby korzystać z tej książki, nie trzeba zaprzątać sobie nim głowy, gdyż wszystkie prezentowane w niej programy przykładowe można uruchamiać bezpośrednio na własnym komputerze. Nie trzeba ich wdrażać za pośrednictwem Internetu.

Szybszy harmonogram udostępniania

Nie tak dawno temu w Javie wprowadzono także inną, istotną zmianę, choć nie dotyczy ona samego języka ani środowiska uruchomieniowego. Chodzi o sposób udostępniania kolejnych wersji Javy. W przeszłości kolejne główne wersje języka były udostępniane co mniej więcej dwa lata lub nawet w dłuższych odstępach czasu. Niemniej jednak po udostępnieniu JDK 9 ten okres czasu pomiędzy wprowadzaniem kolejnych głównych wersji Javy uległ skróceniu. Obecnie oczekuje się, że główne wersje języka będą się pojawiać według ściśle określonego harmonogramu co sześć miesięcy.

Każda z tych głównych wersji języka, określanych jako *feature release*, ma zawierać nowe możliwości, które będą gotowe w momencie udostępniania. Taka zwiększona *częstość udostępniania* sprawia, że programiści używający Javy będą mogli szybciej korzystać z nowych możliwości i usprawnień języka. Najprościej rzecz ujmując, szybszy harmonogram udostępniania będzie bardzo korzystny dla programistów Javy.

Przewiduje się, że w trzyletnich odstępach będą udostępniane wersje LTS Javy, czyli wersje ze *wsparciem długoterminowym*. Wydanie LTS będzie wspierane (a tym samym pozostanie aktualne) przez okres dłuższy niż sześć miesięcy. Pierwszym wydaniem LTS było JDK 11. Drugim wydaniem LTS było JDK 17, i to właśnie ono jest tematem niniejszego, zaktualizowanego wydania książki. Ze względu na stabilność, jaką oferuje wydanie LTS, jest prawdopodobne, że zestaw jego możliwości zdefiniuje podstawę funkcjonalności języka Java na wiele następnych lat. Najnowsze informacje dotyczące wydań LST i harmonogramu ich udostępniania można znaleźć w witrynie firmy Oracle.

Obecnie nowe wersje Javy są zaplanowane na marzec i wrzesień każdego roku. W efekcie JDK 10 udostępniono w marcu 2018 roku, czyli sześć miesięcy po JDK 9. Kolejna wersja, JDK 11, pojawiła się we wrześniu 2018 roku. Było to wydanie LTS. Następnie, w marcu 2019 roku, wydano JDK 12, a później, we wrześniu 2019 roku — JDK 13, i tak dalej. W chwili pisania tego tekstu najnowszym wydaniem jest JDK 17, które jest wydaniem LTS. Oczekuje się, że następne wersje będą się pojawiać co kolejne sześć miesięcy. Oczywiście informacje o najnowszych wersjach Javy można znaleźć w harmonogramie udostępniania.

W czasie kiedy przygotowywałem tę książkę, zapowiadano wprowadzenie do języka Java kilku nowości. Ze względu na szybszy harmonogram udostępniania jest bardzo prawdopodobne, że kilka z nich zostanie wprowadzonych w ciągu paru najbliższych lat. Warto zatem dokładnie analizować informacje o nowościach wprowadzanych w każdej nowej wersji języka. Zapowiada się naprawdę ekscytujący okres dla programistów Javy!

Terminologia Javy

Wprowadzenie do języka Java warto zakończyć wyjaśnieniem terminologii często pojawiającej się w jego kontekście. Chociaż podstawowymi czynnikami sprawczymi powstania Javy były przenośność i bezpieczeństwo, to istotną rolę w nadaniu jej ostatecznego kształtu odegrały również inne czynniki. Ich podsumowanie dokonane przez zespół Java zawiera poniższa tabela 1.1.

Tabela 1.1. *Terminologia Javy*

Termin	Znaczenie
Prostota	Java dysponuje zwięzłym i spójnym zbiorem cech, które ułatwiają jej naukę i wykorzystanie.
Bezpieczeństwo	Java dostarcza środków zapewniających tworzenie bezpiecznych aplikacji internetowych.
Przenośność	Programy Java mogą być wykonywane w dowolnych środowiskach, jeśli tylko istnieje w nich maszyna wirtualna Java.
Obiektowość	Java wykorzystuje nowoczesną filozofię programowania obiektowego.
Niezawodność	Java wspiera niezawodność programów poprzez ścisłe przestrzeganie zgodności typów oraz kontrolę kodu podczas wykonywania.
Wielowątkowość	Java zawiera zintegrowane wsparcie dla programowania wielowątkowego.
Niezależność	Java nie jest związana z określonym typem maszyny czy systemem operacyjnym.
Interpretowalność	Programy Java mogą być wykonywane na różnych platformach dzięki zastosowaniu kodu bajtowego.
Wysoka efektywność	Kod bajtowy Java został zoptymalizowany z myślą o efektywności wykonania.
Rozproszoność	Javę zaprojektowano z myślą o rozproszonym środowisku działania, jakim jest Internet.
Dynamiczność	Programy Java zawierają wystarczającą informację o typie obiektów, co pozwala dodatkowo weryfikować poprawność dostępu podczas wykonywania programu.

Programowanie obiektowe

Kluczową koncepcją języka Java jest programowanie obiektowe. Metodologia obiektowa i Java są nierozdzielnie związane i wobec tego każdy program w języku Java jest obiektowy — przynajmniej do pewnego stopnia. Ze względu na znaczenie obiektowości w Javie warto, byś najpierw ogólnie poznał podstawowe zasady programowania obiektowego, zanim napiszesz pierwszy program w tym języku. W dalszej części książki dowiesz się, w jaki sposób wykorzystać te zasady w praktyce.

Programowanie obiektowe stanowi istotny postęp w sztuce programowania. Metodologie programowania zmieniały się wielokrotnie od momentu powstania pierwszego komputera, głównie w odpowiedzi na rosnącą złożoność programów. Pierwsze komputery programowano, wprowadzając instrukcje kodu maszynowego za pomocą przełączników umieszczonych na panelu sterującym komputera. Rozwiązanie takie sprawdzało się w praktyce, dopóki objętość programów nie przekraczała kilkuset instrukcji kodu maszynowego. Wzrost objętości programów spowodował powstanie języków asemblerowych umożliwiających symboliczną reprezentację instrukcji kodu maszynowego. Dalszy wzrost objętości programów przyczynił się do powstanie języków programowania wysokiego poziomu oferujących programiście jeszcze lepsze sposoby implementowania coraz bardziej złożonych programów. Pierwszym takim językiem, który zdobył szerszą popularność, był oczywiście FORTRAN. Chociaż FORTRAN robił spore wrażenie jako pierwszy krok w dziedzinie języków wysokiego poziomu, to raczej nie był językiem zachęcającym programistów do tworzenia przejrzystego kodu.

W latach 60. ubiegłego wieku narodziło się programowanie strukturalne. Metodę tę stosuje się w językach takich jak C czy Pascal. Języki programowania strukturalnego umożliwiają dość łatwe tworzenie programów o całkiem sporym stopniu komplikacji. Języki te charakteryzuje przede wszystkim możliwość tworzenia samodzielnych podprogramów, stosowania zmiennych lokalnych, różnorakich instrukcji sterujących zamiast instrukcji skoku bezwarunkowego GOTO. Chociaż języki programowania strukturalnego stanowią potężne narzędzie, to nawet one potrafią tworzyć bariery, gdy złożoność projektu przekracza pewne granice.

Każdy milowy krok w rozwoju programowania wymagał stworzenia technik i narzędzi, które pozwoliłyby programiście poradzić sobie ze wzrastającą złożonością programów. Każde nowe podejście do programowania korzystało z najlepszych dokonań poprzedników i rozszerzało je o istotne innowacje. Zanim zaproponowano programowanie obiektowe, wiele projektów dotarło już do granic

możliwości podejścia strukturalnego. Metodologię obiektową wymyślono, aby pomóc projektantom i programistom pokonać kolejne bariery.

Programowanie obiektowe łączy najlepsze idee programowania strukturalnego z kilkoma zupełnie nowymi koncepcjami. W rezultacie zmienia się całkowicie sposób organizacji programu. W ogólnym przypadku program może być zorganizowany na jeden z dwóch sposobów: wokół kodu (co się dzieje) lub wokół danych (co się zmienia). W przypadku zastosowania samych technik programowania strukturalnego programy są zwykle zorganizowane wokół kodu. Inaczej mówiąc, w tym przypadku „kod działa na danych".

Programy obiektowe działają odwrotnie. Są zorganizowane wokół danych i obowiązuje zasada, że „dane sterują dostępem do kodu". W języku programowania obiektowego definiujemy dane oraz procedury, które mogą działać na tych danych. W ten sposób typ danych definiuje precyzyjnie operacje, które mogą być wykonywane na tych danych.

Większość języków programowania obiektowego, w tym również Java, stosuje trzy podstawowe zasady programowania obiektowego: hermetyzację, polimorfizm i dziedziczenie. Przyjrzyjmy się dokładniej każdej z nich.

Hermetyzacja

Hermetyzacja jest mechanizmem programowym wiążącym dane z kodem, który na nich operuje. Rozwiązanie takie zapobiega przypadkowym interferencjom z zewnętrznym kodem oraz niepoprawnemu użyciu danych. W językach obiektowych powiązanie kodu z danymi tworzy się w taki sposób, że powstaje samodzielna **czarna skrzynka**. Zawiera ona wszystkie niezbędne dane i kod. Z takiego połączenia danych i kodu powstają właśnie obiekty. Innymi słowy, obiekt jest konstrukcją umożliwiającą hermetyzację.

Dane i kod wewnątrz obiektu mogą pozostawać *prywatne* dla obiektu lub zostać udostępnione *publicznie*. Prywatne dane i kod są dostępne jedynie dla innych składowych tego samego obiektu. Innymi słowy, prywatne dane i kod nie są dostępne dla programu istniejącego na zewnątrz obiektu. Natomiast publiczne dane i kod, mimo że zdefiniowane wewnątrz obiektu, mogą być używane przez inne części programu. Typowo publiczne składowe obiektu są definiowane, aby utworzyć interfejs kontrolujący sposób wykorzystania prywatnych składowych obiektu.

W języku Java podstawową jednostką hermetyzacji jest **klasa**. Klasa zostanie omówiona szczegółowo w dalszej części książki, teraz jednak pomocne będzie krótkie wprowadzenie. Klasa definiuje postać obiektu. Określa dane obiektu oraz kod, który operuje na tych danych. Java używa specyfikacji klasy do tworzenia **obiektów**. Obiekty są instancjami klasy. Innymi słowy, klasa stanowi plan konstrukcji obiektów.

Kod i dane tworzące obiekt nazywamy **składowymi** klasy. W szczególności dane zdefiniowane w klasie określane są mianem **zmiennych składowych** lub **zmiennych instancji**. Kod działający na tych danych nazywamy **metodami składowymi** lub po prostu **metodami**. **Metoda** jest w języku Java określeniem podprogramu. Jeśli znasz język C/C++, to *metoda* w języku Java stanowi odpowiednik *funkcji* w języku C/C++.

Polimorfizm

Polimorfizm (z greckiego, wiele form) oznacza w programowaniu obiektowym możliwość posługiwania się pewnym zbiorem akcji za pomocą jednego interfejsu. Konkretna akcja zostaje wybrana w konkretnej sytuacji. Najprostszym przykładem polimorfizmu może być kierownica samochodu. Kierownica (czyli interfejs) wygląda zawsze tak samo niezależnie od tego, jaki mechanizm kierowania zastosowano w aucie. Kierownicy używasz zawsze w taki sam sposób niezależnie od tego czy samochód posiada zwykłą przekładnię kierowniczą, czy wyposażony jest we wspomaganie. Jeśli umiesz posługiwać się kierownicą, możesz jeździć dowolnym typem auta.

Ta sama zasada znajduje zastosowanie w programowaniu. Weźmy pod uwagę na przykład stos (rozumiany jako lista obsługiwana zgodnie ze strategią pierwszy na wejściu – ostatni na wyjściu (ang. *last-in, first-out — LIFO*). Możesz napisać program, który będzie wymagał użycia trzech typów stosu. Jeden będzie przechowywał wartości całkowite, drugi wartości zmiennoprzecinkowe, a ostatni znaki.

We wszystkich tych przypadkach algorytm implementujący stos będzie taki sam, mimo że poszczególne stosy przechowują różne dane. W języku programowania strukturalnego musiałbyś jednak utworzyć trzy różne zestawy operacji na stosie różniące się nazwami i parametrami funkcji. W języku Java dzięki polimorfizmowi wystarczy utworzyć jeden ogólny zestaw operacji na stosie, które będą działać we wszystkich trzech przypadkach. Zatem jeśli potrafisz posługiwać się w Javie jednym stosem, to możesz korzystać z najróżniejszych typów stosów.

Koncepcję polimorfizmu można przekazać w najbardziej ogólny sposób jako „jeden interfejs, wiele metod". Oznacza to możliwość zaprojektowania ogólnego interfejsu dla grupy powiązanych akcji. Polimorfizm pozwala ograniczyć złożoność programu poprzez zastosowanie tego samego interfejsu dla określenia **ogólnej klasy akcji**. Zadaniem kompilatora jest wybranie **określonej akcji** (czyli metody), którą należy zastosować w konkretnej sytuacji. Programista nie musi dokonywać tego wyboru w programie. Wystarczy, że używa ogólnego interfejsu.

Dziedziczenie

Dziedziczenie jest procesem, w którym obiekt otrzymuje właściwości innego obiektu. Mechanizm ten wspiera tworzenie hierarchicznych klasyfikacji. Jeśli się nad tym bliżej zastanowisz, to zauważysz, że większością naszej wiedzy możemy zarządzać właśnie za pomocą hierarchicznych (zstępujących) klasyfikacji. Na przykład jabłko odmiany Złota Reneta należy do ogólniejszej klasyfikacji *jabłka*, która z kolei jest częścią klasy *owoc*, która należy do ogólniejszej klasy *żywność*. Klasa *żywność* posiada określone właściwości (np. wartość odżywcza), które odnoszą się do jej klas pochodnych, w tym klasy *owoc*. Oprócz właściwości dziedziczonych po klasie *żywność* klasa *owoc* może posiadać własne, specyficzne właściwości (soczystość, słodkość itd.), które odróżniają ją od innych podklas klasy żywność. Klasa *jabłko* definiuje z kolei właściwości specyficzne dla jabłek (rośnie na drzewach, w klimacie umiarkowanym itd.). Klasa Złota Reneta dziedziczy właściwości wszystkich wymienionych klas i dodaje własne, które czynią ją unikatową.

Gdyby nie hierarchie dziedziczenia, każdy obiekt musiałby jawnie definiować całą swoją charakterystykę. Zastosowanie mechanizmu dziedziczenia powoduje, że obiekt definiuje jedynie te właściwości, które czynią go unikatowym w klasie. Pozostałe atrybuty może dziedziczyć po klasie nadrzędnej. W ten sposób mechanizm dziedziczenia sprawia, że obiekt może być traktowany jako specyficzna instancja ogólniejszej klasy.

Java Development Kit

Poznałeś już teorię leżącą u podstaw Javy, pora zatem zająć się pisaniem programów. Zanim będziesz mógł kompilować i uruchamiać te programy, musisz najpierw zainstalować Java Development Kit (JDK) na swoim komputerze. Gdy powstawała ta książka, najnowszą wersją JDK była JDK 11. Wersja ta jest używana przez Java SE 11 (SE oznacza Standard Edition). Jest to także wersja JDK opisywana w niniejszej książce. Ponieważ JDK 11 zawiera wiele nowości niedostępnych w starszych wersjach języka Java, ważne jest, abyś używał JDK 11 (lub nowszego) do kompilowania i uruchamiania programów omawianych w tej książce. (Pamiętaj, że ze względu na częstsze udostępnianie oczekuje się, że nowe wersje JDK będą się pojawiać co sześć miesięcy. Dlatego nie zdziw się, jeśli się okaże, że jest dostępna wersja JDK z większym numerem). Niemniej w zależności od środowiska roboczego na komputerze może być zainstalowana wcześniejsza wersja JDK. W takim przypadku nowsze możliwości języka Java nie będą dostępne.

Jeśli będziesz musiał zainstalować JDK na swoim komputerze, pamiętaj, że w przypadku nowych wersji języka do pobrania dostępne są zarówno Oracle JDK, jak i wersje open source — OpenJDK. Ogólnie rzecz biorąc, w pierwszej kolejności znajdź wersję JDK, którą chcesz zainstalować. Na przykład w momencie przygotowywania tej książki Oracle JDK można znaleźć na stronie *http://www.oracle.com/java/technologies/downloads/*. Jednocześnie ze strony *http://jdk.java.net* można pobrać wersję open source JDK. Po pobraniu JDK należy zainstalować na komputerze zgodnie z instrukcjami. Gdy zainstalujesz JDK, będziesz w stanie kompilować i wykonywać programy w języku Java.

JDK zawiera dwa podstawowe programy. Pierwszy z nich, `javac`, jest kompilatorem języka Java. Drugi, `java`, jest standardowym interpreterem kodu bajtowego języka Java i jest także określany jako *program do uruchamiania aplikacji*. Ważna informacja: JDK działa wyłącznie w trybie wiersza poleceń, nie jest aplikacją okienkową. Nie jest również zintegrowanym środowiskiem programistycznym (IDE).

Ekspert odpowiada

Pytanie: W książce padło stwierdzenie, że programowanie obiektowe stanowi efektywny sposób tworzenia rozbudowanych programów. Wydaje się zatem, że w przypadku niewielkich programów może wiązać się z istotnym, niepotrzebnym narzutem. Ponieważ w Javie wszystkie programy są w zasadzie obiektowe, czy rzeczywiście ma to niekorzystny wpływ na efektywność działania małych programów?

Odpowiedź: Nie. Jak się przekonasz, w przypadku małych programów obiektowe cechy języka Java są praktycznie transparentne. Chociaż Java rygorystycznie stosuje model obiektowy, pozostawiono programiście sporą swobodę co do stopnia jego wykorzystania. W przypadku małych programów ich obiektowość jest ledwie dostrzegalna. Wraz ze wzrostem ich objętości programista może bez większego wysiłku wprowadzać coraz więcej cech obiektowych.

Oprócz podstawowych narzędzi JDK uruchamianych w wierszu poleceń istnieje wiele doskonałych zintegrowanych środowisk programisty języka Java, takich jak NetBeans czy Eclipse. Są one szczególnie przydatne podczas tworzenia komercyjnych aplikacji. Możesz również używać ich do kompilacji i uruchamiania programów omawianych w tej książce. Jednak zawiera ona wskazówki tylko odnośnie do kompilowania i uruchamiania programów za pomocą JDK. Istnieje ku temu kilka ważnych powodów. Po pierwsze, JDK jest łatwo dostępne dla wszystkich czytelników. Po drugie, wskazówki dotyczące posługiwania się JDK są takie same dla wszystkich czytelników. Co więcej, w przypadku prostych programów omawianych w tej książce narzędzia JDK stanowią również najprostszy sposób ich kompilowania i uruchamiania. Jeśli zdecydujesz się używać środowiska IDE, musisz sam zapoznać się ze sposobami jego użytkowania. Trudno w tym przypadku podać zestaw ogólnych wskazówek ze względu na różnice pomiędzy poszczególnymi środowiskami IDE.

Pierwszy prosty program

Przygodę z Javą zaczniesz od skompilowania i uruchomienia krótkiego programu przedstawionego na listingu 1.1.

Listing 1.1. *Example.java*

```
/*
    Pierwszy program w języku Java

    Nazwij plik Example.java
*/
class Example {
  // Program w Javie rozpoczyna działanie od wywołania main()
  public static void main(String[] args) {
    System.out.println("Java rządzi siecią.");
  }
}
```

W tym celu musisz wykonać następujące kroki:

1. Wprowadź tekst programu.
2. Skompiluj program.
3. Uruchom go.

Wprowadzenie tekstu programu

Teksty programów omawianych w tej książce są dostępne pod adresem *https://ftp.helion.pl/przyklady/javpp9.zip*. Oczywiście równie dobrze możesz wprowadzać je samodzielnie. W takim przypadku musisz użyć odpowiedniego edytora, najlepiej możliwie prostego, a nie zaawansowanego procesora tekstu. Te

ostanie umieszczają w plikach szereg dodatkowych informacji, które nie są zrozumiałe dla kompilatora. Na platformie Windows możesz użyć na przykład programu Notatnik lub innego edytora przeznaczonego dla programistów.

W przypadku większości języków programowania możesz dowolnie wybrać nazwę pliku, w którym umieścisz kod źródłowy programu. W Javie sytuacja wygląda jednak inaczej. Pierwszą zasadą dotyczącą programów w Javie, którą musisz zapamiętać, jest to, że *nazwa pliku źródłowego ma istotne znaczenie*. W naszym przypadku nazwą pliku musi być *Example.java*. Oto dlaczego.

W języku Java plik źródłowy nosi oficjalną nazwę **jednostka kompilacji**. Jest to plik tekstowy, który zawiera (między innymi) definicję jednej lub większej liczby klas (na razie będziesz używać plików źródłowych zawierających definicję tylko jednej klasy). Kompilator języka Java wymaga, aby nazwy plików źródłowych posiadały rozszerzenie *.java*. Jak pewnie zauważyłeś, nazwą klasy definiowanej w naszym przykładzie jest również `Example`. Nie jest to przypadek. W języku Java cały kod musi należeć do klasy. Wymagane jest, aby nazwa głównej klasy była taka sama jak nazwa pliku, który zawiera jej definicję. Znaczenie ma również sposób zapisu tej nazwy za pomocą małych i wielkich liter, które są rozróżniane przez Javę. Na tym etapie zasada tworzenia nazw plików odpowiadających nazwom klas może wydawać Ci się restrykcyjna, ale przekonasz się, że ułatwia ona utrzymanie i organizację kodu programów. Co więcej, jak przekonasz się w dalszej części książki, w niektórych przypadkach jest ona wymagana.

Kompilowanie programu

Aby skompilować program *Example*, wywołaj kompilator `javac`, podając nazwę pliku źródłowego w wierszu wywołania:

```
javac Example.java
```

Kompilator `javac` utworzy plik o nazwie *Example.class* zawierający kod bajtowy naszego programu. Pamiętaj, że kod bajtowy nie jest kodem wykonywalnym. Kod bajtowy może działać jedynie na maszynie wirtualnej Java. Dlatego wynik działania kompilatora nie jest plikiem wykonywalnym.

Aby uruchomić program w języku Java, musisz wywołać interpreter kodu bajtowego, `java`. W wierszu wywołania należy przekazać mu nazwę klasy, którą powinien uruchomić:

```
java Example
```

Uruchomienie programu spowoduje wyświetlenie następującego napisu:

```
Java rządzi siecią.
```

Od tłumacza: aby programy przykładowe poprawnie wyświetlały polskie znaki diakrytyczne na konsoli, musimy podczas uruchamiania programów poinformować Javę o zastosowanym kodowaniu tych znaków. W tym celu użyjemy opcji wywołania maszyny wirtualnej `-Dfile.encoding`. W systemie Windows uruchomimy przykładowy program w następujący sposób:

```
java -Dfile.encoding=CP852 Example
```

W innych systemach może okazać się konieczne użycie innego kodowania niż CP852.

Podczas kompilowania kodu źródłowego programu w języku Java kod bajtowy każdej klasy zostaje umieszczony w osobnym pliku, którego nazwa jest taka sama jak nazwa klasy, a rozszerzeniem nazwy jest *.class*. Właśnie dlatego warto nadawać plikom źródłowym nazwę odpowiadającą nazwie klasy, którą zawierają. Wtedy nazwa pliku źródłowego będzie również odpowiadać nazwie pliku *.class*. Gdy uruchamiasz interpreter Java w pokazany powyżej sposób, podajesz mu nazwę klasy, której kod powinien wykonać. Interpreter automatycznie poszukuje pliku o takiej nazwie i rozszerzeniu *.class*. Jeśli go odnajdzie, wykonuje kod podanej przez Ciebie klasy.

Nim przejdziemy dalej, należy zwrócić uwagę, że zaczynając od JDK 11 Java udostępnia także możliwość wykonywania prostych programów bezpośrednio z poziomu kodów źródłowych, bez ich wcześniejszej, jawnej kompilacji. Technika ta, która może się przydać w niektórych sytuacjach, została opisana w dodatku C. Jednak w przypadku przykładów prezentowanych w tej książce zakładam, że będziesz używać standardowego, opisanego powyżej procesu kompilacji.


Uwaga

Gdy próbujesz skompilować program, a komputer nie może odnaleźć kompilatora `javac` (przy założeniu, że poprawnie zainstalowałeś JDK), to być może powinieneś określić ścieżkę dostępu do narzędzi pakietu JDK. W systemie Windows będzie to oznaczać konieczność dodania tej ścieżki do ścieżek definiowanych przez zmienną środowiska o nazwie `PATH`. Na przykład jeśli zainstalowałeś JDK 17 w katalogu *Program files*, to ścieżką dostępu do jego narzędzi jest *C:\Program files\Java\jdk-17\bin*. (Oczywiście będziesz musiał określić ścieżkę dostępu do plików JDK na swoim komputerze, gdyż może się ona różnić od tej podanej powyżej. Inny może być także numer wersji JDK podany w nazwie katalogu). Być może będziesz musiał zajrzeć do dokumentacji używanego systemu operacyjnego, by dowiedzieć się, jak dodać nazwę katalogu do zmiennej środowiskowej `PATH`, gdyż w różnych systemach robi się to inaczej.

Pierwszy program wiersz po wierszu

Chociaż program *Example.java* jest bardzo krótki, ma kilka kluczowych cech wspólnych wszystkim programom w Javie. Przyjrzyjmy się zatem bliżej każdej jego części.

Program zaczyna się od poniższych wierszy:

```
/*
   Oto prosty program w Javie

   Nazwij plik Example.java
*/
```

Wiersze te stanowią **komentarz**. Podobnie jak większość języków programowania, również Java pozwala na zamieszczanie uwag w kodzie źródłowym programów. Zawartość komentarza jest ignorowana przez kompilator. Zadaniem komentarza jest opis lub wyjaśnienie działania programu każdemu, kto czyta jego kod źródłowy. W tym przypadku komentarz przedstawia program i przypomina, że jego kod należy umieścić w pliku *Example.java*. Oczywiście w przypadku innych programów komentarze używane są głównie do wyjaśnienia sposobu działania pewnych części kodu lub przedstawienia ich zadania.

W języku Java możesz używać dwóch stylów komentowania. Przedstawiony powyżej to **komentarz wielowierszowy**. Ten typ komentarza musi rozpoczynać się sekwencją znaków `/*` i kończyć sekwencją `*/`. Wszystko, co umieścisz pomiędzy tymi sekwencjami, zostanie zignorowane przez kompilator. Jak sugeruje nazwa, komentarz taki może zajmować wiele wierszy w tekście programu.

Kolejny wiersz programu wygląda następująco:

```
class Example {
```

W tym wierszu słowa kluczowego `class` użyłeś, aby zadeklarować definicję nowej klasy. Jak już wspomniałem, klasa stanowi podstawową jednostkę hermetyzacji w Javie. `Example` jest w tym przypadku nazwą klasy. Definicja klasy rozpoczyna się otwierającym nawiasem klamrowym ({) i kończy odpowiadającym mu nawiasem zamykającym (}). Wszystkie elementy umieszczone pomiędzy tymi nawiasami są składowymi klasy. W tej chwili nie powinieneś jeszcze przejmować się szczegółami definicji klasy, wystarczy, że zapamiętasz, że wszystko, co robi program, musi należeć do jakiejś klasy. Z tego powodu wszystkie programy w Javie są (przynajmniej w pewnym stopniu) obiektowe.

W kolejnym wierszu znalazł się **komentarz jednowierszowy**:

```
// Program Java rozpoczyna działanie od wywołania main()
```

W ten sposób poznałeś drugi typ komentarzy w Javie. Komentarz jednowierszowy rozpoczyna się sekwencją znaków `//` i kończy wraz z końcem bieżącego wiersza. Programiści zwykle używają komentarzy wielowierszowych do umieszczania dłuższych objaśnień, a jednowierszowych do krótkich uwag, często opisujących kolejne wiersze programu.

Następny wiersz kodu wygląda następująco:

```
public static void main (String[] args) {
```

Od tego wiersza rozpoczyna się metoda `main()`. Jak już wspomniałem wcześniej, podprogramy w Javie nazywamy **metodami**. Jak sugeruje komentarz poprzedzający ten wiersz kodu, tutaj rozpocznie się działanie programu. Działanie wszystkich programów w Javie rozpoczyna się od wywołania metody

`main()`. Nie wyjaśnię teraz dokładnego znaczenia poszczególnych słów kluczowych umieszczonych w tym wierszu, gdyż wymaga to szczegółowego omówienia kilku kolejnych właściwości Javy. Ponieważ jednak wiersz ten będzie pojawiać się w tej postaci w wielu programach zamieszczonych w tej książce, przyjrzyjmy się krótko jego elementom.

Słowo kluczowe `public` stanowi **modyfikator dostępu**. Określa on dostępność składowej klasy dla innych części programu. Gdy składowa klasy jest poprzedzona modyfikatorem `public`, to jest dostępna również dla kodu spoza klasy, w której została zdefiniowana. (Przeciwieństwem modyfikatora `public` jest `private`, który uniemożliwia dostęp do składowej na zewnątrz klasy). W tym przypadku metoda `main()` musi być zadeklarowana jako `public`, ponieważ musi zostać wywołana przez kod spoza klasy podczas uruchamiania programu. Słowo kluczowe `static` umożliwia wywołanie metody `main()`, zanim zostanie utworzony jakikolwiek obiekt klasy. Jest to konieczne w tym przypadku, ponieważ metoda `main()` zostanie uruchomiona przez maszynę wirtualną, zanim zostaną utworzone jakiekolwiek obiekty. Słowo kluczowe `void` informuje kompilator, że metoda `main()` nie zwraca żadnej wartości. Jak zobaczysz później, metody mogą również zwracać różne wartości. Jeśli nie jesteś pewien, że dokładnie zrozumiałeś znaczenie tych słów kluczowych, nie przejmuj się, zostaną one omówione szczegółowo w kolejnych rozdziałach.

Jak wspomniałem, metoda `main()` zostaje wywołana w momencie uruchamiania programu Java. Jeśli program ten został uruchomiony z jakimiś parametrami, zostaną one przekazane tej metodzie za pośrednictwem zmiennych określonych wewnątrz nawiasów następujących po nazwie metody. Zmienne te nazywamy **parametrami** metody. Jeśli metoda nie ma żadnych parametrów, to za jej nazwą musimy umieścić puste nawiasy. Metoda `main()` ma jeden parametr, `String[] args`. Jest to tablica `args` zawierająca obiekty typu `String` (**tablice** są kolekcjami podobnych obiektów). Obiekty typu `String` przechowują sekwencje znaków. W tym przypadku tablica `args` będzie zawierać argumenty przekazane programowi podczas jego uruchomienia. Nasz program nie czyni z nich żadnego użytku, ale inne programy, które poznasz podczas lektury tej książki, będą ich używać.

Ostatnim znakiem w tym wierszu jest `{`. Sygnalizuje on początek ciała metody `main()`. Cały kod metody umieszczamy pomiędzy otwierającym nawiasem klamrowym i odpowiadającym mu nawiasem zamykającym.

Kolejny wiersz kodu przedstawiony został poniżej. Zwróć uwagę, że jest on umieszczony wewnątrz metody `main()`.

```
System.out.println("Java rządzi siecią");
```

Zadaniem tego wiersza jest wyświetlenie napisu `Java rządzi siecią`, po którym umieszczony zostanie znak nowego wiersza. Zadanie to wykonuje wbudowana metoda `println()`. W tym przypadku wyświetla ona łańcuch znaków, który został jej przekazany. Zobaczysz później, że metody tej można używać również do wyświetlania innych typów informacji. Omawiany wiersz rozpoczyna się od `System.out`. Na tym etapie musisz zadowolić się wyjaśnieniem, że `System` jest predefiniowaną klasą umożliwiającą dostęp do systemu, a `out` jest strumieniem wyjściowym połączonym z konsolą. W ten sposób `System.out` jest obiektem hermetyzującym wyjście konsoli. Fakt, że Java używa obiektu do hermetyzacji wyjścia konsoli, jest kolejnym dowodem jej obiektowej natury.

Jak pewnie się domyśliłeś, wyjście konsoli (i wejście) nie są zbyt często używane przez profesjonalne aplikacje Javy. Większość nowoczesnych programów używa graficznego interfejsu użytkownika i dlatego wejście-wyjście konsoli jest wykorzystywane głównie przez proste programy narzędziowe, programy demonstracyjne oraz kod działający na serwerach. W późniejszych rozdziałach poznasz inne sposoby generowania informacji na wyjściu programów Java, ale na razie będziemy nadal korzystać z metod wejścia-wyjścia konsoli.

Zwróć uwagę, że wiersz wywołujący metodę `println()` kończy się średnikiem. Wiele instrukcji w Javie należy kończyć w taki sposób. Jak się przekonasz, średnik jest bardzo ważnym elementem składni języka Java.

Pierwszy z klamrowych nawiasów zamykających `}` kończy definicję metody `main()`, a drugi definicję klasy `Example`.

Na koniec jeszcze ważna uwaga: Java rozróżnia małe i wielkie litery. Jeśli o tym zapomnisz, wpadniesz w tarapaty. Na przykład: jeśli przez przypadek napiszesz `Main` zamiast `main` lub `PrintLn` zamiast `println`, to nasz przykładowy program przestanie być poprawny. Co więcej, chociaż kompilator Javy *skompiluje* klasy niezawierające metody `main()`, to maszyna wirtualna nie będzie mogła uruchomić takiego programu.

Jeśli zatem błędnie wpiszesz nazwę `main`, to program skompiluje się bez błędu. Jednak interpreter Javy zgłosi błąd, gdyż nie znajdzie metody `main()`.

Obsługa błędów składni

Jeśli jeszcze tego nie zrobiłeś, najwyższa pora wprowadzić tekst programu, skompilować go i uruchomić. Jak pewnie wiele razy już tego doświadczyłeś, podczas wprowadzania tekstu programu łatwo popełnić błąd. Na szczęście podczas kompilacji zostaniesz w takim przypadku poinformowany o **błędzie składni**. Kompilator Java próbuje za wszelką cenę doszukać się sensu we wprowadzonym przez Ciebie tekście i dlatego komunikat o błędzie nie zawsze odzwierciedla prawdziwe źródło problemu. Na przykład w naszym programie przypadkowe pominięcie klamrowego nawiasu otwierającego metodę `main()` spowoduje wyświetlenie przez kompilator następującej informacji o dwóch błędach:

```
Example.java:8: ';' expected
  public static void main(String[] args)
                                        ^
Example.java:11: class, interface, or enum expected
}
^
```

Jak łatwo zauważysz, pierwszy komunikat nie jest poprawny, ponieważ w kodzie nie brakuje średnika, lecz nawiasu klamrowego.

Z tego przykładu wynika wniosek, że gdy program zawiera błąd składni, nie należy do końca ufać komunikatom kompilatora, ponieważ mogą wprowadzać w błąd. Należy zastanowić się nad inną możliwą przyczyną błędu, zwykle analizując w tym celu kilka wierszy kodu poprzedzających komunikat o błędzie. Często bowiem kompilator sygnalizuje błąd dopiero kilka wierszy po jego wystąpieniu.

Drugi prosty program

Nie ma chyba bardziej podstawowej instrukcji języka programowania niż przypisanie wartości do zmiennej. **Zmienna** stanowi reprezentację lokalizacji w pamięci, w której można umieścić pewną wartość. Wartość zmiennej może być zmieniana w trakcie działania programu. Program przedstawiony na listingu 1.2 tworzy dwie zmienne o nazwach `myVar1` i `myVar2`.

Listing 1.2. *Example2.java*

```
/*
    Demonstruje użycie zmiennych
    Nazwij plik Example2.java
*/
class Example2 {
  public static void main(String[] args) {
    int myVar1; // Deklaracja zmiennej   <-- Deklaracja zmiennej
    int myVar2; // Deklaracja kolejnej zmiennej

    myVar1 = 1024; // Przypisanie 1024 do myVar1   <-- Przypisanie wartości do zmiennej

    System.out.println("myVar1 zawiera " + myVar1);

    myVar2 = myVar1 / 2;

    System.out.print("myVar2 zawiera myVar1 / 2: ");
    System.out.println(myVar2);
  }
}
```

Po jego uruchomieniu pojawią się poniższe napisy:

```
myVar1 zawiera 1024
myVar2 zawiera myVar1 / 2: 512
```

W programie tym pojawia się kilka nowych elementów. Na przykład instrukcja

```
int myVar1; // Deklaracja zmiennej
```

zawiera deklarację zmiennej `myVar1` typu całkowitego. W języku Java musisz zadeklarować każdą zmienną, zanim zaczniesz jej używać. Co więcej, musisz również określić typ wartości, które może ona przechowywać, czyli **typ** zmiennej. W tym przypadku zmienna `myVar1` może przechowywać wartości całkowite. W Javie zmienną taką deklarujemy, poprzedzając ją słowem kluczowym `int`. W ten sposób powyższy wiersz zawiera deklarację zmiennej `myVar1` typu `int`.

Kolejny wiersz deklaruje drugą zmienną o nazwie `myVar2`:

```
int myVar2; // Deklaracja kolejnej zmiennej
```

Zwróć uwagę, że deklaracja ta ma taką samą postać jak w przypadku poprzedniej zmiennej, różni się jedynie nazwą zmiennej.

W ogólnym przypadku deklaracja zmiennej ma postać:

```
typ nazwa-zmiennej;
```

W tym przypadku *typ* oznacza typ deklarowanej zmiennej, natomiast *nazwa-zmiennej* to jej nazwa. Oprócz typu `int` w Javie dostępnych jest wiele innych typów.

Kolejny wiersz programu przypisuje zmiennej `myVar1` wartość `1024`:

```
myVar1 = 1024; // Przypisanie 1024 do myVar1
```

W języku Java operator przypisania zapisujemy za pomocą pojedynczego znaku równości. Powoduje on skopiowanie wartości znajdującej się po prawej stronie operatora do zmiennej po jego lewej stronie.

Zadaniem następnego wiersza jest wyświetlenie wartości zmiennej `myVar1` poprzedzonej łańcuchem `myVar1 zawiera`:

```
System.out.println("myVar1 zawiera " + myVar1);
```

Znak + oznacza, że wartość zmiennej `myVar1` należy wyświetlić po poprzedzającym ją łańcuchu znaków. Za pomocą operatora + możesz łączyć dowolnie wiele elementów wyświetlanych za pomocą jednej instrukcji `println()`.

Kolejna instrukcja przypisuje zmiennej `myVar2` wartość zmiennej `myVar1` podzieloną przez 2:

```
myVar2 = myVar1 / 2;
```

Ta instrukcja dzieli wartość zmiennej `myVar1` przez 2, a wynik umieszcza w zmiennej `myVar2`. W ten sposób po jej wykonaniu zmienna `myVar2` będzie zawierać wartość `512`. Wartość zmiennej `myVar1` nie ulegnie zmianie. Podobnie jak w wielu innych językach programowania, tak i w Javie dostępnych jest szereg operatorów arytmetycznych, w tym operator dodawania +, odejmowania –, mnożenia * i dzielenia /.

Kolejne dwa wiersze programu wyglądają następująco:

```
System.out.print("myVar2 zawiera myVar1 / 2: ");
System.out.println(myVar2);
```

Pojawiają się w nich dwa nowe elementy. W pierwszym do wyświetlenia łańcucha `myVar2 zawiera myVar1 / 2:` została użyta wbudowana metoda `print()`. Jej użycie spowoduje, że po wyświetleniu łańcucha znaków *nie* zostanie umieszczony znak nowego wiersza. W efekcie kolejna instrukcja wyjścia będzie kontynuować wyświetlanie w tym samym wierszu. Metoda `print()` działa tak samo jak metoda `println()`, z tą różnicą, że nie umieszcza znaku nowego wiersza na końcu wyświetlanych informacji. Drugą nowością jest użycie samej zmiennej `myVar2` jako parametru wywołania metody `println()`. Zarówno `print()`, jak i `println()` mogą być używane do wyświetlania wartości dowolnego typu wbudowanego w Javie.

Zanim przejdziemy do innych zagadnień, najpierw jeszcze jedna uwaga dotycząca deklarowania zmiennych: jedna instrukcja deklaracji może zawierać deklarację dwu lub więcej zmiennych. Wystarczy w tym celu rozdzielić ich nazwy przecinkiem. Na przykład zmienne `myVar1` i `myVar2` mógłbyś zadeklarować w poniższy sposób:

```
int myVar1, myVar2; // Obie zadeklarowane w jednej instrukcji
```

Inne typy danych

W poprzednim programie użyłeś już zmiennych typu `int`. Jednak zmienne tego typu mogą przechowywać jedynie wartości całkowite. Nie możesz zatem umieszczać w nich wartości ułamkowych. Na przykład zmiennej typu `int` możesz przypisać wartość `18`, ale `18.3` już nie. Na szczęście `int` jest tylko jednym z typów dostępnych w Javie. Wartości ułamkowe możesz reprezentować za pomocą dwóch typów zmiennoprzecinkowych: `float` i `double`, które umożliwiają reprezentację wartości odpowiednio o pojedynczej i podwójnej precyzji. Z tych dwóch typów częściej używany jest `double`.

Zmienną typu `double` zadeklarujesz za pomocą poniższej instrukcji:

```
double x;
```

W tym przypadku `x` jest nazwą zmiennej typu `double`. Ponieważ `x` jest typu zmiennoprzecinkowego, może przechowywać wartości takie jak `122.23`, `0.034` czy `–19.0`.

Aby lepiej zrozumieć różnicę pomiędzy typami `int` i `double`, uruchom poniższy program przedstawiony na listingu 1.3.

Listing 1.3. *Example3.java*

```
/*
   Program ilustruje różnice
   pomiędzy int i double
*/
class Example3 {
  public static void main(String[] args) {
    int v;   // Deklaracja zmiennej typu int
    double x; // Deklaracja zmiennej typu double

    v = 10; // Przypisuje zmiennej var wartość 10

    x = 10.0; // Przypisuje zmiennej x wartość 10.0

    System.out.println("Początkowa wartość zmiennej v: " + v);
    System.out.println("Początkowa wartość zmiennej x: " + x);
    System.out.println(); // Wyświetla pusty wiersz

    // Dzieli wartość obu zmiennych przez 4
    v = v / 4;
    x = x / 4;

    System.out.println("v po dzieleniu: " + v);
    System.out.println("x po dzieleniu: " + x);
  }
}
```

Wyświetla pusty wiersz

Ekspert odpowiada

Pytanie: Dlaczego Java ma różne typy danych dla liczb całkowitych i zmiennoprzecinkowych? To znaczy dlaczego wszystkie wartości liczbowe nie są tego samego typu?

Odpowiedź: Java udostępnia różne typy danych, dzięki czemu można pisać wydajne programy. Na przykład arytmetyka liczb całkowitych jest szybsza niż obliczenia zmiennoprzecinkowe. Tak więc jeśli nie potrzebujesz wartości ułamkowych, nie musisz ponosić kosztów związanych ze stosowaniem typów `float` lub `double`. Poza tym ilość pamięci wymagana dla jednego typu danych może być mniejsza niż dla innego. Dzięki udostępnieniu różnych typów Java umożliwia jak najlepsze wykorzystanie zasobów systemowych. Wreszcie niektóre algorytmy wymagają (lub przynajmniej używają) określonego typu danych. Ogólnie rzecz biorąc, Java dostarcza wiele wbudowanych typów, aby zapewnić maksymalną elastyczność.

Jego wykonanie spowoduje wyświetlenie następujących informacji:

```
Początkowa wartość zmiennej v: 10
Początkowa wartość zmiennej x: 10.0

v po dzieleniu: 2  <-- Część dziesiętna utracona
x po dzieleniu: 2.5  <-- Część dziesiętna zachowana
```

Jak łatwo zauważysz, w przypadku dzielenia wartości zmiennej przez 4 mamy do czynienia z dzieleniem wartości całkowitej i jego wynikiem jest 2, gdyż ułamkowa reszta z dzielenia zostaje utracona. Zostaje ona zachowana w przypadku dzielenia zmiennej x, która jest typu zmiennoprzecinkowego — `double`.

W programie tym jest jeszcze jedna nowość. Aby wyświetlić pusty wiersz, wywołujemy metodę `println()` bez argumentów.

Przykład 1.1. Zamiana galonów na litry

GalToLit.java

Chociaż poprzednie przykłady umożliwiły Ci poznanie kilku istotnych właściwości języka Java, nie wykonywały żadnych użytecznych działań. Mimo że na tym etapie nie wiesz jeszcze zbyt wiele o Javie, to możesz już wykorzystać nabyte informacje do stworzenia przydatnego programu. W tym przykładzie stworzymy program, którego zadaniem będzie zamiana galonów na litry. W tym celu zadeklarujemy dwie zmienne typu `double`. Jedna przechowywać będzie liczbę galonów, a druga liczbę litrów uzyskaną wskutek zamiany. Jeden galon odpowiada 3,7854 litra. Zatem aby zamienić galony na litry, wystarczy pomnożyć ich liczbę przez 3,7854. Na koniec program wyświetli liczbę galonów i odpowiadającą im liczbę litrów.

1. Utwórz nowy plik o nazwie *GalToLit.java*.
2. Wprowadź w tym pliku tekst programu przedstawiony na listingu 1.4.

Listing 1.4. *GalToLit.java*

```
/*
    Wypróbuj to 1.1
    Program zamieniający galony na litry

    Nazwij plik GalToLit.java
*/
class GalToLit {
  public static void main(String[] args) {
    double gallons; // Przechowuje liczbę galonów
    double liters;  // Przechowuje liczbę litrów (po zamianie)

    gallons = 10; // 10 galonów

    liters = gallons * 3.7854; // Zamienia na litry

    System.out.println(gallons + " galonów odpowiada " + liters + " litrom.");
  }
}
```

3. Skompiluj program w poniższy sposób.

   ```
   javac GalToLit.java
   ```

4. Uruchom program jak poniżej.

   ```
   java -Dfile.encoding=CP852 GalToLit
   ```

   Program wyświetli napis:

   ```
   10.0 galonów odpowiada 37.854 litrom.
   ```

5. W obecnej postaci program wyświetla liczbę litrów odpowiadającą 10 galonom. Zmieniając wartość zmiennej `gallons`, możesz uzyskać informację o liczbie litrów odpowiadającej innej liczbie galonów.

Dwie instrukcje sterujące

Instrukcje umieszczone wewnątrz metody wykonywane są jedna po drugiej. Możesz to zmienić, używając różnych instrukcji sterujących dostępnych w Javie. Chociaż zostaną one przedstawione szczegółowo później, dwie z nich omówię teraz krótko, abyś mógł używać ich w kolejnych przykładach.

Instrukcja if

Za pomocą instrukcji warunkowej `if` możesz dokonywać wyboru wykonywanych fragmentów programu. Instrukcja ta działa podobnie jak instrukcje IF w innych językach programowania. Określa tok realizacji programu zależnie od tego, czy podany warunek zostanie spełniony czy nie. W najprostszej postaci wygląda ona następująco:

```
if (warunek) instrukcja;
```

Tutaj *warunek* reprezentuje wyrażenie logiczne. (Wyrażenie logiczne to takie, którego wynikiem jest prawda lub fałsz). Jeśli jest prawdziwe, zostanie wykonana *instrukcja*. W przeciwnym razie zostanie ona pominięta. Oto przykład:

```
if (10 < 11) System.out.println("10 jest mniejsze od 11 ");
```

W tym przykładzie rzeczywiście 10 jest mniejsze od 11, zatem wyrażenie warunkowe jest prawdziwe i instrukcja zostaje wykonana. Inaczej niż w kolejnym przykładzie:

```
if (10 < 9) System.out.println("ten tekst nie zostanie wyświetlony ");
```

Ponieważ 10 nie jest mniejsze od 9, metoda `println()` nie zostanie wywołana.

Java definiuje pełny zestaw operatorów relacyjnych, których możesz używać w wyrażeniach warunkowych. Przedstawiłem go w tabeli 1.2.

Tabela 1.2. *Operatory relacyjne w języku Java*

Operator	Znaczenie
<	Mniejsze od
<=	Mniejsze od lub równe
>	Większe od
>=	Większe od lub równe
==	Równe
!=	Różne od

Zwróć uwagę, że sprawdzenie równości wymaga użycia dwóch znaków równości.

Program ilustrujący zastosowanie instrukcji `if` przedstawiłem na listingu 1.5.

Listing 1.5. *IfDemo.java*

```
/*
  Demonstruje instrukcję warunkową if

  Nazwij plik IfDemo.java
*/

class IfDemo {
  public static void main(String[] args) {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) System.out.println("a jest mniejsze od b");
```

```
    // Poniższa instrukcja nie zostanie wykonana
    if(a == b) System.out.println("To wywołanie nie zostanie wykonane");

    System.out.println();

    c = a - b; // c zawiera -1

    System.out.println("c zawiera -1");
    if(c >= 0) System.out.println("c nie jest ujemne");
    if(c < 0) System.out.println("c jest ujemne");

    System.out.println();

    c = b - a; // c zawiera teraz 1

    System.out.println("c zawiera 1");
    if(c >= 0) System.out.println("c jest nieujemne");
    if(c < 0) System.out.println("c jest ujemne");

  }
}
```

Efekt działania programu jest następujący:

```
a jest mniejsze od b

c zawiera –1
c jest ujemne

c zawiera 1
c jest nieujemne
```

Zwróć uwagę na poniższy wiersz programu.

```
int a, b, c;
```

Zawiera on deklarację trzech zmiennych — `a`, `b`, `c` — w postaci listy nazw zmiennych rozdzielonych przecinkami. Jak już wcześniej wspomniałem, jeśli deklarujesz dwie lub więcej zmiennych tego samego typu, możesz to uczynić za pomocą pojedynczej instrukcji. Wystarczy rozdzielić nazwy zmiennych przecinkami.

Pętla for

Tę samą sekwencję kodu możesz wykonać wielokrotnie, tworząc tak zwaną **pętlę**. Pętle są używane zawsze, gdy jakieś zadanie trzeba wykonać wiele razy — zastosowanie ich jest znacznie łatwiejsze niż próbowanie wielokrotnego wstawiania do kodu tej samej instrukcji lub bloku instrukcji. Java udostępnia rozbudowany asortyment instrukcji pętli. Na razie przyjrzymy się jedynie instrukcji `for`. Najprostsza postać pętli `for` wygląda następująco:

```
for(inicjalizacja; warunek; iteracja) instrukcja;
```

Najczęściej *`inicjalizacja`* sprowadza się do nadania wartości początkowej zmiennej sterującej pętli. *`Warunek`* jest wyrażeniem logicznym sprawdzającym wartość zmiennej sterującej. Jeśli wyrażenie jest spełnione (prawdziwe), zostaje wykonana *`instrukcja`*, a pętla `for` kontynuuje iterację. W przeciwnym razie pętla kończy swoje działanie. Wyrażenie *`iteracja`* określa sposób zmiany wartości zmiennej sterującej podczas kolejnych wykonań pętli. Na listingu 1.6 przedstawiłem krótki program ilustrujący działanie pętli `for`.

Listing 1.6. *ForDemo.java*

```
/*
    Demonstruje pętlę for

    Nazwij plik ForDemo.java
*/
```

```
class ForDemo {
  public static void main(String[] args) {
    int count;

    for(count = 0; count < 5; count = count + 1) ←——— Ta pętla wykonuje 5 iteracji
      System.out.println("Wartość zmiennej sterującej count: " + count);

    System.out.println("Pętla wykonana!");
  }
}
```

Efekt jego działania będzie wyglądać jak poniżej:

```
Wartość zmiennej sterującej count: 0
Wartość zmiennej sterującej count: 1
Wartość zmiennej sterującej count: 2
Wartość zmiennej sterującej count: 3
Wartość zmiennej sterującej count: 4
Pętla wykonana!
```

W tym przykładzie `count` jest zmienną sterującą pętli. Podczas inicjalizacji pętli zmienna `count` otrzymuje wartość zero. Przed każdym wykonaniem pętli (również pierwszym) sprawdzany jest warunek `count < 5`. Jeśli jest on prawdziwy, wywołana zostaje metoda `println()`, a następnie wykonana zostaje iteracja pętli, która w powyższym przykładzie polega na powiększeniu wartości zmiennej `count` o 1. Proces ten jest kontynuowany do momentu, w którym wyrażenie warunkowe przestaje być prawdziwe. W profesjonalnych programach rzadko spotkasz sposób zapisu iteracyjnej części pętli, który zastosowaliśmy w naszym programie:

```
count = count + 1;
```

Java udostępnia specjalny operator inkrementacji, który pozwala efektywniej wykonać tę samą operację. Operator ten zapisujemy jako ++ (czyli za pomocą dwóch sąsiadujących znaków plus). Operator inkrementacji zwiększa wartość zmiennej o 1. Za jego pomocą możemy zapisać część iteracyjną naszej pętli jako

```
count++;
```

Ostatecznie więc nasza pętla przyjmie następującą postać:

```
for(count = 0; count < 5; count++)
```

Warto, abyś ją wypróbował. Jak zauważysz, sposób działania pętli nie zmieni się.

Java udostępnia również operator dekrementacji, zapisywany jako `--`. Zmniejsza on wartość zmiennej o jeden.

Bloki kodu

Kolejnym kluczowym elementem programów w Javie jest **blok kodu**. Przez blok kodu rozumiemy grupę dwóch lub więcej instrukcji. Tworząc blok kodu, umieszczamy grupę instrukcji wewnątrz nawiasów klamrowych. Blok kodu jest jednostką logiczną, którą możemy umieścić zamiast pojedynczej instrukcji. Na przykład blok taki może być wykonywany przez instrukcję warunkową `if` lub w pętli `for`. Przyjrzyj się poniższej instrukcji warunkowej `if`:

```
if (w < h) { ←——— Początek bloku
  v = w * h;
  w = 0;
} ←——— Koniec bloku
```

W tym przykładzie, jeśli `w` jest mniejsze od `h`, wykonane zostaną obie instrukcje umieszczone w bloku. W ten sposób obie instrukcje wewnątrz bloku stanowią logiczną całość i jedna nie może zostać wykonana bez drugiej. Jeśli potrzebujesz powiązać ze sobą dwie lub więcej instrukcji, to powinieneś umieścić je właśnie w jednym bloku. Zastosowanie bloków kodu pozwala zwiększyć przejrzystość programów i efektywność ich działania.

Na listingu 1.7 przedstawiam program, który używa bloku kodu, aby zapobiec dzieleniu przez zero.

Listing 1.7. *BlockDemo.java*

```
/*
   Demonstruje blok kodu

   Nazwij plik BlockDemo.java
*/
class BlockDemo {
  public static void main(String[] args) {
    double i, j, d;

    i = 5;
    j = 10;

    // Instrukcja warunkowa decyduje o wykonaniu bloku kodu
    if(i != 0) {
      System.out.println("i nie jest równe zero");
      d = j / i;
      System.out.print("j / i równa się " + d);
    }
  }
}
```

Instrukcja if decyduje o wykonaniu całego bloku

Na skutek jego wykonania wyświetlone zostaną poniższe informacje:

```
i nie jest równe zero
j / i równa się 2.0
```

W tym przykładzie instrukcja warunkowa `if` decyduje o wykonaniu całego bloku kodu, a nie tylko pojedynczej instrukcji. Jeśli wyrażenie warunkowe instrukcji jest prawdziwe (a tak jest w tym przypadku), wykonane zostaną wszystkie trzy instrukcje umieszczone wewnątrz bloku. Spróbuj zmienić wartość początkową zmiennej `i` na zero i obserwuj wynik działania programu. Zobaczysz, że pominięty zostanie cały blok kodu.

Ekspert odpowiada

Pytanie: Czy stosowanie bloków kodu powoduje pogorszenie efektywności działania programów? Innymi słowy, czy znaki { i } oznaczają konieczność wykonania dodatkowego kodu?

Odpowiedź: Nie. Ze stosowaniem bloków kodu nie jest związany żaden narzut. Wręcz przeciwnie, ponieważ pozwalają one uprościć kod wielu algorytmów, to ich zastosowanie powoduje wzrost efektywności działania programów. Znaki { i } istnieją jedynie w kodzie źródłowym i ich obecność nie powoduje wykonania żadnego dodatkowego kodu przez Javę.

W dalszej części książki przekonasz się, że bloki kodu posiadają dodatkowe właściwości i zastosowania. Jednak głównym powodem ich istnienia jest tworzenie logicznych jednostek instrukcji w kodzie programów.

Średnik i pozycja kodu w wierszu

W języku Java średnik jest **separatorem**. Jest on często stosowany do kończenia instrukcji. Można by rzec, że średnik oznacza zakończenie pewnej logicznej całości.

Jak się przed chwilą dowiedziałeś, blok kodu stanowi zbiór powiązanych logicznie instrukcji ujętych w znaki nawiasów klamrowych. Dlatego też blok kodu *nie* kończy się średnikiem. Koniec bloku kodu jest oznaczony zamykającym nawiasem klamrowym.

Java nie rozpoznaje znaku końca wiersza jako końca instrukcji. Z tego powodu nie ma znaczenia, w jaki sposób podzielisz kod pomiędzy kolejne wiersze.

Na przykład zapis:

```
x = y;
y = y + 1;
System.out.println(x + " " + y);
```

nie różni się z punktu widzenia Javy niczym od poniższego:

```
x = y; y = y + 1; System.out.println(x + " " + y);
```

Co więcej, poszczególne elementy instrukcji możesz również umieścić w osobnych wierszach. Na przykład poniższy sposób zapisu jest akceptowany w Javie:

```
System.out.println("Bardzo długi napis" +
                    x + y + z +
                   "dalszy ciąg napisu");
```

Podział długich wierszy w powyższy sposób poprawia czytelność programu i zapobiega konieczności „zawijania" zbyt długich wierszy przez edytor tekstu.

Wcięcia

Analizując kod poprzednich przykładów z pewnością zauważyłeś, że niektóre instrukcje zostały poprzedzone wcięciami w tekście. Jak przed chwilą wyjaśniłem, Java pozostawia programistom dużą swobodę odnośnie do sposobu umieszczania instrukcji w tekście programów. Jednak lata praktyki wielu programistów doprowadziły do wypracowania powszechnie akceptowanego stylu wcinania kodu, który poprawia jego czytelność. Styl ten będę konsekwentnie stosować w tej książce i polecam to również Tobie. Zasadą tego stylu jest stosowanie kolejnego poziomu wcięcia po każdym nawiasie otwierającym oraz powrót poziom wyżej po każdym nawiasie zamykającym. Niektóre instrukcje wymagają stosowania dodatkowych wcięć, co omówię później.

Przykład 1.2. Ulepszony konwerter galonów na litry

GalToLitTable.java

Użyjesz teraz pętli `for`, instrukcji warunkowej `if` i bloku kodu, aby ulepszyć konwerter galonów na litry. Nowa wersja będzie wyświetlać kompletną tabelę konwersji z zakresu od 1 do 100 galonów. Po każdych 10 galonach wyświetlony zostanie pusty wiersz odstępu. Zwróć szczególną uwagę na zastosowanie zmiennej `counter`, która będzie w tym celu zliczać liczbę wyświetlonych wierszy.

1. Utwórz nowy plik o nazwie *GalToLitTable.java*.
2. Umieść w nim poniższy kod programu przedstawiony na listingu 1.8.

Listing 1.8. *GalToLitTable.java*

```
/*
    Wypróbuj to 1.2

    Program wyświetla tabelę zamiany
    galonów na litry

    Nazwij plik GalToLitTable.java
*/
class GalToLitTable {
  public static void main(String[] args) {
    double gallons, liters;
    int counter;

    counter = 0; <-------------------------- Licznik wierszy otrzymuje początkową wartość zero
    for(gallons = 1; gallons <= 100; gallons++) {
      liters = gallons * 3.7854; // Zamiana na litry
      System.out.println(gallons + " galonów to " +
```

```
                        liters + " litrów.");

      counter++;  <-- Inkrementacja licznika z każdą iteracją pętli
      // W co dziesiątym wierszu wyświetlamy pusty wiersz (odstęp)
      if(counter == 10) {  <-- Jeśli licznik wierszy jest równy 10, wyświetla pusty wiersz
        System.out.println();
        counter = 0; // Zeruje licznik wierszy
      }
    }
  }
}
```

3. Skompiluj program za pomocą następującego wywołania:

```
javac GalToLitTable.java
```

4. Uruchom go za pomocą poniższej komendy:

```
java GalToLitTable
```

Oto fragment tabeli wyników tworzonej przez program[1]:

```
1.0 galonów to 3.7854 litrów.
2.0 galonów to 7.5708 litrów.
3.0 galonów to 11.356200000000001 litrów.
4.0 galonów to 15.1416 litrów.
5.0 galonów to 18.927 litrów.
6.0 galonów to 22.712400000000002 litrów.
7.0 galonów to 26.4978 litrów.
8.0 galonów to 30.2832 litrów.
9.0 galonów to 34.0686 litrów.
10.0 galonów to 37.854 litrów.

11.0 galonów to 41.6394 litrów.
12.0 galonów to 45.424800000000005 litrów.
13.0 galonów to 49.2102 litrów.
14.0 galonów to 52.9956 litrów.
15.0 galonów to 56.781 litrów.
16.0 galonów to 60.5664 litrów.
17.0 galonów to 64.3518 litrów.
18.0 galonów to 68.1372 litrów.
19.0 galonów to 71.9226 litrów.
20.0 galonów to 75.708 litrów.

21.0 galonów to 79.49340000000001 litrów.
22.0 galonów to 83.2788 litrów.
23.0 galonów to 87.0642 litrów.
24.0 galonów to 90.84960000000001 litrów.
25.0 galonów to 94.635 litrów.
26.0 galonów to 98.4204 litrów.
27.0 galonów to 102.2058 litrów.
28.0 galonów to 105.9912 litrów.
29.0 galonów to 109.7766 litrów.
30.0 galonów to 113.562 litrów.
```

Słowa kluczowe języka Java

Obecnie zdefiniowanych jest sześćdziesiąt jeden słów kluczowych języka Java (patrz tabela 1.3). Słowa te w połączeniu ze składnią operatorów i separatorów tworzą definicję języka Java. Ogólnie

[1] Prawdopodobnie spodziewałeś się, że 3 galony to 11,3562 litra. Jednak program wyświetlił wartość 11.356200000000001. Musisz przyzwyczaić się, że ze względu na binarną reprezentację liczb zmiennoprzecinkowych wyniki nawet najprostszych działań (a nawet wartości stałych zmiennoprzecinkowych) mogą różnić się nieco od spodziewanych dziesiętnych wartości zmiennoprzecinkowych — *przyp. tłum.*

rzecz biorąc, słów tych nie wolno stosować jako nazw zmiennych, klas czy metod. Jednak 16 spośród nich to tak zwane kontekstowe słowa kluczowe, co oznacza, że są one traktowane jako słowa kluczowe tylko wtedy, gdy są używane z możliwością języka, do której się odnoszą. Obsługują one możliwości dodane do Javy w ciągu ostatnich kilku lat. Dziesięć spośród nich odnosi się do modułów — są to słowa kluczowe `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` i `with`. Rekordy są deklarowane przy użyciu słowa kluczowego `record`. Zapieczętowane klasy i interfejsy używają słów kluczowych `sealed`, `non-sealed` i `permits`, `yield` jest używane przez rozszerzoną instrukcję `switch`, a `var` obsługuje wnioskowanie typów zmiennych lokalnych. Ponieważ są one zależne od kontekstu, ich dodanie nie miało wpływu na już istniejący kod. Co więcej, zaczynając od JDK 9, sam znak podkreślenia jest uznawany za słowo kluczowe, aby nie można było używać go jako nazwy czegoś w kodzie programu. Począwszy od JDK 17, `strictfp` nie ma już żadnego efektu i jest niepotrzebne. Jest to jednak nadal słowo kluczowe Javy.

Tabela 1.3. *Słowa kluczowe języka Java*

`abstract`	`assert`	`boolean`	`break`	`byte`	`case`
`catch`	`char`	`class`	`const`	`continue`	`default`
`do`	`double`	`else`	`enum`	`exports`	`extends`
`final`	`finally`	`float`	`for`	`goto`	`if`
`implements`	`import`	`instanceof`	`int`	`interface`	`long`
`module`	`native`	`new`	`non-sealed`	`open`	`opens`
`package`	`permits`	`private`	`protected`	`provides`	`public`
`record`	`requires`	`return`	`sealed`	`short`	`static`
`strictfp`	`super`	`switch`	`synchronized`	`this`	`throw`
`throws`	`to`	`transient`	`transitive`	`try`	`uses`
`var`	`void`	`volatile`	`while`	`with`	`_`

Słowa kluczowe `const` i `goto` zostały zarezerwowane, ale nie są używane. Początkowo dla języka Java zarezerwowano również kilka innych słów kluczowych, ale obecna definicja języka zawiera jedynie słowa kluczowe przedstawione w tabeli.

Oprócz tych słów kluczowych Java rezerwuje również trzy inne, stosowane w niej od samego początku, a są to `true`, `false` i `null`. Reprezentują one wartości zdefiniowane w języku. Również tych słów nie wolno używać jako nazw zmiennych, klas itd.

Identyfikatory

W języku Java przez identyfikator rozumiemy nazwę nadaną metodzie, zmiennej lub każdemu innemu elementowi zdefiniowanemu przez użytkownika. Nazwy zmiennych mogą rozpoczynać się dowolną literą alfabetu, znakiem podkreślenia lub znakiem dolara. Po tych znakach może pojawić się kolejna litera, cyfra, znak dolara lub podkreślenia. (Znak dolara, `$`, nie jest przeznaczony od ogólnego użytku). Ten ostatni może być stosowany dla poprawienia czytelności nazwy, na przykład `licznik_wierszy`. Java rozróżnia małe i wielkie litery. Zatem w Javie `mavar` i `MyVar` to dwa różne identyfikatory zmiennych. Oto kilka przykładów poprawnych identyfikatorów:

`Test`	`x`	`y2`	`MaxLoad`
`up`	`_top`	`my_var`	`sample23`

Pamiętaj, że identyfikator nie może rozpoczynać się cyfrą. Na przykład identyfikator `12x` nie jest poprawny.

Nie wolno również używać słów kluczowych języka Java jako identyfikatorów. Zabronione jest też wykorzystywanie w tym celu nazw metod standardowych, takich jak `println`. Niezależnie od tych zakazów dobra praktyka programistyczna nakazuje stosowanie identyfikatorów odzwierciedlających znaczenie lub zastosowanie elementów, których są nazwami.

Biblioteki klas

Przykładowe programy przedstawione w tym rozdziale wykorzystywały dwie wbudowane metody języka Java: `println()` i `print()`. Metody te są używane za pośrednictwem składowej `System.out`. Klasa `System` to predefiniowana klasa, która jest automatycznie dołączana do Twoich programów. Środowisko Java bazuje na kilku bibliotekach klas, z których każda zawiera wiele metod służących na przykład do obsługi wejścia-wyjścia, przetwarzania łańcuchów znakowych, obsługi sieci czy tworzenia graficznego interfejsu użytkownika. Standardowe klasy języka Java umożliwiają również tworzenie okienkowego interfejsu programów. Java jako całość jest zatem połączeniem języka programowania i klas standardowych. Jeśli masz zamiar zostać profesjonalnym programistą Javy, z czasem będziesz musiał nauczyć się korzystać ze standardowych klas tego języka. W tej książce pojawią się różne elementy standardowych bibliotek klas i metod, ale pełne ich poznanie będzie wymagać od Ciebie samodzielnego studiowania tego tematu.

Test sprawdzający

1. Czym jest kod bajtowy i dlaczego ma on takie znaczenie dla zastosowań Javy w Internecie?
2. Wymień trzy podstawowe zasady programowania obiektowego.
3. Od czego zaczyna się działanie programu w języku Java?
4. Czym jest zmienna?
5. Które z poniższych nazw zmiennych nie są poprawne?
   - **A.** `count`
   - **B.** `$count`
   - **C.** `count27`
   - **D.** `67count`
6. Jak tworzymy w tekście programu komentarz jednowierszowy, a jak wielowierszowy?
7. Przedstaw ogólną postać instrukcji `if`. Przedstaw ogólną postać pętli `for`.
8. W jaki sposób tworzymy blok kodu?
9. Grawitacja Księżyca stanowi około 17% ziemskiej. Napisz program, który obliczy Twój ciężar na Księżycu.
10. Zaadaptuj program przedstawiony na listingu 1.8, aby wyświetlał tabelę konwersji cali na metry. Wyświetl ją dla 12 stóp, cal po calu. Umieść pusty wiersz co 12 cali. (Jeden metr równa się w przybliżeniu 39,37 cala).
11. Jeśli pomylisz się, wprowadzając kod źródłowy programu, jaki błąd pojawi się podczas kompilacji?
12. Czy pozycja instrukcji w wierszu ma znaczenie?

Java™: twórz, kompiluj i uruchamiaj najlepszy kod!

Java cały czas dynamicznie się rozwija, a każda wersja przynosi nowe rozwiązania. Ten wszechstronny język pozwala na tworzenie stron WWW, aplikacji internetowych, aplikacji mobilnych i wysoko specjalistycznego oprogramowania dla przeróżnych urządzeń technicznych. Jest to możliwe między innymi dzięki licznym bibliotekom i narzędziom wspierającym proces kodowania. Mimo upływu lat Java wciąż pozostaje żywym i nowoczesnym językiem wybieranym przez profesjonalistów.

To dziewiąte wydanie znakomitego podręcznika programowania dla początkujących, starannie zaktualizowane i uzupełnione o informacje dotyczące Java Platform, Standard Edition 17. Książka rozpoczyna się od solidnej dawki wiedzy o kompilacji i uruchamianiu programu w Javie, słowach kluczowych i istotnych konstrukcjach w tym języku. Krok po kroku przedstawia kolejne, coraz bardziej zaawansowane zagadnienia dotyczące obiektów, dziedziczenia czy wyjątków, a także współbieżności, typów sparametryzowanych, wyrażeń lambda i modułów. Poszczególne partie materiału są bogato ilustrowane przykładami kodu z komentarzami, a ponadto uzupełnione praktycznymi ćwiczeniami, testami sprawdzającymi, wskazówkami i dodatkowymi informacjami. Przejrzysty układ podręcznika i jasny, zrozumiały język zdecydowanie ułatwiają naukę.

W książce znajdziesz:

- treści dobrane pod kątem efektywnej i łatwej nauki
- eksperckie wskazówki i podpowiedzi
- praktyczne przykłady zastosowania nabytych umiejętności
- utrwalające wiedzę testy sprawdzające
- przykładowe kody uzupełnione zrozumiałymi komentarzami

Herbert Schildt jest autorem popularnych książek i niekwestionowanym autorytetem w zakresie programowania w Javie. Napisał wiele cenionych podręczników programowania w Javie, C, C++ i C#. Interesuje się wszystkimi aspektami informatyki, ale jego największą pasją są języki programowania.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0479-8
9 788328 904798
Cena: 129,00 zł

Mc Graw Hill