ORACLE

Java

Java
Kompendium programisty
Wydanie XI

Herbert Schildt

Oracle Press

Helion

Tytuł oryginału: Java: The Complete Reference, Eleventh Edition

Tłumaczenie: Piotr Rajca, na podstawie „Java. Kompendium programisty”
w tłumaczeniu Rafała Jońcy i Mikołaja Szczepaniaka

ISBN: 978-83-283-5882-9

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javk11
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

CZĘŚĆ II
Biblioteka języka Java

Dodatki

ROZDZIAŁ

1

Historia i ewolucja języka Java

Aby zrozumieć język Java, trzeba najpierw dobrze zrozumieć powody jego powstania, modelujące go siły i spadek, który odziedziczył. Podobnie jak wcześniejsze popularne języki programowania komputerów, Java jest mieszanką najlepszych elementów swoich poprzedników, połączoną z innowacyjną koncepcją związaną z jej unikatową misją. Kolejne rozdziały niniejszej książki zajmują się praktycznymi aspektami języka — składnią, bibliotekami i aplikacjami. Ten rozdział jest inny, ponieważ wyjaśnia genezę powstania języka Java, pomaga zrozumieć jego znaczenie i rozwój przez wszystkie kolejne lata.

Choć język Java jest bez wątpienia ściśle powiązany ze środowiskiem internetowym, nie należy zapominać o tym, iż Java to przede wszystkim język programowania. Rozwój istniejących języków programowania, a także tworzenie nowych języków wynika z:

- chęci dostosowania się do zmieniającego się środowiska i zastosowań,
- implementacji udoskonaleń związanych ze sztuką programowania.

Jak się przekonasz, oba przedstawione powody miały bardzo podobny wpływ na tworzenie języka Java.

Rodowód Javy

Java jest powiązana z językiem C++, który wywodzi się bezpośrednio z języka C. Innymi słowy, Java odziedziczyła swój charakter po tych dwóch językach. Z języka C zaczerpnęła składnię, a z C++ większość elementów związanych z obiektowością. Co więcej, kilka aspektów charakterystyki języka Java powstało w odpowiedzi na jej poprzedników. To jednak nie wszystko — autorzy wyciągnęli wnioski z doświadczeń z wieloma innymi językami programowania stosowanymi przez kilka ostatnich dziesięcioleci. Ten podrozdział opisuje ciąg zdarzeń i sił, które doprowadziły do powstania języka Java. Jak się przekonasz, każda innowacja w modelu języka ma na celu rozwiązanie podstawowego problemu, którego nie udawało się rozwiązać poprzednim językom. Java nie jest tu wyjątkiem.

Narodziny nowoczesnego języka — C

Język C zelektryzował cały komputerowy świat. Jego siły oddziaływania nie należy lekceważyć, gdyż w znaczący sposób zmienił podejście do programowania. Powstanie języka C wynikało z potrzeby zastosowania strukturalnego i wydajnego języka wysokiego poziomu, który zastąpiłby kod asemblerowy przy pisaniu programów systemowych. Gdy projektuje się nowy język programowania, trzeba podjąć pewne ważne decyzje, odpowiadając sobie na poniższe pytania:

- Łatwość użycia czy siła?
- Bezpieczeństwo czy wydajność?
- Sztywność czy rozszerzalność?

Przed językiem C programiści musieli wybierać te języki, które były lepiej dostosowane do konkretnych zadań. Na przykład język FORTRAN bardzo dobrze nadawał się do pisania wydajnych programów matematycznych, ale nie radził sobie dobrze z programami systemowymi. Język BASIC był bardzo prosty w nauce, ale nie miał dużych możliwości, a brak strukturalności powodował, że nie nadawał się do pisania złożonych programów. Języki asemblerowe były bardzo wydajne, ale trudno było się ich nauczyć lub wygodnie przenosić między systemami. Poza tym wyszukiwanie błędów w programie asemblerowym jest niezmiernie trudne.

Innym typowym problemem wczesnych języków programowania takich jak BASIC, COBOL lub FORTRAN był brak ich dostosowania do zasad programowania strukturalnego — wszystkie te języki niemal na każdym kroku korzystały z instrukcji `GOTO` jako instrukcji sterującej. Z tego powodu programy napisane w tych językach miały tendencję do charakteryzowania się tak zwanym „kodem spaghetti" — wieloma dziwnymi skokami i rozgałęzieniami, które czyniły zrozumienie działania programu praktycznie niemożliwym. Istniejące w tamtym czasie języki strukturalne takie jak Pascal nie zostały zaprojektowane z myślą o wydajności i brakowało im pewnych elementów pozwalających pisać bardzo różne programy. (Standardowe odmiany języka Pascal dostępne w tamtym czasie nie nadawały się do pisania kodu działającego na poziomie systemu).

Innymi słowy, przed pojawieniem się języka C nie istniał żaden inny język na tyle wszechstronny, by mógł być stosowany niemal wszędzie. Potrzeba zaistnienia takiego języka ciągle wzrastała. Na początku lat 70. XX wieku rewolucja komputerowa nabierała rozpędu i popyt na oprogramowanie szybko przerastał możliwości ówczesnych programistów. Szczególnie w środowiskach akademickich pojawiły się próby stworzenia lepszego języka programowania komputerów. W tym czasie zaczęła pojawiać się inna, bardzo ważna siła — sprzęt komputerowy powszedniał i wydawało się, że wkrótce osiągnie masę krytyczną. Nie trzymano już komputerów pod kłódką. Po raz pierwszy programiści mieli praktycznie swobodny dostęp do mocy obliczeniowych (swoich komputerów). Taka wolność zachęca do eksperymentowania. Co więcej, umożliwia programistom pisanie własnych narzędzi. W „przeddzień" powstania języka C wszystko było przygotowane na przyjęcie nowego języka programowania.

Język C został wymyślony i po raz pierwszy zaimplementowany przez Dennisa Ritchiego na komputerze DEC PDP-11, działającym na systemie operacyjnym UNIX. Język C wykorzystał pewne elementy starszego języka o nazwie BCPL, wymyślonego przez Martina Richardsa. Z kolei język BCPL korzystał z elementów języka B autorstwa Kena Thompsona. Język C powstał na początku lat 70. XX wieku. Przez wiele lat standard języka C wyznaczał kompilator dostarczany wraz z systemem operacyjnym UNIX oraz książka *The C Programming Language* autorstwa Briana Kerninghana i Dennisa Ritchiego (Prentice-Hall, 1978)[1]. Język C doczekał się standaryzacji w grudniu 1989 roku, gdy został uznany przez instytut ANSI.

Dla wielu powstanie języka C to początek nowoczesnych języków programowania. Nareszcie nowy język programowania rozwiązał większość problemów z poprzednimi językami. Powstał użyteczny, wydajny i strukturalny język programowania, który nie był trudny do przyswojenia. Co więcej, był to język kierowany do *programistów*. Przed pojawieniem się języka C języki komputerowe były projektowane pod kątem zagadnień akademickich lub agend rządowych. Język C jest inny. Został zaprojektowany, zaimplementowany i wykorzystany przez praktykujących programistów, więc odpowiadał ich sposobowi pracy. Poza tym programiści ci dokładnie przetestowali, przemyśleli i poprawili cały język, gdyż miał być ich głównym środowiskiem pracy. Między innymi z tych powodów język C szybko zyskał wielu zapalonych zwolenników i został ciepło przyjęty przez całą społeczność programistów. W skrócie, język C został zaprojektowany przez programistów dla programistów. Jak się przekonasz, z językiem Java jest podobnie.

[1] Polskie wydanie: *Język ANSI C. Programowanie. Wydanie II*, Helion, Gliwice 2010 — *przyp. tłum.*

Język C++ — następny krok

Na przełomie lat 70. i 80. XX wieku język C stał się dominującym językiem programowania i w zasadzie jest często stosowany aż do dziś. Ponieważ język C okazał się takim sukcesem, zapewne wiele osób stawia sobie pytanie, dlaczego zaistniała potrzeba wymyślenia czegoś nowego? Odpowiedź jest prosta: **złożoność**. Ponieważ złożoność pisanych programów cały czas wzrastała, zachodziła potrzeba lepszego zarządzania nią. Język C++ był odpowiedzią na tę potrzebę. Aby zrozumieć, dlaczego zarządzanie złożonością leży u podstaw powstania języka C++, rozważmy następujący przykład.

Podejście do programowania uległo znaczącej zmianie od momentu wymyślenia komputera. Na przykład początkowo programowanie odbywało się ręcznie przez wpisywanie poszczególnych instrukcji maszynowych za pomocą przełączników. Gdy programy zawierały tylko do kilkuset instrukcji, takie podejście było wystarczająco dobre. Ponieważ jednak programy rozrastały się, wymyślono języki asemblerowe, aby programista nie musiał korzystać z instrukcji maszynowych, a jedynie ich reprezentacji symbolicznej. Ponieważ programy nadal się rozrastały, powstały języki wysokiego poziomu, gdzie programista miał więcej narzędzi ułatwiających zapanowanie nad całością aplikacji.

Pierwszym powszechnie stosowanym językiem był oczywiście FORTRAN. Choć okazał się olbrzymim krokiem w przód, z pewnością nie zachęcał do pisania łatwych i zrozumiałych programów. W latach 60. XX wieku pojawiła się koncepcja **programowania strukturalnego**. Została ona doprowadzona do perfekcji w takich językach jak C. Dzięki językom strukturalnym programista po raz pierwszy mógł stosunkowo łatwo pisać złożone programy. Niestety, nawet programowanie strukturalne ma swoje granice — po przekroczeniu pewnej złożoności programiści przestawali panować nad całym projektem. Na początku lat 80. XX wieku wiele projektów osiągnęło granice wytrzymałości modelu strukturalnego. Aby rozwiązać zaistniały problem, opracowano nową koncepcję — **programowanie obiektowe** (ang. *object-oriented programming — OOP*). Model obiektowy zostanie dokładniej omówiony w dalszej części książki; na tym etapie wystarczy krótkie wprowadzenie: programowanie obiektowe to metodyka pomagająca organizować złożone programy dzięki wykorzystaniu dziedziczenia, hermetyzacji i polimorfizmu.

Choć w ogólnej analizie język C jest doskonałym językiem programowania, próg złożoności, z którą potrafi sobie poradzić, jest w nim stosunkowo niski. Gdy kod programu zawiera od 25 do 100 tysięcy wierszy, zarządzanie nim w języku C staje się udręką. Język C++ pozwala przełamać tę barierę, gdyż ułatwia programiście ogarnięcie większych projektów.

Język C++ został opracowany przez Bjarne Stroustrupa w 1979 roku, gdy pracował on w laboratoriach Bella w Murray Hill w stanie New Jersey w USA. Stroustrup początkowo nazwał swój język „C z klasami". Jednak w 1983 roku zmienił nazwę na C++. Język C++ rozszerza C o elementy obiektowe. Ponieważ bazuje na języku C, dziedziczy po nim wszystkie cechy, atrybuty i zalety. Stanowi to główną podstawę sukcesu języka C++. Twórca języka C++ nie chciał wymyślać całkowicie nowego języka, a jedynie rozszerzyć już istniejący, doskonały język.

Podwaliny języka Java

Na przełomie lat 80. i 90. XX wieku programowanie obiektowe z wykorzystaniem języka C++ znajdowało się u szczytu. Przez moment wydawało się nawet, że programiści znaleźli wreszcie swój idealny język programowania. Ponieważ język C++ łączył w sobie wydajność i składnię języka C, a dodatkowo wprowadzał obiektowość, pozwalał tworzyć bardzo różne programy. Podobnie jak w przeszłości zmienił się jednak układ sił i trzeba było ponownie pchnąć ewolucję języków programowania do przodu. W ciągu zaledwie kilku lat strony WWW i ogólnie internet stały się bardzo popularne. To zdarzenie zapowiadało kolejną rewolucję w programowaniu.

Powstanie języka Java

Język Java został wymyślony w roku 1991 przez Jamesa Goslinga, Patricka Naughtona, Chrisa Wartha, Eda Franka i Mike'a Sheridana zatrudnionych w firmie Sun Microsystems. Wykonanie pierwszej działającej wersji zajęło 18 miesięcy. Początkowo język nosił nazwę Oak, ale w 1995 roku

został przemianowany na Java. Między początkową implementacją Oak pod koniec 1992 roku a publicznym ogłoszeniem języka Java wiosną 1995 roku w rozwój tego języka było zaangażowanych wiele osób. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin i Tim Lindholm to tylko kilka z osób, które najbardziej przyczyniły się do powstania dojrzałej wersji prototypu języka.

Zadziwiające jest to, że oryginalnym celem powstania języka Java nie był internet! Główną motywacją było wymyślenie języka niezależnego od platformy sprzętowej (neutralnego architektonicznie), który mógłby służyć do tworzenia oprogramowania dla różnych urządzeń domowego użytku, na przykład kuchenek mikrofalowych lub pilotów zdalnego sterowania. Nie jest tajemnicą, że jako kontrolerów używa się różnych rodzajów procesorów. Problem z językami C i C++ (a także wieloma innymi) polega na tym, że zostały one zaprojektowane do kompilowania dla konkretnej platformy sprzętowej. Choć kod napisany w języku C++ może zostać skompilowany na praktycznie każdym procesorze, wymaga to zastosowania kompilatora dostosowanego do tego procesora. Kompilatory są kosztowe i czasochłonne do napisania. Potrzebne było prostsze i wydajniejsze kosztowo rozwiązanie. Aby znaleźć odpowiednie rozwiązanie, Gosling i inni zaczęli prace nad przenośnym, niezależnym od platformy sprzętowej językiem, który mógłby posłużyć do napisania kodu działającego na różnych procesorach i systemach operacyjnych. Efektem tych prac było powstanie języka Java.

Gdy zaczęła się faza ustalania szczegółów związanych z Javą, pojawił się drugi, bardzo istotny czynnik, który odegrał bardzo ważną kwestię w kierunku rozwoju Javy. Czynnikiem tym były strony WWW. Gdyby internet nie zaczął dynamicznie rozwijać się dokładnie w tym samym czasie co implementacja Javy, nowy język być może zostałby zepchnięty do pozycji dziwacznego języka programowania elektroniki użytkowej. Wraz z rozwojem internetu pojawiła się potrzeba nowego rodzaju języka programowania, który byłby przenośny, gdyż internet nie rozróżnia systemów operacyjnych i rodzajów procesorów.

Większość programistów na początku swojej kariery uczy się, że w pełni przenośne programy, choć pożądane, są nieosiągalne. Potrzeba tworzenia wydajnych, przenośnych (niezależnych od platformy sprzętowej) programów jest niemal tak stara jak samo programowanie, ale często pozostawała z boku z powodu innych, pilniejszych problemów. Ponieważ później świat komputerów podzielił się na trzy części reprezentowane przez obozy Intela, Macintosha i systemów uniksowych, potrzeba przenośności została znacznie ograniczona, bo programiści rzadko opuszczali swoje obozy. Wraz z pojawieniem się internetu problem przenośności pojawił się ze wzmożoną siłą, ponieważ internet stanowi uniwersum skupiające różnej maści komputery, systemy operacyjne, procesory itp. Pomimo tej różnorodności użytkownicy chcą uruchamiać ten sam program niezależnie od tego, gdzie się znajdą. To, co dawniej było irytującym, ale mało istotnym problemem, nagle stało się znaczącą niedogodnością.

W roku 1993 dla osób pracujących nad projektem języka Java stało się oczywiste, że problemy przenośności związane z kontrolerami umieszczanymi w urządzeniach elektronicznych będzie można odnaleźć również w przypadku kodu programów dostępnych przez internet. W zasadzie przy niewielkim nakładzie środków można język przewidziany do rozwiązywania niewielkich problemów zmodyfikować na potrzeby internetu (duża skala). Innymi słowy, główny nacisk na rozwój języka przeszedł z elektroniki użytkowej na internet. Gdy więc pojawiła się potrzeba języka programowania niezależnego od architektury sprzętowej, język Java stał się naturalnym wyborem.

Jak już wcześniej wspomniałem, język Java dziedziczy wiele elementów po językach C i C++. Jest to celowe działanie. Projektanci języka wiedzieli, że zachowanie składni podobnej do języka C oraz zasad obiektowości znanych z języka C++ uczynią nowy język bardzo pociągającym dla całej rzeszy doświadczonych programistów języków C i C++. Poza tymi powierzchownymi podobieństwami Java posiada również wiele innych cech, które przyczyniły się wcześniej do sukcesu języków C i C++. Po pierwsze, została zaprojektowana, przetestowana i udoskonalona przez praktykujących programistów. Język wyrósł z potrzeb i doświadczenia ludzi, którzy go wymyślali. Innymi słowy, jest to język programistów. Po drugie, Java jest językiem spójnym. Po trzecie, język ten daje programiście pełną kontrolę (jeżeli nie uwzględnia się ograniczeń wprowadzanych przy uruchamianiu programu przez internet). Jeśli programuje się z głową, będzie to wyraźnie widać w napisanych programach. Oczywiście prawdziwa jest również sytuacja odwrotna. Ujmę to inaczej — Java nie jest językiem do nauki programowania, ale językiem dla profesjonalnych programistów.

Z racji podobieństw między językami Java i C++ niektóre osoby sądzą, że Java jest po prostu „internetową wersją języka C++". Jest to poważny błąd. Java jest inna zarówno w kwestiach praktycznych, jak i projektowych. Choć twórcy języka Java korzystali z doświadczeń języka C++, Java z pewnością nie stanowi rozszerzenia języka C++. Prosty przykład: Java nie jest w żaden sposób zgodna z językiem C++. Oczywiście podobieństwa są znaczne — programista, który wcześniej programował w języku C++, po przejściu do Javy poczuje się jak w domu. Java nie została zaprojektowana, by zastąpić język C++, ale raczej by rozwiązać pewnego rodzaju problemy. Język C++ rozwiązuje innego rodzaju problemy. Oba języki mogą współistnieć przez wiele lat.

Jak wspomniałem na początku rozdziału, języki komputerowe ewoluują z dwóch powodów: w związku ze zmianami środowiska i chęcią przeniesienia sztuki programowania na wyższy poziom. Java ma za zadanie zapełnić lukę związaną z pisaniem programów niezależnych od platformy sprzętowej, udostępnianych za pośrednictwem internetu. Z drugiej strony, wpływa ona także na sposób pisania programów — w szczególności rozszerza i doskonali paradygmat obiektowy znany z języka C++, wprowadza zintegrowaną obsługę przetwarzania wielowątkowego i udostępnia bibliotekę ułatwiającą dostęp do internetu. O wyjątkowości Javy nie decyduje jednak ten czy inny element — Java jest wyjątkowym językiem programowania jako całość. Jest wprost doskonałą odpowiedzią na wyzwania nowego, rozproszonego uniwersum przetwarzania komputerowego. Jest tym dla programowania w internecie, czym język C był dla programowania systemowego — rewolucyjną siłą zmieniającą świat.

Powiązanie z językiem C#

Wpływ języka Java jest coraz bardziej widoczny w świecie języków programowania. Wiele innowacyjnych cech, konstrukcji i pomysłów stało się częścią innych, nowszych języków. Sukcesu Javy po prostu nie można zignorować.

Bodaj najlepszym przykładem wpływu Javy na inne języki jest język C#, który został stworzony przez firmę Microsoft dla frameworku .NET. Język C# jest ściśle powiązany z Javą: ma bardzo podobną składnię, obsługuje programowanie rozproszone i korzysta z tego samego modelu obiektowego. Oczywiście istnieją pewne różnice między tymi językami, ale ogólne wrażenie jest bardzo podobne. To zapożyczenie koncepcji z Javy przez język C# jest chyba najlepszym dowodem, że język Java wywiera ogromny wpływ na sposób myślenia o językach programowania.

Jak Java wywarła wpływ na internet

Internet stanowił jakby odskocznię, dzięki której Java stała się jednym z głównych języków programowania. Okazuje się jednak, że także język Java nie pozostał dłużny i wywarł duży wpływ na internet. Java nie tylko pozwoliła uprościć programowanie rozwiązań internetowych, ale wprowadziła zupełnie nowy rodzaj programu sieciowego, tzw. aplet, który zmienił sposób postrzegania treści w internecie. Java oferuje też rozwiązania dla kilku najważniejszych problemów związanych z funkcjonowaniem internetu: przenośności i bezpieczeństwa. Przyjrzyjmy się nieco bliżej tym zagadnieniom.

Aplety Javy

W czasie, gdy Java powstawała, jedną z najbardziej fascynujących spośród jej możliwości były aplety. **Aplet** to specjalny rodzaj napisanego w Javie programu, który został zaprojektowany do przesyłania w internecie i automatycznego wykonywania wewnątrz przeglądarki WWW obsługującej Javę. Kiedy użytkownik klika odpowiedni link, aplet jest automatycznie pobierany i uruchamiany wewnątrz przeglądarki. Aplety w założeniu miały być małymi programami. Zwykle były stosowane do wyświetlania danych dostarczanych przez serwer, do obsługi danych wejściowych użytkownika lub do lokalnego wykonywania (bez udziału serwera) prostych funkcji, na przykład kalkulatora kosztów kredytu. Krótko mówiąc, aplet umożliwia przeniesienie części możliwości funkcjonalnych z poziomu serwera na poziom klienta.

Wprowadzenie apletów było ważne, ponieważ w tamtym czasie rozszerzyło zbiór obiektów, które można swobodnie przesyłać w cyberprzestrzeni. Ogólnie istnieją dwie szerokie kategorie obiektów przesyłanych pomiędzy serwerem a klientem: pasywne informacje i dynamiczne, aktywne programy. Kiedy na przykład użytkownik czyta wiadomość poczty elektronicznej, w praktyce przegląda pasywne dane. Nawet kiedy użytkownik pobiera jakiś program, kod tego programu wciąż ma postać pasywnych danych (przynajmniej do momentu uruchomienia). Zupełnie inaczej jest w przypadku apletu, czyli dynamicznego, samouruchamiającego się programu. Taki program jest aktywnym agentem na komputerze klienta, ale jego działanie jest inicjowane przez serwer.

W tym początkowym okresie stosowania języka Java aplety miały dla niego kluczowe znaczenie. Stanowiły ilustrację jego możliwości i zapewnianych przezeń korzyści, nadawały nowy, fascynujący wymiar stronom WWW i pozwalały programistom na nieskrępowane poznawanie wszelkich możliwości języka. Choć zapewne nawet dziś można jeszcze spotkać jakieś aplety, to jednak wraz z upływem czasu ich znaczenie bardzo zmalało. Z powodów, które przedstawię później, od JDK 9 aplety były wycofywane, a w JDK 11 całkowicie usunięto wsparcie dla nich.

Bezpieczeństwo

Niezależnie od tego, jak bardzo pożądane są dynamiczne, sieciowe programy, stanowią one poważny problem pod względem bezpieczeństwa i przenośności. Oczywistym jest, że nie można dopuścić, by program automatycznie pobierany z internetu i uruchamiany na komputerze klienta wyrządził jakieś szkody. Co więcej, taki program musi działać w wielu różnych środowiskach i systemach operacyjnych. Jak się przekonasz, w języku Java problem ten został rozwiązany w efektywny i elegancki sposób. Przyjrzyjmy się teraz każdemu z tych zagadnień nieco dokładniej, zaczynając od bezpieczeństwa.

Jak powszechnie wiadomo, pobranie „zwykłego” programu z internetu wiąże się z ryzykiem wprowadzenia do systemu wirusa, konia trojańskiego lub innego złośliwego kodu. Istotą problemu jest możliwość wykonywania groźnych operacji przez złośliwy kod, który zyskuje nieuprawniony dostęp do zasobów systemowych. Program wirusa może na przykład odczytać poufne informacje, jak numery kart kredytowych, salda rachunków bankowych czy hasła, przeszukując zawartość lokalnego systemu plików komputera. Aby było możliwe bezpieczne pobieranie i uruchamianie programów Javy na komputerze klienta, konieczne jest zapobieganie tego rodzaju atakom wykonywanym przy ich użyciu.

Projektanci Javy zapewnili taką ochronę, zamykając aplikacje w środowisku wykonawczym Javy, które wyklucza możliwość uzyskiwania dostępu do pozostałych zasobów komputera. (Sposób osiągnięcia tego celu zostanie omówiony nieco później). Możliwość pobierania programów bez ryzyka ewentualnych uszkodzeń to bodaj najbardziej innowacyjny aspekt Javy.

Przenośność

Przenośność to jeden z najważniejszych aspektów internetu, ponieważ z internetem łączy się mnóstwo różnych typów komputerów i systemów operacyjnych. Skoro program Javy ma być uruchamiany na niemal każdym komputerze połączonym z internetem, musi istnieć sposób przystosowania tego programu do działania w różnych systemach. Innymi słowy, niezbędny jest mechanizm pozwalający pobrać tę samą aplikację przez internet i wykonać ją na różnych procesorach, w różnych systemach operacyjnych i w różnych przeglądarkach. Tworzenie różnych wersji apletu dla różnych komputerów byłoby niepraktyczne. *Ten sam* kod aplikacji musi działać na *wszystkich* komputerach. W tej sytuacji konieczne jest stworzenie mechanizmu generowania przenośnego, wykonywalnego kodu. Jak się niedługo przekonamy, za zapewnianie przenośności odpowiada ten sam mechanizm, który ułatwia zapewnianie bezpieczeństwa.

Magia języka Java — kod bajtowy

Kluczem pozwalającym Javie rozwiązać problemy bezpieczeństwa i przenośności jest fakt, że wyjściem generowanym przez kompilator języka Java nie jest kod wykonywalny, ale kod bajtowy. **Kod bajtowy** to wysoce zoptymalizowany zbiór instrukcji zaprojektowanych do wykonywania przez system wykonawczy Javy nazywany **maszyną wirtualną Javy** (*JVM*, z ang. *Java Virtual Machine*). Maszyna wirtualna to tak naprawdę **interpreter kodu bajtowego**. Takie rozwiązanie jest o tyle zaskakujące, że z uwagi na wydajność większość nowoczesnych języków programowania jest kompilowana do kodu wykonywalnego, nie interpretowana. Z drugiej strony, rozwiązanie zaproponowane przez Javę (uruchamianie programów dzięki maszynie wirtualnej) rozwiązuje główne problemy związane z pobieraniem programów z internetu. Oto wyjaśnienie.

Przetłumaczenie programu do kodu bajtowego ułatwia uruchamianie go w wielu różnych środowiskach. Powód jest bardzo prosty: dla każdej platformy komputerowej należy jedynie zaimplementować maszynę wirtualną. Gdy istnieje pakiet uruchomieniowy dla danego systemu, można na nim uruchomić dowolny program napisany w Javie. Choć szczegóły związane z maszyną wirtualną nie muszą być takie same w każdym systemie, wszystkie maszyny potrafią poprawnie zinterpretować ten sam kod bajtowy. Gdyby program w języku Java był kompilowany do postaci wykonywalnej, należałoby wygenerować wiele wersji tego programu dla różnych systemów połączonych z internetem. Wygenerowanie wszystkich możliwych wariantów byłoby oczywiście niewykonalne. Wykonywanie kodu bajtowego przez maszynę wirtualną Javy jest więc najprostszym sposobem tworzenia naprawdę przenośnych programów.

Wykonywanie programu przez maszynę wirtualną Javy pomaga także zapewnić odpowiedni poziom bezpieczeństwa. Ponieważ to maszyna wirtualna wszystkim steruje, może zarządzać wykonywaniem programu. A zatem JVM może utworzyć ograniczone środowisko wykonawcze nazywane **piaskownicą** (ang. *sandbox*), które będzie zawierać program i ograniczać jego dostęp do lokalnego komputera. Co więcej, bezpieczeństwo jest dodatkowo zwiększane dzięki ograniczeniom istniejącym w samym języku Java.

Ogólnie, jeśli program zostanie skompilowany do postaci pośredniej i będzie interpretowany przez maszynę wirtualną, z pewnością będzie działał wolniej niż program skompilowany do postaci wykonywalnej. Na szczęście w przypadku Javy różnica w szybkości nie jest znacząca. Ponieważ kod bajtowy jest wysoce zoptymalizowany, maszyna wirtualna potrafi wykonywać go znacznie szybciej, niż mogłoby się wydawać.

Choć Java została początkowo zaprojektowana jako język interpretowany, w zasadzie nie istnieją przeciwwskazania techniczne uniemożliwiające kompilację „w locie" z kodu bajtowego na kod wykonywalny w celu zwiększenia wydajności. Z tego powodu firma Sun zaczęła wykorzystywać technologię HotSpot już wkrótce po pierwszym wydaniu Javy. HotSpot dostarcza kompilator typu JIT (*Just-In-Time*) dla kodu bajtowego. Gdy kompilator JIT stanowi część maszyny wirtualnej Javy, wybrane fragmenty kodu bajtowego są kompilowane w czasie rzeczywistym do postaci wykonywalnej. Warto pamiętać o tym, iż programy Javy nie są kompilowane do kodu wykonywalnego w całości. Zamiast tego kompilator JIT kompiluje tylko fragmenty kodu, gdy jest to naprawdę przydatne. Innymi słowy, nie jest kompilowany cały program, a jedynie te fragmenty, które naprawdę mogą zyskać na kompilacji. Pozostały kod jest po prostu interpretowany. Pomimo tych ograniczeń kompilacja JIT zapewnia znaczące przyspieszenie działania aplikacji. Nawet w przypadku kompilacji kodu bajtowego przenośność i bezpieczeństwo są nadal zachowane, ponieważ to maszyna wirtualna zarządza całym środowiskiem.

I jeszcze jedna uwaga: począwszy od JDK 9, niektóre środowiska Javy są wyposażone także w kompilator **z wyprzedzeniem** (ang. *ahead-of-time compiler*), który może być używany do kompilowania kodów bajtowych do kodu rodzimego nie podczas jego wykonywania, lecz jeszcze **zanim** to nastąpi. Taka kompilacja z wyprzedzeniem jest możliwością wyspecjalizowaną i nie zastępuje tradycyjnego, opisanego wcześniej sposobu kompilacji kodu Javy. Ze względu na wysoce wyspecjalizowany charakter tego rozwiązania kompilacja z wyprzedzeniem nie zostanie opisana w tej książce.

Wychodząc poza aplety

Minęły już dwie dekady od momentu wprowadzenia pierwszej wersji Javy, a w ciągu tych lat w języku wprowadzono wiele zmian. W czasach, 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 kilka lat. Jak można się spodziewać, także język Java ulegał zmianom, a wraz z nim zmieniały się 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. Obecnie 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, w którym to wsparcie dla apletów zostało całkowicie usunięte.

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, umożliwiało dynamiczne pobieranie aplikacji ze stron WWW. Był to mechanizm uruchomieniowy przydatny 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 one 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 można 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. Choć szczegółowy opis strategii wdrażania aplikacji wykracza poza ramy tematyczne niniejszej książki, to jednak jest to zagadnienie, na które warto zwracać uwagę podczas dalszej nauki Javy.

Szybszy harmonogram udostępniania

W Javie wprowadzono ostatnio 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. Natomiast po udostępnieniu JDK 9 ten okres 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. 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.

Obecnie nowe wersje Javy są zaplanowane na marzec i wrzesień każdego roku. W efekcie JDK 10 został udostępniony w marcu 2018 roku, czyli sześć miesięcy po JDK 9. Kolejna wersja, JDK 11, pojawiła się we wrześniu 2018 roku. Oczekuje się, że następne wersje będą się pojawiać co kolejne sześć miesięcy. 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 kolejnej wersji języka. Zapowiada się naprawdę ekscytujący okres dla programistów Javy!

Serwlety — Java po stronie serwera

Kod wykonywany po stronie klienta to tylko jedna strona równania klient-serwer. Niedługo po pierwszym wydaniu Javy było oczywiste, że jej innowacyjne cechy będą bezcenne także po stronie serwera. W efekcie powstała koncepcja **serwletu** (ang. *servlet*). Serwlet to niewielki program wykonywany na serwerze.

Serwlety służą do tworzenia dynamicznie generowanej treści, która jest następnie dostarczana do klienta. Na przykład sklep internetowy może używać serwletu do odnajdywania w bazie danych ceny określonego towaru. Informacje o cenie można następnie wykorzystać do dynamicznego wygenerowania strony internetowej, która zostanie wysłana do przeglądarki. Mimo że dynamiczne generowanie treści jest możliwe także za pomocą innych mechanizmów, na przykład interfejsu CGI (od ang. *Common Gateway Interface*), serwlet oferuje wiele istotnych zalet, w tym wyższą wydajność.

Ponieważ serwlety (jak wszystkie programy Javy) są kompilowane do kodu bajtowego i wykonywane przez maszynę wirtualną Javy, zapewniają niespotykaną gdzie indziej przenośność. Oznacza to, że ten sam serwlet może być stosowany w wielu różnych środowiskach serwerów. Jedynym wymaganiem jest obsługa przez serwer wirtualnej maszyny Javy i kontenera serwletów. Obecnie pisanie kodu wykonywanego po stronie serwera jest jednym z głównych zastosowań języka Java.

Hasła języka Java

Żaden opis historii języka Java nie może się obejść bez podania haseł przyświecających powstaniu tego języka. Choć głównymi motorami związanymi z powstaniem tego języka są: przenośność i bezpieczeństwo, inne czynniki także wpływały na ostateczny kształt Javy. Poniżej przedstawiam listę podstawowych haseł, którymi kierował się zespół opracowujący język Java:

- prostota,
- bezpieczeństwo,
- przenośność,
- obiektowość,
- niezawodność,
- wielowątkowość,
- neutralność architektury,
- interpretowalność,
- wysoka wydajność,
- rozproszenie,
- dynamika.

Dwa z tych haseł — bezpieczeństwo i przenośność — zostały już omówione. Pokrótce wyjaśnię więc pozostałe.

Prostota

Java została tak zaprojektowana, by była prosta do przyswojenia przez profesjonalnego programistę i by jednocześnie mogła być używana wydajnie. Gdy ma się jakiekolwiek doświadczenie programistyczne, opanowanie języka Java nie stanowi większego problemu. Jeśli dodatkowo dobrze rozumie się programowanie obiektowe, nauka Javy przebiegnie jeszcze szybciej. Co więcej, gdy wcześniej programowało się w języku C++, przestawienie się na język Java zajmuje tylko kilka dni. Ponieważ Java dziedziczy składnię po językach C i C++, a elementy obiektowe po języku C++, większość programistów nie ma najmniejszych problemów z jej opanowaniem.

Obiektowość

Choć na powstanie Javy miały wpływ inne języki, nie była ona projektowana w taki sposób, by jej kod źródłowy był zgodny z jakimkolwiek innym językiem. Zespół projektowy nie był obarczony dziedzictwem przeszłości i w ten sposób powstało czyste, użyteczne i pragmatyczne podejście do obiektowości. Swobodne czerpanie z pomysłów znanych z wielu wcześniejszych środowisk obiektowych pozwoliło językowi Java zachować równowagę między podejściem purystów („wszystko jest obiektem") a podejściem pragmatyków („zejdź mi z drogi"). Model obiektów w Javie jest prosty i rozszerzalny, natomiast typy proste (na przykład liczby całkowite) ze względów wydajnościowych nie są obiektami.

Niezawodność

Wieloplatformowe środowisko internetowe stawia niezwykłe żądania programom, ponieważ muszą one działać bez przeszkód na różnych systemach. Z tego powodu nadano duży priorytet takiemu zaprojektowaniu języka, by pisane w nim programy były niezawodne. Java ogranicza programistę w kilku kwestiach, ale jest to związane tylko z wymuszaniem szybszego znajdowania błędów. Z drugiej strony, pisząc program w języku Java, nie trzeba martwić się, że popełni się typowe błędy programistyczne. Ponieważ Java jest językiem ze ścisłą kontrolą typów, kod jest sprawdzany już na etapie kompilacji. Nie oznacza to jednak, że kod nie jest sprawdzany także w trakcie wykonywania programu. Wielu błędów, które są szczególnie trudne do wykrycia, ponieważ ujawniają się tylko w określonych przypadkach, po prostu nie można popełnić w Javie. Zapewnienie przewidywalnego sposobu działania aplikacji przez sam język jest jednym z kluczowych aspektów Javy.

Aby lepiej zrozumieć sposób zapewniania niezawodności w Javie, warto przeanalizować dwa główne powody powstawania błędów w programach: błędy związane z zarządzaniem pamięcią i źle obsłużone wyjątki (błędy wykonania programu). Zarządzanie pamięcią jest trudnym i niewdzięcznym zadaniem w tradycyjnych językach programowania. Na przykład w językach C i C++ programista często ręcznie alokuje i zwalnia dynamicznie przydzielaną pamięć. Czasami prowadzi to do problemów, ponieważ programiści albo zapominają zwolnić zaalokowaną wcześniej pamięć, albo co gorsza, zwalniają pamięć, która jest jeszcze używana przez inny fragment aplikacji. Java praktycznie eliminuje te problemy, gdyż zarządza alokacją i zwalnianiem pamięci za programistę. (W zasadzie zwalnianie pamięci jest w pełni automatyczne, gdyż Java zawiera mechanizm zwalniania nieużywanej pamięci). Wyjątki w tradycyjnych środowiskach wynikają często z próby dzielenia przez zero lub błędu „nie znaleziono pliku" i muszą być wychwytywane za pomocą niezgrabnych, trudnych w interpretacji konstrukcji. Java pomaga rozwiązać tego rodzaju problemy, zapewniając obiektową obsługę wyjątków. W dobrze napisanym programie w języku Java wszystkie błędy wykonania mogą — a w zasadzie powinny — być obsłużone przez program.

Wielowątkowość

Java została zaprojektowana tak, by spełnić rzeczywiste wymagania związane z tworzeniem interaktywnych, sieciowych aplikacji. W tym celu w sam język zostały wbudowane mechanizmy programowania wielowątkowego, aby program mógł wykonywać wiele zadań jednocześnie. System wykonawczy Javy dostarczany jest z eleganckim, choć złożonym rozwiązaniem synchronizacji

międzyprocesowej, który umożliwia tworzenie sprawnie działających aplikacji wielowątkowych. Dzięki wbudowaniu wszystkiego w sam język programista nie musi martwić się podsystemem wielozadaniowości, a jedynie odpowiednim zachowaniem programu.

Neutralność architektury

Główny nacisk w trakcie projektowania języka Java położono na żywotność i przenośność. Jednym z wielu problemów programistów jest to, iż nie ma gwarancji, że napisany dziś program będzie poprawnie działał także jutro (nawet na tym samym komputerze). Pojawiają się coraz to nowsze systemy operacyjne, procesory, dostępne zasoby, które mogą spowodować, że pogram przestanie poprawnie funkcjonować. Projektanci języka Java podjęli kilka konkretnych decyzji i tak zmodyfikowali język oraz maszynę wirtualną, aby można było uniknąć opisanych wcześniej sytuacji. Oto założenie projektantów języka: „napisz raz, uruchom gdziekolwiek, kiedykolwiek, wiecznie". Można powiedzieć, że w dużej mierze udało się spełnić to założenie.

Interpretowalność i wysoka wydajność

Jak wspomniałem wcześniej, język Java umożliwia tworzenie aplikacji wieloplatformowych dzięki kompilacji kodu źródłowego na reprezentację pośrednią nazywaną kodem bajtowym. Kod ten może być wykonywany w dowolnym systemie, dla którego napisano maszynę wirtualną Javy. Większość wcześniejszych rozwiązań wieloplatformowych znacząco traciła na wydajności. Jak wcześniej napisałem, kod bajtowy został starannie zaprojektowany w taki sposób, by był łatwy do konwersji na kod maszynowy za pomocą wydajnego kompilatora JIT. Systemy wykonawcze Javy udostępniają ten kompilator bez konieczności rezygnowania choćby z cząstki wieloplatformowości.

Rozproszenie

Java została zaprojektowana dla rozproszonego środowiska internetu, ponieważ obsługuje protokół TCP/IP. W zasadzie dostęp do zasobu za pomocą adresu URL niewiele różni się od dostępu do pliku. Java obsługuje **zdalne wywoływanie metod** (*RMI*, z ang. *Remote Method Invocation*), co umożliwia wywoływanie metod na odległych komputerach.

Dynamika

Programy napisane w języku Java zawierają znaczną ilość informacji przydatnych w czasie wykonywania aplikacji, które są używane do weryfikacji i dostępu do obiektów. Umożliwia to dynamiczne dołączanie kodu w bezpieczny i przewidywalny sposób. Jest to ważny element środowiska apletów, których fragmenty mogą być dynamicznie aktualizowane w trakcie pracy systemu.

Ewolucja Javy

Pierwsze wydanie Javy na pewno można nazwać rewolucją, ale na tym nie zakończył się rozwój całego języka. W odróżnieniu od innych systemów, które na ogół rozwijają się bardzo powoli, w sposób przyrostowy, Java ewoluuje w niezmiernie szybkim tempie. Niedługo po wydaniu Javy 1.0 projektanci mieli już gotową wersję 1.1. Liczba wprowadzonych zmian była tak duża, że z pewnością nie oddaje jej tak niewielka zmiana w numeracji wersji. W Javie 1.1 pojawiło się wiele nowych elementów biblioteki, ponownie zdefiniowano obsługę zdarzeń przez aplety i zmieniono wiele elementów wcześniejszej biblioteki. Wycofano (uznano za przestarzałe) kilka elementów oryginalnie zdefiniowanych w Javie 1.0. Innymi słowy, Java 1.1 zarówno dodała, jak i odjęła pewne elementy oryginalnej specyfikacji.

Kolejnym głównym wydaniem Javy była Java 2, gdzie liczba 2 oznaczała „drugą generację". Pojawienie się tego wydania przeniosło Javę do „nowej ery" i było naprawdę ważnym wydarzeniem. Pierwsze wydanie Javy 2 nosi numer 1.2. Może się to wydawać nieco dziwne, ale wynika po prostu z odmiennego numerowania wersji wewnątrz bibliotek, a innego dla samych nazw wydań. Wraz z tym wydaniem firma Sun zmieniła nazwę produktu na J2SE (Java 2 Platform Standard Edition) i numer wersji zaczął być dodawany właśnie do niej.

Java 2 przyniosła wiele nowych elementów, między innymi bibliotekę Swing i kolekcje (Collections Framework), a także rozszerzenie maszyny wirtualnej Javy i innych narzędzi. Wraz z wydaniem Javy 2 wycofano się z pewnych rozwiązań. Do najważniejszych decyzji tego typu należało zabronienie wykorzystywania metod `suspend()`, `resume()` i `stop()` klasy `Thread`.

Następne znaczące wydanie Javy to J2SE 1.3. Było to pierwsze znaczące uaktualnienie od wersji 1.2 i polegało głównie na rozbudowaniu istniejącej funkcjonalności oraz skonkretyzowaniu środowiska uruchomieniowego. Ogólnie istnieje zgodność kodu źródłowego programu napisanego w wersji 1.2 i wersji 1.3. Choć w tym wydaniu nie pojawiło się wiele zmian w samej składni języka, było ono nie mniej ważne od pozostałych.

Wydanie J2SE 1.4 ponownie zwiększyło możliwości Javy, gdyż zawierało kilka istotnych uaktualnień, poprawek i dodatków. Na przykład wprowadzono nowe słowo kluczowe `assert`, łańcuch wyjątków i kanałowy system wejścia-wyjścia. Dokonano także zmian w klasach kolekcji i sieci, a także wielu innych pomniejszych zmian. Mimo tych wszystkich poprawek wersja 1.4 zachowała niemalże stuprocentową zgodność kodu źródłowego z poprzednimi wersjami.

Kolejne wydanie Javy, oznaczone J2SE 5, można śmiało nazwać rewolucyjnym. Inaczej niż we wcześniejszych aktualizacjach, które — choć istotne — wprowadzały zmiany przyrostowo, J2SE 5 w zasadniczy sposób rozszerza zasięg, użyteczność i zakres języka. Aby zrozumieć rozmiar zmian w wersji J2SE 5, warto przyjrzeć się liście dodanych nowych elementów języka:

- typy sparametryzowane,
- metadane,
- automatyczne otaczanie i wydobywanie typów prostych,
- wyliczenia,
- rozszerzona pętla `for` typu `for-each`,
- zmienna liczba argumentów,
- import statyczny,
- formatowane wejście-wyjście,
- narzędzia związane ze współbieżnością.

Powyższa lista nie zawiera drobnych usprawnień czy udoskonaleń wprowadzanych przyrostowo. Każdy element listy reprezentuje znaczące rozszerzenie języka Java. Niektóre elementy, jak typy sparametryzowane, rozszerzona pętla `for` i zmienna liczba argumentów, to nowe elementy składni. Inne elementy, na przykład automatyczne otaczanie typów prostych, zmieniają semantykę języka. Metadane dodają całkowicie nowy wymiar programowania. We wszystkich przypadkach wpływ poszczególnych elementów wykracza poza ich bezpośrednie oddziaływanie. To wydanie zmienia charakter Javy.

Znaczenie tych nowych funkcji zostało uwzględnione w numeracji — nowa wersja otrzymała numer 5. Tradycyjnie nowa wersja powinna przyjąć numer 1.5. Jednak przejście z wersji 1.4 na 1.5 nie oddawałoby stopnia wprowadzonych zmian, więc zdecydowano się na przejście od razu do wersji 5. Nowa wersja produktu nosiła więc nazwę J2SE 5, a nazwa platformy programistycznej — JDK 5. Aby jednak zachować jednolitość w numeracji wersji biblioteki, **numer wewnętrzny nowego wydania** to 1.5. Innymi słowy, **zewnętrzny numer wydania** to 5, a wewnętrzny to 1.5.

Kolejne wydanie Javy nazwano Java SE 6. Firma Sun ponownie zdecydowała się na zmianę nazwy platformy języka Java. Jak łatwo zauważyć, zrezygnowano ze stosowanego wcześniej numeru 2. Oznacza to, że od tego wydania platforma nazywa się **Java SE**, a oficjalny produkt nosi nazwę **Java Platform, Standard Edition 6**. Pakiet Java Developer's Kit nazwano JDK 6. Tak jak w przypadku wydania J2SE 5 liczba 6 w nazwie Java SE 6 reprezentuje wersję produktu. Wewnętrznym numerem wersji w tym wydaniu jest 1.6.

Wydanie Java SE 6 zbudowano na bazie wydania J2SE 5, wprowadzając przyrostowo pewne usprawnienia. W wersji Java SE 6 nie dodano żadnych istotnych elementów do samego języka Java, ale rozszerzono biblioteki API, dodano wiele nowych pakietów i udoskonalono środowisko wykonawcze. W międzyczasie, w tzw. długim cyklu życia (to określenie typowe dla Javy) tej wersji wydano wiele aktualizacji. Ogólnie Java SE 6 jest jednym z wielu kroków w historii rozwoju platformy J2SE 5.

Kolejne wydanie Javy, które nazwano Java SE 7, obejmowało pakiet Java Developer's Kit nazwany JDK 7 i nosiło wewnętrzny numer wersji 1.7. Wersja Java SE 7 była pierwszym głównym wydaniem Javy od momentu przejęcia firmy Sun Microsystems przez koncern Oracle. Wydanie Java SE 7 zawierało wiele nowych elementów, w tym ważne dodatki do języka programowania i bibliotek API. Wprowadzono w nim też nowe rozwiązania w systemie wykonawczym Javy z myślą o obsłudze języków innych niż Java, jednak z perspektywy programistów Javy najbardziej interesujące były nowości w tym języku i w jego bibliotece.

Nowe elementy języka programowania opracowano w ramach **projektu Coin**. Celem tego projektu było wskazanie wielu drobnych zmian w języku Java, które należy wprowadzić wraz z wydaniem JDK 7. Mimo że nowe rozwiązania określono mianem drobnych zmian, ich wprowadzenie w jednym wydaniu miało całkiem spory wpływ na kod Javy. W rzeczywistości wielu programistów uznało, że właśnie te zmiany należały do najważniejszych cech wersji Java SE 7. Nowe elementy języka wprowadzone w tej wersji Javy wymieniono na poniższej liście:

- **łańcuchy** jako wyrażenia sterujące instrukcji `switch`;
- binarne stałe całkowitoliczbowe;
- znaki podkreślenia w stałych numerycznych;
- rozszerzone instrukcje `try` (nazwane `try`-*with-resources*), które obsługuje automatyczne zarządzanie zasobami (na przykład strumienie mogą teraz być automatycznie zamykane, kiedy nie są już potrzebne);
- wnioskowanie typu (za pomocą operatora `<>` — ang. *diamond operator*) podczas konstruowania egzemplarza typu sparametryzowanego;
- rozszerzona obsługa wyjątków, dzięki której jedna konstrukcja `catch` może przechwycić co najmniej dwa wyjątki (tzw. *multi-catch*), oraz lepsza weryfikacja typów w przypadku ponownie generowanych wyjątków;
- mimo że zmieniła się składnia Javy, poprawiono ostrzeżenia kompilatora dotyczące pewnych typów metod z konstrukcją *varargs* i przekazano programiście większą kontrolę nad tymi ostrzeżeniami.

Jak widać, mimo że elementy wprowadzone w ramach projektu Coin w założeniu miały stanowić drobne usprawnienia języka Java, korzyści wynikające z tych zmian były całkiem spore. W szczególności instrukcja `try`-*with-resources* zasadniczo wpłynęło na sposób tworzenia kodu operującego na strumieniach. Także możliwość używania łańcuchów (typu `String`) do sterowania instrukcją `switch` była jednym z długo oczekiwanych udoskonaleń, które w wielu sytuacjach pozwoliło znacznie uprościć kodowanie.

W wersji Java SE 7 wprowadzono też wiele zmian w bibliotece API Javy. Do najważniejszych nowości należały usprawnienia frameworku NIO i dodanie frameworku Fork/Join. Framework NIO (od ang. *New I/O*) dodano do Javy w wersji 1.4. Okazało się jednak, że zmiany zaproponowane w wydaniu Java SE 7 zasadniczo zwiększyły możliwości tych rozwiązań. Zmiany były na tyle znaczące, że zaczęto często stosować termin **NIO.2** w kontekście nowego frameworku.

Framework Fork/Join zawiera cenne mechanizmy wspomagające **programowanie równoległe**. Programowanie równoległe to popularna nazwa rozmaitych technik, których celem jest efektywne korzystanie z mocy komputerów dysponujących więcej niż jednym procesorem, w tym systemów wielordzeniowych. Zaletą środowisk wielordzeniowych jest perspektywa istotnego wzrostu wydajności programów. Framework Fork/Join wspomaga programowanie równoległe na dwa sposoby:

- upraszcza tworzenie i używanie zadań, które mogą być wykonywane współbieżnie;
- automatycznie wykorzystuje moc wielu procesorów.

Oznacza to, że za pomocą frameworku Fork/Join możesz ławo tworzyć skalowalne aplikacje, które będą automatycznie wykorzystywały procesory dostępne w środowisku działania. Oczywiście nie każdy algorytm może efektywnie wykorzystywać zalety przetwarzania równoległego, ale w niektórych przypadkach programowanie równoległe pozwala uzyskać ogromny wzrost wydajności algorytmów.

Kolejne wydanie Javy nosiło nazwę Java SE 8 i obejmowało pakiet JDK 8. Jego wewnętrznym numerem było 1.8. Stanowiło ono bardzo znaczącą aktualizację Javy, gdyż wprowadzało nowy element wywierający ogromny wpływ na cały język, a mianowicie **wyrażenia lambda** (ang. *lambda expressions*). Wpływ wyrażeń lambda był i będzie ogromny — powodują one zmianę nie tylko sposobu formułowania rozwiązań programowych, lecz także sposobu pisania kodu w Javie. Zgodnie z informacjami podanymi w rozdziale 15. wyrażenia lambda dodają do Javy możliwość programowania funkcyjnego. Jednocześnie stosowanie wyrażeń lambda może pozwolić na uproszczenie i skrócenie kodu koniecznego do tworzenia niektórych konstrukcji, takich jak niektóre rodzaje klas anonimowych. Wprowadzenie wyrażeń lambda spowodowało także dodanie do języka nowego operatora (`->`) oraz nowego elementu składni.

Dodanie wyrażeń lambda miało także głęboki wpływ na biblioteki Javy, do których dodano nowe opcje pozwalające na korzystanie z tych wyrażeń. Jedną z najważniejszych zmian było dodanie nowego API strumieni, umieszczonego w pakiecie `java.util.stream`. Zapewnia ono możliwość tworzenia potoków danych i zostało zoptymalizowane pod kątem wykorzystania wyrażeń lambda. Kolejnym nowym pakietem był `java.util.function`. Zawiera on definicje wielu **interfejsów funkcyjnych** (ang. *functional interfaces*), zapewniających dodatkowe wsparcie dla stosowania wyrażeń lambda. Także w innych miejscach API można znaleźć dalsze modyfikacje związane z wyrażeniami lambda.

Kolejna nowa możliwość języka powiązana z wprowadzeniem wyrażeń lambda jest związana z **interfejsami**. Zaczynając od JDK 8, można już definiować domyślną implementację metody udostępnianej przez interfejs. Jeśli programista nie określi własnej implementacji metody domyślnej, to zostanie zastosowana domyślna implementacja podana w interfejsie. Pozwala to na stopniowe, łagodne modyfikowanie interfejsów, do których można dodawać nowe metody bez konieczności wprowadzania zmian w już istniejącym kodzie. Ułatwia to także implementowanie interfejsów w przypadkach, gdy można zastosować rozwiązania domyślne. Kolejne zmiany wprowadzone w JDK 8 obejmują nowe API do obsługi dat i czasu, adnotacje typów, możliwość stosowania przetwarzania równoległego podczas sortowania tablic i tak dalej.

Kolejną wersją była Java 9. Towarzyszył jej pakiet dla programistów JDK 9. JDK 9 miał wewnętrzny numer 9. Była to ważna wersja Javy, zawierająca wiele znaczących usprawnień i rozszerzeń, obejmujących zarówno sam język, jak i jego biblioteki. Podobnie jak JDK 5 i JDK 8, także JDK 9 ma ogromny wpływ na język i na jego biblioteki.

Główną nowością wprowadzoną w JDK 9 były **moduły**, umożliwiające określanie powiązań i zależności różnych fragmentów kodu składających się na aplikację. Moduły wzbogaciły także o nowy wymiar mechanizm kontroli dostępu języka Java. Wprowadzenie modułów zaowocowało dodaniem do języka nowego elementu składniowego i kilku słów kluczowych. Oprócz tego do JDK dodany został nowy program narzędziowy o nazwie **jlink**, pozwalający tworzyć obrazy wykonywanych programów zawierające tylko i wyłącznie niezbędne moduły. Przygotowano także nowy typ pliku — JMOD. Wprowadzenie modułów miało też znaczący wpływ na bibliotekę API, gdyż, począwszy od JDK 9, biblioteka ta została zorganizowana z wykorzystaniem modułów.

Choć moduły stanowią kluczowe rozszerzenie wprowadzone w języku Java, to jednak pod względem koncepcyjnym są one całkiem proste. Co więcej, dzięki temu, że starszy kod, który nie korzysta z modułów, jest obsługiwany bez żadnych ograniczeń, moduły można zacząć wprowadzać do swojego cyklu wytwarzania aplikacji w dowolnym momencie. Nie trzeba natychmiast wprowadzać modułów do istniejącej już bazy kodu. Innymi słowy, moduły dodają znaczące możliwości funkcjonalne bez modyfikowania kwintesencji języka Java.

Oprócz modułów JDK 9 zawierał także kilka innych nowych możliwości. Do szczególnie interesujących należy JShell — narzędzie ułatwiające przeprowadzanie interaktywnych eksperymentów z kodem i naukę programowania w Javie. (Krótkie wprowadzenie do JShell można znaleźć w dodatku B). Inną ciekawą nowością są prywatne metody interfejsów, stanowiące kolejne rozszerzenie domyślnych metod interfejsów wprowadzonych w JDK 8. W JDK 9 wprowadzono również mechanizm wyszukiwania w dokumentacji generowanej przez narzędzie *javadoc* i powiązany z tą nową możliwością znacznik `@index`. Podobnie jak wszystkie inne wersje JDK, także JDK 9 zawiera wiele rozszerzeń wprowadzonych w bibliotece Javy.

Z reguły jest tak, że w każdej wersji języka Java największą uwagę przyciągają nowe możliwości. Niemniej jednak w JDK 9 jest pewna niezwykle ważna możliwość, z której zrezygnowano: aplety. Począwszy od tej wersji JDK, nie zaleca się już tworzenia apletów w nowych projektach. Zgodnie z wyjaśnieniami, które zamieściłem we wcześniejszej części tego rozdziału, w związku z coraz gorszym wsparciem dla apletów w przeglądarkach WWW (jak również z innych powodów), w JDK 9 cały interfejs programistyczny (API) służący do tworzenia apletów został uznany za przestarzały.

Kolejną wersją była Java SE 10 (JDK 10). Jak już wyjaśniłem, od JDK 10 kolejne wersje Javy mają się pojawiać według ścisłego harmonogramu, w którym odstępy pomiędzy wersjami głównymi wynoszą sześć miesięcy. W efekcie Java 10 została wydana w marcu 2018 roku, czyli sześć miesięcy po udostępnieniu JDK 9. Podstawową nowością dodaną w JDK 10 było wsparcie dla *wnioskowania typów zmiennych lokalnych*. Ta nowa cecha języka umożliwia wnioskowanie typu zmiennej lokalnej na podstawie jej inicjalizatora, co sprawia, że nie trzeba go już jawnie określać. Korzystanie z mechanizmu wnioskowania typów zmiennych lokalnych jest możliwe dzięki wprowadzeniu do Javy kontekstowego identyfikatora **`var`**, pełniącego rolę zarezerwowanej nazwy typu. Mechanizm wnioskowania typów może uprościć kod, gdyż eliminuje konieczność podawania nadmiarowych informacji o typach zmiennych, które można wywnioskować na podstawie inicjalizatorów. Może także uprościć deklaracje w przypadkach, kiedy typ jest trudny do określenia lub gdy nie można go jawnie podać. Wnioskowanie typów zmiennych lokalnych stało się powszechnie stosowanym elementem nowoczesnego stylu programowania. Wprowadzenie tego mechanizmu pomaga Javie dotrzymać kroku zmieniającym się trendom w projektowaniu języków programowania. Oprócz szeregu innych zmian w JDK 10 przedefiniowano także łańcuch wersji Javy — zmieniono znaczenie poszczególnych numerów wersji, tak by lepiej odpowiadały nowemu harmonogramowi udostępniania kolejnych wersji języka.

W momencie przygotowywania tej książki najnowszą wersją Javy była Java SE 11 (JDK 11). Została ona udostępniona we wrześniu 2018 roku, czyli sześć miesięcy po JDK 10. Podstawową nową możliwością języka wprowadzoną w JDK 11 jest wsparcie dla stosowania `var` w wyrażeniach lambda. Oprócz wielu zmian i modyfikacji w API Javy, w JDK 11 dodano nowe API sieciowe, co z pewnością zainteresuje szerokie grono programistów. API to, określane jako *HTTP Client API*, jest umieszczone w pakiecie `java.net.http`; udostępnia ono zaktualizowaną, rozszerzoną i poprawioną obsługę operacji sieciowych używanych przez klienty HTTP. Dodano także kolejny tryb działania dla programu uruchomieniowego Javy, który pozwala na bezpośrednie wykonywanie prostych programów zapisanych w jednym pliku źródłowym. Oprócz tego w JDK 11 usunięto też kilka wcześniej dostępnych możliwości. Ze względu na historyczne znaczenie zapewne najbardziej interesujące będzie usunięcie wsparcia dla apletów. Należy pamiętać, że od JDK 9 tworzenie apletów było już niezalecane. Jednak w JDK 11 wsparcie dla apletów całkowicie usunięto. Usunięto również wsparcie dla innej technologii służącej do uruchamiania aplikacji pisanych w Javie — Java Web Start. Ze uwagi na ciągły rozwój środowisk uruchomieniowych obie te technologie błyskawicznie traciły na znaczeniu i popularności. Kolejną znaczącą zmianą wprowadzoną w JDK 11 było usunięcie z niego biblioteki JavaFX. Obecnie ten framework GUI stanowi odrębny projekt typu open source. Ponieważ JavaFX nie jest już elementem JDK, nie będę jej opisywać w tej książce.

I jeszcze jedna uwaga dotycząca ewolucji Javy: w roku 2006 rozpoczęto proces przekształcania Javy w oprogramowanie typu open source. Obecnie dostępne są implementacje Javy stanowiące oprogramowanie open source. Uczynienie z Javy oprogramowania o otwartym kodzie źródłowym zwiększa dynamiczny charakter rozwoju tego języka. Ostateczne analizy pokazują, że tradycyjna już innowacyjność Javy nie jest zagrożona. Java pozostaje żywym, nowoczesnym językiem, czyli dokładnie takim, za jaki środowisko programistów przyzwyczaiło się ją uważać.

Informacje zamieszczone w tej książce zostały zaktualizowane o nowości wprowadzone w JDK 11. Jak już jednak pokazałem w powyższych rozważaniach, historia Javy jest pełna dynamicznych zmian. Podczas nauki Javy warto zatem obserwować nowości wprowadzane w jej kolejnych wersjach. Najprościej mówiąc: ewolucja Javy trwa!

Kultura innowacji

Już od samego początku Java była centrum kultury innowacji. Jej pierwsze wydanie zmieniło podejście do programowania dla internetu. Wirtualna maszyna Javy oraz kod bajtowy zmieniły sposób myślenia o bezpieczeństwie i przenośności. Przenaszalny kod spowodował, że strony WWW ożyły. System JCP (*Java Community Process*) zmienił sposób wprowadzania nowych elementów do języka. Świat Javy w zasadzie nigdy się nie zatrzymuje. Java SE 7 to najnowsza odsłona w trwającej, dynamicznej historii Javy.

Nie znajdziesz bardziej wyczerpującego omówienia Javy!

Mimo upływu lat Java wciąż pozostaje jednym z najważniejszych języków programowania, konsekwentnie wybieranym przez profesjonalnych deweloperów. Jest równocześnie nowoczesna i dojrzała. Twórcy Javy od początku jej istnienia stawiali na elastyczność i stale dostosowywali ten język do zmieniających się warunków pracy programistów. Od pierwszego wydania Java zapewnia narzędzia do programowania rozwiązań internetowych, jest więc naturalnym wyborem programistów tworzących aplikacje internetowe. Jej możliwości jednak są o wiele większe — i rosną z każdą kolejną wersją. Konieczne jest więc sukcesywne zapoznawanie się z nowościami i ze zmianami wprowadzanymi w poszczególnych wydaniach Java SE.

Ta książka jest jedenastym wydaniem wyczerpującego kompendium programisty Javy, w pełni zaktualizowanym, uzupełnionym o nowości wprowadzone w Java SE 11. Opisuje język kompleksowo: jego składnię, słowa kluczowe oraz najistotniejsze zasady programowania. Zawiera także informacje o najważniejszych składnikach biblioteki Javy, takich jak klasy wejścia-wyjścia, framework Collections, biblioteka strumieni oraz narzędzia programowania współbieżnego. Oczywiście szczegółowo został tu opisany inny niezwykle ważny element — system modułów Javy. Przedstawiono też interaktywne narzędzie programistyczne JShell. Podręcznik jest przejrzysty, napisany jasnym i zrozumiałym językiem, co zdecydowanie ułatwia naukę. Co ważne, poszczególne zagadnienia zilustrowano licznymi przykładowymi fragmentami kodu źródłowego. Z tak przygotowanego materiału skorzystają wszyscy programiści Javy, zarówno początkujący, jak i profesjonalni deweloperzy.

W tej książce między innymi:

- zasady programowania obiektowego
- klasy wejścia-wyjścia oraz obsługa wyjątków
- interfejsy i pakiety
- wnioskowanie typów zmiennych lokalnych
- obsługa zdarzeń, moduły i wyrażenia lambda
- AWT, Swing, JavaBean i serwlety

Herbert Schildt jest niekwestionowanym autorytetem w dziedzinie programowania w Javie. Od ponad trzydziestu lat pisze cenione książki do nauki tego języka, a także C, C++ i C#. Pasjonuje się wszystkim, co jest związane z przetwarzaniem komputerowym, zwłaszcza językami programowania. Od lat aktywnie działa na rzecz ich standaryzacji.

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 978-83-283-5882-9

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 199,00 zł

Java™
Oracle Press™