Java. Kompendium programisty

Autor: Herbert Schildt
Tłumaczenie: Rafał Jońca, Mikołaj Szczepaniak, Jakub Thiele-Wieczorek
ISBN: 83-7361-862-7
Tytuł oryginału: Java: The Complete Reference, J2SE 5 Edition (Complete Reference)
Format: B5, stron: 1112

Popularność języka Java stale rośnie. Programiści z całego świata wykorzystują go do tworzenia zarówno prostych aplikacji, jak i złożonych systemów. Podstawowa zaleta Javy — przenośność kodu — powoduje, że programy napisane w Javie możemy spotkać nie tylko na dyskach komputerów i serwerów, ale również w telefonach komórkowych i innych urządzeniach mobilnych. Java jest ciągle rozwijana — w każdej kolejnej wersji pojawiają się nowe elementy, ułatwiające realizację coraz bardziej złożonych zagadnień programistycznych.

„Java. Kompendium programisty" to doskonały przewodnik po najnowszym wcieleniu języka Java, noszącym oznaczenie J2SE5. Każdy twórca aplikacji w Javie znajdzie tu niezbędne do swojej pracy informacje. Autor — Herb Schildt, znany z wielu best-sellerowych pozycji dotyczących programowania w Javie i C++ — opisuje wszystkie elementy języka Java w wersji 5. Typy danych, metody, konstrukcje, podstawowe biblioteki i techniki programistyczne — wszystko zostało opisane prostym i zrozumiałym językiem oraz zilustrowane przykładami.

- Historia języka Java
- Podstawowe założenia programowania obiektowego
- Typy danych i zmienne
- Operatory
- Klasy, metody, pakiety i interfejsy
- Wątki i wyjątki
- Elementy bibliotek Javy
- Operacje wejścia i wyjścia
- Programowanie sieciowe
- Biblioteki AWT i Swing
- JavaBeans i serwlety

W tym podręczniku znajdziesz odpowiedzi na wszystkie pytania związane z Javą.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Typy danych, zmienne i tablice

Ten rozdział dotyczy trzech najbardziej podstawowych elementów Javy: typów danych, zmiennych i tablic. Podobnie jak wszystkie nowoczesne języki programowania, Java obsługuje kilka podstawowych typów danych. Używamy tych typów do deklarowania zmiennych i tworzenia tablic. Rozwiązanie proponowane w tej kwestii przez Javę jest czyste, wydajne i spójne.

Java to język ze ścisłą kontrolą typów

Trzeba podkreślić, iż Java jest językiem ze ścisłą kontrolą typów. Między innymi właśnie z tego stwierdzenia bierze się bezpieczeństwo i solidność Javy. Przekonajmy się, co to oznacza. Po pierwsze, każda zmienna ma ściśle określony typ, każde wyrażenie ma typ. Każdy typ jest dokładnie zdefiniowany. Po drugie, wszystkie przypisania, jawne lub przez parametry w wywołaniach metod, są sprawdzane pod kątem zgodności typów. Nie istnieje automatyczna konwersja niezgodnych typów występująca w niektórych innych językach programowania. Kompilator Javy sprawdza wszystkie wyrażenia i parametry, aby sprawdzić, czy typy są zgodne. Wszelkie niezgodności powodują zgłoszenie błędów, które muszą zostać poprawione, by doprowadzić kompilację klasy do szczęśliwego końca.

Typy proste

Język Java definiuje osiem **prostych** typów danych: `byte`, `short`, `int`, `long`, `float`, `double` i `boolean`. Typy proste nazywane są czasem typami **podstawowymi**. W książce oba stwierdzenia będą stosowane zamiennie. Typy proste dzielimy na cztery grupy:

- typy całkowite — do tej grupy należą typy `byte`, `short`, `int` i `long`, które zawsze określają liczby całkowite ze znakiem;
- typy zmiennoprzecinkowe — do tej grupy należą typy `float` i `double`, które reprezentują liczby z ułamkami;
- typy znakowe — ta grupa zawiera tylko typ `char`, który reprezentuje pojedynczy znak, na przykład literę lub cyfrę;
- typy logiczne — ta grupa zawiera tylko typ `boolean`, który jest specjalnym typem reprezentującym wartości typu prawda lub fałsz.

Przedstawione typy można stosować w oryginalnej postaci, tworzyć z nich tablice lub na ich podstawie kreować własne klasy (klasa definiuje nowy typ danych). Innymi słowy, typy te stanowią podstawę wszelkich innych typów tworzonych w programie.

Typy proste reprezentują pojedyncze wartości — nie są złożonymi obiektami. Jest to w zasadzie jedyne odstępstwo od w pełni obiektowego modelu w języku Java. Powód takiego rozwiązania jest bardzo prosty — wydajność. Uczynienie z typów prostych obiektów spowodowałoby zbyt znaczący spadek wydajności.

Typy proste mają ściśle określony zakres wartości i zachowanie w operacjach matematycznych. Języki takie jak C lub C++ pozwalają, aby rozmiar typu `int` zmieniał się w zależności od wykorzystywanego środowiska. W Javie jest inaczej. Przenośność programów Javy wymaga, aby wszystkie typy miały ściśle zdefiniowany zakres. Na przykład typ `int` jest zawsze 32-bitowy, niezależnie od platformy sprzętowej. W ten sposób programista ma pewność, że napisany przez niego program będzie działał poprawnie na innej architekturze systemowej **bez dokonywania żadnych modyfikacji**. Choć takie wymuszanie rozmiaru typów całkowitych potrafi zmniejszyć wydajność w pewnych środowiskach, jest wymogiem przenośności.

Przyjrzyjmy się po kolei poszczególnym typom danych.

Typy całkowite

Java definiuje cztery typy całkowite: `byte`, `short`, `int` oraz `long`. Wszystkie te typy mogą przechowywać wartości ze znakiem, czyli liczby ujemne i dodatnie. Java nie obsługuje typów bez znaku. Choć większość innych języków programowania obsługuje liczby całkowite z i bez znaku, projektanci Javy stwierdzili, że takie rozróżnienie nie jest potrzebne. Koncepcja **bez znaku** była najczęściej stosowana do sterowania zachowaniem **najbardziej znaczącego bitu** liczby, który definiuje **znak**. W rozdziale 4. dokładnie omówię operator przesunięcia w prawo bez znaku, który praktycznie eliminuje potrzebę stosowania typów bezznakowych.

Rozmiaru typu całkowitego nie należy rozumieć jako liczby bitów zajmowanych przez dany typ, ale raczej jako **zachowanie** zdefiniowane dla danego typu. Środowisku wykonawcze Javy może wewnętrznie używać dowolnej liczby bitów dla danego typu, o ile zachowuje się on tak, jak powinien. W rzeczywistości typy `byte` i `short` są zewnętrznie

zaimplementowane jako liczby 32-bitowe (zamiast 8- i 16-bitowych), aby poprawić wydajność, gdyż właśnie taki jest rozmiar słowa większości komputerów osobistych.

Rozmiar i zakres typów całkowitych są bardzo różne, co przedstawia tabela 3.1.

Tabela 3.1. *Rozmiar i zakres typów całkowitych*

Nazwa	Rozmiar	Zakres
`long`	64	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
`int`	32	od –2 147 483 648 do 2 147 483 647
`short`	16	od –32 768 do 32 767
`byte`	8	od –128 do 127

Przyjrzyjmy się poszczególnym typom liczb całkowitych.

Typ byte

Najmniejszym typem całkowitym jest `byte`. Jest to 8-bitowy typ znakowy od zakresie od –128 do 127. Zmienne typu `byte` są szczególne przydatne wtedy, gdy przetwarza się strumień danych odczytany z pliku lub otrzymany przez sieć. Poza tym przydaje się do obróbki surowych danych binarnych, które mogą nie być bezpośrednio zgodne z innymi wbudowanymi typami.

Zmienne bajtowe deklaruje się za pomocą słowa kluczowego `byte`. Poniższy przykład deklaruje dwie zmienne bajtowe o nazwach `b` i `c`.

```
byte b, c;
```

Typ short

Typ `short` to typ 16-bitowy o zakresie od –32 768 do 32 767. Prawdopodobnie jest to najrzadziej stosowany typ danych, ponieważ jest on tak zdefiniowany że jego pierwszy bit jest najbardziej znaczący (tak zwany format **big-endian**). Typ ten jest najlepszy dla komputerów 16-bitowych, które w zasadzie już wymarły.

Oto kilka przykładów deklaracji zmiennych tego typu.

```
short s;
short t;
```

Określenia endian wskazują na sposób w przechowywania w pamięci typów wielobajtowych takich jak `short`, `int` lub `long`. Typ `short` składa się z dwóch bajtów, ale który z tych bajtów ma być bardziej znaczący: pierwszy czy drugi? Mówimy, że procesor jest typu **big-endian**, jeśli najpierw pojawia się bardziej znaczący bajt, a dopiero po nim mniej znaczący. Procesory takie jak SPARC lub PowerPC są typu **big-endian**, natomiast procesory serii Intell x86 są typu **little-endian**.

Typ int

Prawdopodobnie najczęściej stosowanym typem jest 32-bitowy typ `int` o zakresie od –2 147 483 648 do 2 147 483 647. Poza innymi zastosowaniami, zmienne tego typu są najczęściej wykorzystywane w strukturach sterujących lub jako indeksy tablic. Jeśli jakieś wyrażenie zawiera zlepek zmiennych typu `byte`, `short`, `int` i stałych liczb, jest ono automatycznie konwertowane do typu `int` przed dokonaniem jakichkolwiek obliczeń.

Typ `int` jest najbardziej elastyczny i wydajny, więc powinien być stosowany zawsze wtedy, gdy trzeba zliczać wartości, przechodzić po kolejnych elementach tablicy lub wykonywać działania arytmetyczne na liczbach całkowitych. Choć może się wydawać, iż typy `short` i `byte` pozwalają zaoszczędzić miejsce, nie ma żadnej gwarancji, iż wewnętrznie nie będą reprezentowane przez typ `int`. Pamiętaj, że typ określa zachowanie, a nie rzeczywisty rozmiar. (Jedynym wyjątkiem są tablice, w których gwarantuje się, że typ `byte` będzie zajmował tylko jeden bajt na element, typ `short` 2 bajty na element, a typ `int` 4 bajty na element).

Typ long

Typ `long` to 64-bitowy typ stosowany wszędzie tam, gdzie wiadomo, iż zakres typu `int` nie jest wystarczający. Zakres typu `long` jest naprawdę imponujący, więc przydaje się w momencie wykonywania obliczeń na bardzo dużych liczbach. Poniżej znajduje się przykładowy program, który oblicza liczbę kilometrów, jaką przebędzie światło w podanej liczbie dni.

```
// Obliczanie odległości przebytej przez światło za pomocą zmiennych typu long.
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // przybliżona prędkość światła w kilometrach na sekundę
    lightspeed = 186000;

    days = 1000; // określenie liczby dni

    seconds = days * 24 * 60 * 60; // konwersja na sekundy

    distance = lightspeed * seconds; // obliczenie odległości

    System.out.print("W " + days);
    System.out.print(" dni światło przebędzie ");
    System.out.println(distance + " kilometrów.");
  }
}
```

Wykonanie programu spowoduje wyświetlenie następującego komunikatu.

```
W 1000 dni światło przebędzie 25920000000000 kilometrów.
```

Z pewnością taki wynik nie zmieściłby się w zmiennej typu `int`.

Typy zmiennoprzecinkowe

Liczby zmiennoprzecinkowe, nazywane również liczbami **rzeczywistymi**, stosowane są zawsze tam, gdzie potrzebna jest informacja z dokładnością do ułamków liczb. Wykonywanie obliczeń takich jak pierwiastek kwadratowy lub sinus wymaga zastosowania typu o bardzo wysokiej precyzji, czyli typu zmiennoprzecinkowego. Java implementuje standardowy (IEEE-754) zbiór typów i operatorów zmiennoprzecinkowych. Istnieją dwa typy zmiennoprzecinkowe, `float` i `double`, które odpowiadają liczbom pojedynczej i podwójnej precyzji. Tabela 3.2 przedstawia rozmiar i zakresy obu typów.

Tabela 3.2. *Rozmiary i zakresy typów zmiennoprzecinkowych*

Nazwa	Rozmiar	Przybliżony zakres
`double`	64	od 4.9e–324 do 1.8e+308
`float`	32	od 1.4e–045 do 3.4e+038

Oba typy omawiam w kolejnych punktach.

Typ float

Typ `float` określa wartość zmiennoprzecinkową **pojedynczej precyzji**, która używa 32 bajtów pamięci. Pojedyncza precyzja jest szybsza na niektórych procesorach i zajmuje dwa razy mniej miejsca od podwójnej precyzji, ale z drugiej strony jest mniej dokładna dla bardzo małych lub bardzo dużych wartości. Zmienne typu `float` przydają się wtedy, gdy potrzebna jest część ułamkowa, ale nie zależy nam na dużej precyzji. Na przykład, ten typ doskonale nadaje się do reprezentacji złotówek i groszy.

Oto przykład deklaracji zmiennych typu `float`.

```
float hifhtemp, lowtemp;
```

Typ double

Zmienne o podwójnej precyzji zajmującej 64-bity oznacza się za pomocą słowa kluczowego `double`. Niektóre nowoczesne procesory potrafią szybciej wykonywać obliczenia na liczbach podwójnej precyzji, gdyż zostały odpowiednio zoptymalizowane. Wszystkie funkcje trygonometryczne i nie tylko — `sin()`, `cos()`, `sqrt()` — zwracają wartości typu `double`. Jeśli trzeba zachować dobrą dokładność dla wielu iteracyjnie wykonywanych obliczeń lub obsługiwać bardzo duże liczby, typ `double` jest najlepszym wyborem.

Poniższy program wykorzystuje zmienne typu `double` do obliczenia pola koła.

```
// Obliczenie pola koła.
class Area {
  public static void main(String args[]) {
    double pi, r, a;
```

```
    r = 10.8; // promień koła
    pi = 3.1416; // przybliżona wartość pi
    a = pi * r * r; // obliczenie pola

    System.out.println("Pole koła wynosi " + a);
  }
}
```

Typ znakowy

W Javie typem danych używanym do przechowywania znaków jest typ `char`. Programiści języka C lub C++ powinni jednak uważać, gdyż typ ten **nie** oznacza dokładnie tego samego, co w języku C. W języku C lub C++ typ `char` to 8-bitowy typ całkowity, ale w Javie jest całkiem inaczej, gdyż do reprezentacji znaków używa się unikodu. Unikod (ang. *Unicode*) definiuje pełny zbiór znaków używanych przez większość języków, jakimi obecnie posługują się ludzie na całym świecie. Jest to unifikacja dziesiątek dawnych zbiorów znaków: łacińskiego, greckiego, arabskiego, cyrylicy, hebrajskiego, Katakana, Hangul i wielu innych. Z tego powodu unikod wymaga 16 bitów. Ponieważ Java obsługuje unikod, typ `char` jest 16-bitowy i ma zakres od 0 do 65 535. Nie istnieją ujemne wartości znaków. Zbiór znaków ASCII znajduje się w zakresie od 0 do 127, a 8-bitowy zbiór znaków ISO Latin 1 znajduje się w zakresie od 0 do 255. Ponieważ Java została tak zaprojektowana, aby jej aplety mogły być uruchamiane na całym świecie, zastosowanie unikodu do reprezentacji znaków wydaje się sensownym wyborem. Oczywiście unikod jest mniej wydajny dla języków takich jak angielski, niemiecki czy francuski, które łatwo mieściłyby się w 8 bitach. Zapewnienie globalnej przenośności ma swoją cenę.

Więcej informacji na temat kodowania znaków Unicode znajduje się na witrynie *www.unicode.org*.

Poniżej znajduje się prosty program obrazujący wykorzystanie zmiennych typu `char`.

```
// Przykład użycia typu danych char.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // kod dla X
    ch2 = 'Y';

    System.out.print("ch1 i ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

Wykonanie programu spowoduje wyświetlenie następującego wyniku.

```
ch1 i ch2: X Y
```

Zauważ, że zmiennej ch1 została przypisana wartość 88, która w kodzie ASCII i Unicode odpowiada znakowi X. Jak wspomniałem, zbiór znaków ASCII zajmuje pierwsze 127 znaków zbioru znaków Unicode. Z tego powodu w Javie nadal można stosować wiele sztuczek ze znakami.

Choć typ `char` nie jest typem całkowitym, to jednak w wielu sytuacjach działa dokładnie tak samo jak typ `int`. Umożliwia dodanie dwóch znaków do siebie lub zwiększenie wartości zmiennej znakowej. Rozważmy następujący program.

```
// Zmienne typu char zachowują się jak zmienne typu int.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 zawiera znak " + ch1);

    ch1++; // inkrementacja ch1
    System.out.println("teraz ch1 ma wartość " + ch1);
  }
}
```

Wykonanie programu spowoduje wyświetlenie następującego wyniku.

```
ch1 zawiera znak X
teraz ch1 ma wartość Y
```

W programie zmienna `ch1` najpierw przyjmuje wartość `'X'`. Następnie jest inkrementowana. Po tej operacji zmienna `ch1` zawiera wartość `'Y'`, czyli następny znak w ciągu znaków ASCII (i Unicode).

Typ logiczny

Java posiada typ prosty dla wartości logicznych o nazwie `boolean`. Może on przyjąć tylko jedną z dwóch wartości: `true` (prawda) lub `false` (fałsz). Ten właśnie typ zwracają wszystkie operatory relacji, na przykład `a < b`. Co więcej, typ `boolean` jest **wymagany** przez wszystkie wyrażenia warunkowe w strukturach sterujących takich jak `if` oraz `for`.

Poniższy program obrazuje wykorzystanie typu `boolean`.

```
// Przykład użycia typu boolean.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b wynosi " + b);
    b = true;
    System.out.println("b wynosi " + b);

    // wartość logiczna potrafi sterować instrukcją if
```

```
    if(b) System.out.println("To zostało wykonane.");

    b = false;
    if(b) System.out.println("To nie zostało wykonane.");

    // wynik operatora relacji jest wartością logiczną
    System.out.println("10 > 9 to wartość " + (10 > 9));
  }
}
```

Wykonanie programu spowoduje wyświetlenie następującego wyniku.

```
b wynosi false
b wynosi true
To zostało wykonane.
10 > 9 to wartość true
```

W tym programie warto zauważyć trzy interesujące kwestie. Po pierwsze, próba wyświetlenia zawartości zmiennej logicznej powoduje wyświetlenie tekstu „true" lub „false". Po drugie, sama wartość zmiennej wystarcza do sterowania działaniem instrukcji `if`. Nie trzeba na przykład pisać następującej konstrukcji.

```
if(b == true) ...
```

Po trzecie, wynik działania operatora relacji, na przykład <, jest wartością typu `boolean`. Z tego powodu wyrażenie `10 > 9` zawsze zwróci wartość „true". Potrzebujemy dodatkowych nawiasów wokół `10 > 9`, ponieważ operator + ma wyższy priorytet niż operator >.

Bliższe spojrzenie na literały

W rozdziale 2. pokrótce wspomniałem o literałach. Ponieważ omówiłem już podstawowe typy danych, warto dokładniej się im przyjrzeć.

Literały będące liczbami całkowitymi

Liczby całkowite są zapewne najczęściej stosowanym typem danych w każdym programie komputerowym. Każda liczba całkowita wpisana na stałe w programie jest literałem liczbowym. Kilka przykładów: `1`, `2`, `3` i `42`. Są to wartości dziesiętne, czyli liczby o podstawie 10. W Javie w literałach całkowitych można stosować jeszcze dwie inne podstawy: **ósemkową** (podstawa 8) i **szesnastkową** (podstawa 16). Wartości ósemkowe muszą zaczynać się od 0. Liczby dziesiętne nie mogą mieć na początku cyfry 0. Z tego powodu napisanie `09` spowoduje zgłoszenie błędu kompilatora, ponieważ 9 wykracza poza zakres liczb ósemkowych (od 0 do 7). Liczby szesnastkowe są stosowane przez programistów znacznie częściej niż ósemkowe, gdyż ułatwiają rozróżnienie poszczególnych bajtów liczby. Wartości szesnastkowe muszą zaczynać się od konstrukcji `0x` lub `0X`. Zakres dla liczb szesnastkowych wynosi od 0 do 15, przy czym litery od A do F (lub od a do f) zastępują wartości od 10 do 15.

Literały całkowite tworzą wartość typu `int`, czyli 32-bitową liczbę całkowitą. Ponieważ Java jest językiem o ścisłej kontroli typów, niektóre osoby dziwią się, że można przypisać literał całkowity do innego typu, takiego jako `byte` lub `long`, bez powodowania błędu kompilatora. Po prostu kompilator potrafi sobie poradzić z takimi sytuacjami. Gdy literał przypisuje się do typu `byte` lub `short`, błąd nie jest generowany, jeśli literał reprezentuje wartość z przedziału docelowego typu. Literał zawsze zostanie poprawnie przypisany do typu `long`. Aby jednak określić literał typu `long`, trzeba jawnie wskazać kompilatorowi, że dana wartość jest typu `long`. W tym celu należy dodać literę `l` lub `L` na końcu literału, na przykład `0x7fffffffffffffffL` lub `9223372036854775807L` w celu przypisania największej dopuszczalnej wartości typu `long`.

Literały zmiennoprzecinkowe

Liczby zmiennoprzecinkowe to wartości dziesiętne wraz z częścią ułamkową. Mogą zostać podane w notacji standardowej lub naukowej. **Notacja standardowa** wymaga podania części całkowitej liczby i po kropce części ułamkowej, na przykład 2.0, 3.14150 lub 0.6667. **Notacja naukowa** używa notacji standardowej, czyli liczby zmiennoprzecinkowej, ale dodatkowo zawiera informację o potędze liczby 10, przez której wartość trzeba pomnożyć wcześniej podaną liczbę. Eksponentę wskazuje się za pomocą znaku `E` lub `e`, po którym występuje dodatnia lub ujemna liczba całkowita oznaczająca potęgę. Oto przykłady notacji naukowej: `6.022E23`, `314159E-05` i `2e+100`.

Domyślnie literały zmiennoprzecinkowe traktowane są jako wartości podwójnej precyzji (typ `double`). Aby wymusić pojedynczą precyzję (typ `float`), trzeba do stałej dodać literę `F` lub `f`. Można też jawnie określić typ double, dodając na końcu literę `D` lub `d`. Domyślna podwójna precyzja zajmuje 64 bity pamięci, natomiast mniej dokładna pojedyncza precyzja wymaga tylko 32 bitów.

Literały logiczne

Literały logiczne są proste, ponieważ istnieją tylko dwie wartości logiczne: `true` i `false`. Wartości te nie konwertują się na żadną reprezentację numeryczną. Innymi słowy, w Javie literał `true` nie jest równy 1, a literał `false` nie jest równy 0. Literały mogą być przypisane tylko do zmiennych typu `boolean` lub zostać użyte w operatorach logicznych.

Literały znakowe

Znaki w Javie są niejako indeksami ze zbioru znaków Unicode. Są 16-bitowymi wartościami, które można konwertować do typu całkowitego lub wykonywać na nich działania arytmetyczne takie jak dodawanie i odejmowanie. Literał znakowy zawsze znajduje się wewnątrz apostrofów. Wszystkie znaki ASCII można wpisać bezpośrednio w cudzysłowach, na przykład `'a'`, `'z'` lub `'@'`. Dla znaków, których nie można wpisać bezpośrednio, istnieją specjalne sekwencje sterujące. Na przykład wpisanie znaku apostrofu wymaga użycia konstrukcji `'\''`, a wpisanie znaku nowego wiersza konstrukcji `'\n'`. Istnieje także mechanizm bezpośredniego wpisania znaku jako wartości ósemkowej lub szesnastkowej. Dla notacji ósemkowej trzeba najpierw wpisać lewy ukośnik, a następnie

podać trzycyfrową liczbę, na przykład `'\141'` dla litery `'a'`. Dla notacji szesnastkowej najpierw trzeba wpisać konstrukcję `\u`, a następnie podać cztery cyfry szesnastkowe, na przykład `'\u0061'` oznacza literę `'a'`, natomiast `'\ua432'` to jeden ze znaków japońskiego języka Katakana. Tabela 3.3 przedstawia dostępne sekwencje sterujące.

Tabela 3.3. *Sekwencje sterujące dla znaków*

Sekwencja sterująca	Opis
`\dddd`	Znak jako liczba ósemkowa (ddd)
`\uxxxx`	Znak Unicode jako liczba szesnastkowa (xxxx)
`\'`	Apostrof
`\"`	Cudzysłów
`\\`	Lewy ukośnik
`\r`	Powrót karetki
`\n`	Nowy wiersz (nazywany również przesunięciem papieru)
`\f`	Wysunięcie kartki
`\t`	Znak tabulacji
`\b`	Cofnięcie

Literały tekstowe

Ciągi znaków w Javie określa się tak samo, jak w większości innych języków — umieszczając tekst między cudzysłowami. Oto kilka przykładów ciągów znaków.

```
"Witaj świecie"
"dwa\nwiersze"
"\"Tekst w cudzysłowach.\""
```

W literałach tekstowych działają dokładnie te same sekwencje sterujące co w przypadku znaków. Java wymaga, aby ciąg znaków zaczynał się i kończył w tym samym wierszu. Nie istnieje coś takiego jak znak sterujący kontynuacji wiersza, jak w niektórych innych językach programowania.

W niektórych językach, na przykład C lub C++, ciągi znaków są zaimplementowane jako tablice znaków. W Javie jest inaczej. Ciągi znaków (typ `String`) są tak naprawdę obiektami. Ponieważ w Javie zaimplementowano teksty jako obiekty, ich wykorzystanie jest zarówno bardzo proste, jak i wyjątkowo rozbudowane.

Zmienne

Zmienna to podstawowa jednostka przechowywania informacji w programie Javy. Zmienną deklaruje się przez podanie nazwy identyfikatora, typu i opcjonalnej inicjalizacji. Poza tym wszystkie zmienne mają określony zasięg, który określa ich widoczność i czas życia. Tymi aspektami zajmę się nieco później.

Deklaracja zmiennej

W Javie wszystkie zmienne trzeba zadeklarować, zanim się z nich skorzysta. Podstawowa postać deklaracji zmiennej wygląda następująco.

```
typ identyfikator [ = wartość][, identyfikator [ = wartość] ...];
```

Element *typ* to jeden z typów prostych, nazwa klasy lub interfejsu. (Klasy i interfejsy zostaną dokładniej omówione w kolejnych rozdziałach tej części książki). Element *identyfikator* to nazwa zmiennej. Inicjalizacji zmiennej dokonuje się, używając znaku równości i podając *wartość*. Pamiętaj, że wyrażenie inicjalizacji musi powodować powstanie wartości takiego samego (lub zgodnego) typu jak tworzona zmienna. Aby zadeklarować kilka zmiennych, używa się listy oddzielanej przecinkami.

Oto kilka przykładów deklaracji zmiennych różnych typów. Zauważ, że niektóre są inicjalizowane.

```
int a, b, c;          // deklaruje trzy zmienne całkowite a, b i c
int d = 3, e, f = 5;  // deklaruje trzy zmienne całkowite, inicjalizuje d i f
byte z = 22;          // deklaruje i inicjalizuje z
double pi = 3.14159;  // deklaruje aproksymację liczby pi
char x = 'x';         // zmienna x ma wartość 'x'
```

Identyfikatory w swoich nazwach nie zawierają żadnych informacji na temat tego, jakiego są typu. W Javie każda poprawna nazwa identyfikatora może zostać przypisana do dowolnego typu.

Inicjalizacja dynamiczna

Choć poprzednie przykłady używały stałych do inicjalizacji, Java umożliwia inicjalizację dynamiczną wyrażeniami poprawnymi w momencie deklaracji zmiennej.

Poniżej znajduje się prosty program, który oblicza długość przeciwprostokątnej trójkąta prostokątnego na podstawie długości dwóch przyprostokątnych.

```
// Przykład inicjalizacji dynamicznej.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // c jest inicjalizowane dynamicznie
      double c = Math.sqrt(a * a + b * b);

      System.out.println("Przeciwprostokątna ma wartość " + c);
    }
}
```

W programie pojawia się deklaracja trzech zmiennych lokalnych — `a`, `b` i `c`. Dwie pierwsze są inicjalizowane stałymi, ale zmienna c jest inicjalizowana dynamicznie na długość przeciwprostokątnej (przy użyciu twierdzenia Pitagorasa). Poza tym program wykorzystuje wbudowaną metodę języka Java o nazwie `sqrt()`, będącą składową klasy

`Math`. Metoda zwraca pierwiastek kwadratowy z przekazanego argumentu. Dzięki przykładowemu programowi łatwo stwierdzić, że inicjalizacja dynamiczna może wykorzystać dowolne elementy poprawne w momencie wykonywania inicjalizacji, włączając w to wywołania metod, wykorzystanie innych zmiennych lub stałych.

Zasięg i czas życia zmiennych

Do tej pory wszystkie zmienne były deklarowane na początku metody `main()`. Tak naprawdę Java dopuszcza deklarowanie zmiennych w dowolnym bloku. W rozdziale 2. zdefiniowałem blok jako fragment kodu zaczynający się od otwierającego nawiasu klamrowego, a kończący na zamykającym nawiasie klamrowym. Blok definiuje **zasięg**, czyli rozpoczęcie nowego bloku to utworzenie nowego zasięgu. Zasięg określa, które obiekty są widziane przez inne części programu. Dodatkowo określa czas życia obiektów.

Wiele innych języków programowania definiuje dwa rodzaje zasięgów: lokalny i globalny. Niestety, te tradycyjne zasięgi nie pasują najlepiej do ścisłego, obiektowego modelu stosowanego przez Javę. Choć możliwe jest wykonanie czegoś na kształt zasięgu globalnego, to jest to raczej wyjątek aniżeli reguła. W Javie występuje inny podział zasięgów: na te zdefiniowane przez klasę i te zdefiniowane przez metodę. W zasadzie podział ten jest nieco sztuczny. Ponieważ jednak zasięg klasy posiada pewne unikatowe właściwości i atrybuty, które nie mają zastosowania w zasięgu metody, takie rozróżnienie ma sens. Z powodu tych różnic omówienie zasięgu klasy (i zadeklarowanych w nim zmiennych) odkładam aż do rozdziału 6., w którym dokładniej zostaną opisane klasy. Na razie skupię się na zasięgu związanym z metodami.

Zasięg definiowany przez metodę rozpoczyna się od otwierającego nawiasu klamrowego. Jeśli metoda posiada parametry, należą one do zasięgu metody. Ponieważ dokładne omówienie parametrów znajduje się dopiero w rozdziale 5., tutaj wspomnę tylko, że parametry działają dokładnie tak samo jak inne zmienne metody.

Ogólnie zmienne zadeklarowane wewnątrz zasięgu nie są widziane (czyli dostępne) przez kod znajdujący się poza danym zasięgiem. Z tego powodu zadeklarowanie zmiennej wewnątrz zasięgu powoduje przypisanie jej do konkretnego miejsca, a także chroni ją przez niepowołanym dostępem i modyfikacją. Właściwie zasady związane z zasięgiem są podstawą hermetyzacji.

Zasięg można zagnieżdżać. Na przykład za każdym razem, gdy tworzy się blok kodu, powstaje nowy, zagnieżdżony zasięg. W takiej sytuacji zasięg zewnętrzny zawiera zasięg wewnętrzny. Innymi słowy, zmienne zadeklarowane w zasięgu zewnętrznym będą widoczne w zasięgu wewnętrznym. Sytuacja odwrotna nie jest prawdziwa — zmienne zadeklarowane wewnątrz zasięgu wewnętrznego nie są widoczne na zewnątrz niego.

Aby lepiej zrozumieć efekt zagnieżdżania zasięgów, rozważmy następujący program.

```
// Przykład zasięgu bloku.
class Scope {
  public static void main(String args[]) {
    int x; // widziany przez cały kod metody
```

```
    x = 10;
    if(x == 10) { // początek nowego zasięgu
      int y = 20; // o tej zmiennej wie tylko ten blok

      // tutaj znana jest zarówno zmienna x, jak i y
      System.out.println("x i y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Błąd! Zmienna y nie jest znana.

    // nadal znamy wartość zmiennej x
    System.out.println("x wynosi " + x);
  }
}
```

Zgodnie z komentarzami, zmienna `x` jest deklarowana na początku metody `main()` i jest widziana przez cały jej kod. Wewnątrz bloku `if` dochodzi do deklaracji zmiennej `y`. Ponieważ blok określa zasięg, zmienna jest widoczna tylko wewnątrz bloku. Właśnie z tego powodu poza blokiem wiersz `y = 100;` został wyłączony za pomocą komentarza. Jeśli zostałby włączony, spowodowałby zgłoszenie błędu w trakcie kompilacji, ponieważ `y` nie jest widoczne poza swoim blokiem. Wewnątrz bloku `if` można korzystać ze zmiennej `x`, gdyż została ona zdefiniowana w bloku zewnętrznym, do którego ma dostęp blok wewnętrzny.

Wewnątrz bloku deklaracja zmiennej może pojawić się w dowolnym wierszu, ale jest poprawna dopiero po nim. Z tego powodu zadeklarowanie zmiennej na początku metody powoduje, że jest ona dostępna dla całego kodu tej metody. Deklaracja zmiennej na końcu bloku nie ma sensu, ponieważ żaden fragment kodu nie będzie miał do niej dostępu. Poniższy fragment kodu nie jest poprawny, gdyż zmienna `count` nie może zostać użyta przed jej zadeklarowaniem.

```
// Ten fragment zawiera błąd!
count = 100; // Ojej! Nie można użyć zmiennej count zanim nie zostanie zadeklarowana!
int count;
```

Warto pamiętać o innej bardzo istotnej kwestii: zmienne są tworzone w momencie wejścia w ich zasięg i niszczone przy wychodzeniu z danego zasięgu. Oznacza to, że zmienna nie będzie przechowywała swojej wartości, gdy znajdzie się poza zasięgiem. Właśnie z tego powodu zmienne metody nie zachowują swoich wartości między kolejnymi wywołaniami metody. Podobna sytuacja dotyczy bloków — zmienna utraci swoją wartość, gdy wyjdziemy z bloku. Innymi słowy, czas życia zmiennej jest ściśle związany z jej zasięgiem.

Jeśli deklaracja zmiennej zawiera inicjalizację, zmienna ta zostanie ponownie zainicjalizowana przy każdym wejściu do bloku, do którego jest przypisana. Rozważmy następujący program.

```
// Przykład czasu życia zmiennej.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y jest inicjalizowane przy każdej iteracji pętli
```

```
        System.out.println("y wynosi " + y); // zawsze zostanie wyświetlone –1
        y = 100;
        System.out.println("teraz y wynosi " + y);
      }
    }
}
```

Wykonanie programu spowoduje wyświetlenie następującego wyniku.

```
y wynosi -1
teraz y wynosi 100
y wynosi -1
teraz y wynosi 100
y wynosi -1
teraz y wynosi 100
```

Jak łatwo zauważyć, przy każdej nowej iteracji pętli `for` zmienna `y` jest ponownie inicjalizowana wartością –1. Choć później zostaje jej przypisana wartość 100, jest ona tracona w kolejnej iteracji.

Ostatnia uwaga: choć bloki mogą być zagnieżdżane, nie można zadeklarować zmiennej o takiej samej nazwie jak zmienna z bloku zewnętrznego. Poniższego programu nie uda się skompilować.

```
// Ten program się nie skompiluje.
class ScopeErr {
   public static void main(String args[]) {
     int bar = 1;
     {               // tworzy nowy zasięg
       int bar = 2; // Błąd kompilacji -- zmienna bar jest już zadeklarowana!
     }
   }
}
```

Konwersja typów i rzutowanie

Jeśli ktoś wcześniej pisał programy komputerowe, zapewne wie, że często zachodzi potrzeba przypisania wartości jednego typu do zmiennej innego typu. Jeśli oba typy są ze sobą zgodne, Java dokona automatycznej konwersji. Na przykład zawsze można przypisać wartość typu `int` do zmiennej `long`. Niestety, nie wszystkie typy są ze sobą zgodne i z tego powodu niejawna konwersja nie zawsze jest dozwolona. Na przykład Java nie wykona automatycznej konwersji z typu `double` do typu `byte`. Na szczęście nadal można dokonać takiej konwersji, ale trzeba to zrobić jawnie, używając tak zwanego **rzutowania typów**. Przyjrzyjmy się teraz konwersji automatycznej oraz rzutowaniu.

Automatyczna konwersja typów

Gdy jeden typ danych jest przypisywany do zmiennej innego typu, **automatyczna konwersja typu** zostanie wykonana, jeśli zostaną spełnione oba poniższe warunki:

- oba typy danych są zgodne,
- typ docelowy jest pojemniejszy (w sensie zakresu) od typu źródłowego.

Po spełnieniu obu warunków dochodzi do tak zwanej **konwersji rozszerzającej**. Na przykład typ `int` jest na tyle pojemny, że zawsze potrafi pomieścić wszystkie wartości typu `byte`, więc Java nie wymaga jawnego rzutowania.

W przypadku konwersji rozszerzającej, typy numeryczne (typy całkowite i zmiennoprzecinkowe) są ze sobą zgodne. Z drugiej strony, typy numeryczne nie są zgodne z typami `char` i `boolean`. Dodatkowo, typy `char` i `boolean` nie są zgodne między sobą.

Wspomniano już wcześniej, że Java dokonuje automatycznej konwersji literałów całkowitych do zmiennych typu `byte`, `short` oraz `long`.

Rzutowanie dla typów niezgodnych

Choć konwersja automatyczna jest pomocna, nie pokrywa wszelkich możliwych sytuacji. Na przykład zachodzi potrzeba konwersji z typu `int` do typu `byte`. Nie dojdzie w tej sytuacji do konwersji automatycznej, ponieważ typ `byte` jest mniejszy od typu `int`. Teki rodzaj konwersji jest często nazywany konwersją zawężającą, gdyż dokonuje jawnego ograniczenia wartości źródłowej do zakresu docelowego typu.

Aby dokonać konwersji między dwoma niezgodnymi typami, trzeba użyć rzutowania. **Rzutowanie** to po prostu jawna konwersja typu. Jego ogólna postać jest następująca.

```
(typ-docelowy) wartość
```

Element *typ-docelowy* określa typ, do którego ma zostać skonwertowana *wartość*. Poniższy fragment kodu dokonuje konwersji z typu `int` do typu `byte`. Jeśli wartość w zmiennej `a` jest większa od dopuszczalnego zakresu typu `byte`, zostanie wykonana operacja modulo (reszta z dzielenia liczby `int` przez zakres nowego typu) ograniczająca tę wartość do zakresu `byte`.

```
int a;
byte b;
// ...
b = (byte) a;
```

Inny rodzaj konwersji wystąpi, gdy liczba zmiennoprzecinkowa będzie konwertowana do typu całkowitego — wystąpi wtedy tak zwane **obcięcie**. Liczby całkowite nie mają części ułamkowej. Z tego powodu przy opisanej konwersji tracona jest informacja na temat ułamka. Jeśli na przykład przypiszemy wartość 1,23 do typu całkowitego, uzyskamy wartość 1. Ułamek 0,23 zostanie obcięty. Oczywiście, jeśli ogólna wartość będzie za duża, aby zmieścić się w docelowym typie, zostanie dodatkowo wykonana redukcja modulo dla zakresu docelowego typu.

Poniższy program przedstawia kilka konwersji wymagających rzutowania.

```
// Przykłady rzutowania.
class Conversion {
```

```
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nKonwersja z int na byte.");
    b = (byte) i;
    System.out.println("i oraz b " + i + " " + b);

    System.out.println("\nKonwersja z double na int.");
    i = (int) d;
    System.out.println("d oraz i " + d + " " + i);

    System.out.println("\nKonwersja z double na byte.");
    b = (byte) d;
    System.out.println("d oraz b " + d + " " + b);
  }
}
```

Wynik działania programu jest następujący.

```
Konwersja z int na byte.
i oraz b 257 1
Konwersja z double na int.
d oraz i 323.142 323
Konwersja z double na byte.
d i b 323.142 67
```

Przyjrzyjmy się poszczególnym konwersjom. Gdy wartość 257 jest rzutowana do typu `byte`, wynikiem jest reszta z dzielenia 257 przez 256 (zakres typu `byte`), czyli wartość 1. Gdy zmienną d konwertujemy do typu `int`, tracimy część ułamkową. Gdy zmienną d konwertujemy do typu `byte`, tracimy część ułamkową **oraz** dodatkowo dochodzi do redukcji modulo 256, co powoduje uzyskanie wartości 67.

Automatyczne rozszerzanie typów w wyrażeniach

Poza przypisaniami, istnieje jeszcze inne miejsce, w którym może dojść do konwersji typów: w wyrażeniach. Rozważmy następującą sytuację. W wyrażeniu precyzja wymagana w obliczeniach pośrednich wykracza poza zakres operandów. Oto przykład takiej sytuacji.

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Wynik wykonania działania `a * b` z pewnością przekroczy dopuszczalny rozmiar operandów typu `byte`. Z tego powodu Java automatycznie rozszerza (promuje) w wyrażeniu każdy operand typu `byte` lub `short` do typu `int`. Oznacza to, że obliczenie działania `a * b` jest wykonywane dla typu `int`, zamiast dla typu `byte`. Wynik wyrażenia pośredniego, 2000, jest więc poprawny, choć zmienne `a` i `b` są typu `byte`.

Choć automatyczne rozszerzanie typów jest pomocne, czasem powoduje tajemnicze błędy kompilacji. Na przykład poniższy kod, który wydaje się poprawny, nie daje się skompilować.

```
byte b = 50;
b = b * 2; // Błąd! Niemożliwe przypisanie wartości typu int do byte!
```

Kod próbuje przypisać wartość wyniku mnożenia 50*2, czyli całkowicie poprawną liczbę typu byte, z powrotem do zmiennej typu byte. Ponieważ jednak operandy zostały automatycznie rozszerzone do typu int, wynik wyrażenia także jest typu int. Zgodnie z wcześniejszymi zasadami, nie można automatycznie przypisać typu int do zmiennej typu byte, nawet jeśli wynik całej operacji mieści się w zakresie docelowego typu.

W przypadku, gdy w pełni rozumie się konsekwencje przepełnienia, można wykorzystać jawne rzutowanie.

```
byte b = 50;
b = (byte)(b * 2);
```

Spowoduje to poprawne przypisanie wartości 100 do zmiennej b.

Zasady rozszerzania typu

Poza wspomnianym wcześniej przypadkiem rozszerzania typów byte i short do typu int, Java definiuje kilka **zasad rozszerzania typów** stosowanych w wyrażeniach. Oto te zasady. Po pierwsze, wszystkie wartości typu byte lub short są rozszerzane do typu int (wcześniejszy przykład). Po drugie, jeśli którykolwiek z operandów jest typu long, całe wyrażenie jest rozszerzane do tego typu. Po trzecie, jeśli którykolwiek z operandów jest typu float, całe wyrażenie jest rozszerzane do tego typu. Po czwarte, jeśli którykolwiek z operandów jest typu double, wynikiem wykonania wyrażenia jest typ double.

Poniższy program obrazuje to, w jaki sposób każda z wartości wyrażenia jest rozszerzana w celu dopasowania się do zakresu pozostałych argumentów.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("wynik = " + result);
  }
}
```

Przyjrzyjmy się bliżej rozszerzaniu typów występującym w poniższym wierszu programu.

```
double result = (f * b) + (i / c) - (d * s);
```

W pierwszym podwyrażeniu, `f * b`, `b` zostaje rozszerzone do typu `float`, więc całe podwyrażenie jest typu `float`. W następnym podwyrażeniu, `i / c`, `c` zostaje rozszerzone do typu `int`, więc całe podwyrażenie jest typu `int`. W kolejnym podwyrażeniu, `d * s`, `s` zostaje rozszerzone do typu `double`, więc całe podwyrażenie jest typu `double`. Następnie rozważane są trzy wartości pośrednie typów `float`, `int` oraz `double`. Wynikiem dodania typu `float` do typu `int` jest `float`. Na końcu wynikiem odejmowania typu `double` od typu `float` jest rozszerzenie do typu `double`, co w efekcie powoduje zwrócenie całego wyrażenia typu `double`.

Tablice

Tablica to zbiór zmiennych tego samego typu, do których odwołujemy się przy użyciu wspólnej nazwy. Można tworzyć tablice dowolnego typu o jednym lub wielu wymiarach. Konkretny element tablicy jest dostępny poprzez swój indeks. Tablice na ogół służą do grupowania powiązanych ze sobą informacji.

Jeżeli dobrze zna się język C lub C++ należy uważać, ponieważ w Javie tablice działają inaczej niż we wcześniej wymienionych językach.

Tablice jednowymiarowe

Tablica jednowymiarowa to po prostu lista zmiennych tego samego typu. Aby utworzyć tablicę, trzeba najpierw zadeklarować zmienną tablicową odpowiedniego typu. Ogólna postać deklaracji tablicy jednowymiarowej jest następująca.

```
typ nazwa-zmiennej[];
```

Element *typ* określa typ bazowy tablicy, czyli typ poszczególnych elementów przechowywanych w tablicy. Innymi słowy, typ bazowy określa, jakiego rodzaju dane będą mogły być przechowywane w tablicy. Poniższa deklaracja tablicy o nazwie `month_days` będzie przechowywać liczby całkowite typu `int`.

```
int month_days[];
```

Choć deklaracja określa, że nazwa `month_days` oznacza zmienną tablicową, tak naprawdę nie powstała jeszcze żadna tablica. Java ustawi wartość zmiennej `month_days` na wartość `null`, co oznacza, że tablica niczego nie przechowuje. Aby połączyć `month_days` z rzeczywistą tablicą liczb całkowitych, trzeba ją najpierw zaalokować za pomocą operatora `new`. Operator ten powoduje alokację odpowiedniej ilości pamięci.

Operator `new` zostanie dokładniej omówiony w kolejnych rozdziałach. Na razie wystarczy wiedzieć jedynie, iż alokuje on pamięć dla tablic. Poniżej znajduje się ogólna postać wykorzystania operatora do utworzenia tablicy jednowymiarowej.

```
nazwa-zmiennej = new typ[rozmiar];
```

Element *typ* oznacza typ danych, dla których powstaje tablica. Element *rozmiar* określa liczbę elementów tablicy, a element *nazwa-zmiennej* to nazwa zadeklarowanej wcześniej zmiennej tablicowej. Wynika z tego, że do alokacji tablicy potrzebna jest informacja na temat typu i liczby elementów, które mają się w niej znaleźć. Elementy tablicy alokowane operatorem `new` są automatycznie zerowane. Poniższy kod alokuje 12-elementową tablicę liczb całkowitych i przypisuje ją do zmiennej `month_days`.

```
month_days = new int[12];
```

Po wykonaniu tej instrukcji zmienna `month_days` odnosi się do tablicy 12 liczb całkowitych. Co więcej, wszystkie elementy tablicy przyjęły wartość 0.

Podsumujmy: utworzenie tablicy jest procesem dwuetapowym. Najpierw deklaruje się zmienną odpowiedniego typu tablicowego. Następnie alokuje się za pomocą operatora `new` pamięć, która będzie przechowywała elementy tablicy, po czym przypisuje się ją do zmiennej tablicowej. Wynika z tego, że w Javie wszystkie tablice są alokowane dynamicznie. Jeśli koncepcja alokacji dynamicznej nic Czytelnikowi nie mówi, nie należy się przejmować, ponieważ zostanie ona opisana dokładniej w dalszej części książki.

Po zaalokowaniu tablicy do jej poszczególnych elementów odwołujemy się, podając indeks elementu zawarty w nawiasach kwadratowych. Indeksy wszystkich tablic rozpoczynają się od 0. Poniższy kod przypisuje wartość 28 drugiemu elementowi tablicy `month_days`.

```
month_days[1] = 28;
```

Poniższy kod spowoduje wyświetlenie wartości przechowywanej w elemencie o indeksie 3.

```
System.out.println(month_days[3]);
```

Po złożeniu wszystkich części otrzymujemy program, który tworzy tablicę liczby dni w każdym z miesięcy.

```
// Przykład tablicy jednowymiarowej.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("Kwiecień ma " + month_days[3] + " dni.");
  }
}
```

Uruchomienie programu spowoduje wyświetlenie liczby dni w miesiącu kwietniu. Jak wspomniano, indeksy tablic w Javie zaczynają się od zera, więc liczbę dni dla kwietnia odczytujemy jako `month_days[3]`.

Można połączyć deklarację zmiennej tablicowej z alokacją pamięci. Oto przykład.

```
int month_days[] = new int[12];
```

Na ogół w profesjonalnie napisanych programach stosuje się takie właśnie rozwiązanie.

Tablice można inicjalizować w momencie deklaracji. Sposób wykonania tego zadania nie różni się znacząco od inicjalizacji typów prostych. **Inicjalizacja tablicy** polega na podaniu listy wyrażeń oddzielonych przecinkami, zawartej wewnątrz nawiasów klamrowych. Przecinki rozdzielają wartości poszczególnych elementów tablicy. Zostanie automatycznie utworzona tablica na tyle duża, aby pomieściła wszystkie przekazane elementy. Nie trzeba w takiej sytuacji stosować operatora `new`. Poniżej znajduje się zmodyfikowana wersja poprzedniego przykładu. Tym razem jednak do określenia liczby dni w miesiącach stosuje inicjalizację tablicy.

```
// Ulepszona wersja poprzedniego programu.
class AutoArray {
  public static void main(String args[]) {
    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    System.out.println("Kwiecień ma " + month_days[3] + " dni.");
  }
}
```

Po uruchomieniu programu pojawi się dokładnie taki sam komunikat jak dla poprzedniego przykładu.

Java zawsze sprawdza, czy nie próbuje się zapisać lub odczytać wartości spoza zakresu tablicy. Innymi słowy, system wykonawczy Javy sprawdza, czy wszystkie stosowane indeksy znajdują się w poprawnym zakresie. Na przykład przy każdym odwołaniu do elementu tablicy `month_days` maszyna wirtualna sprawdzi, czy indeks to wartość z zakresu od 0 do 11. Gdy zostanie przekazana wartość spoza zakresu (liczba ujemna lub powyżej długości tablicy), program zgłosi błąd wykonania.

Kolejny program w nieco bardziej zaawansowany sposób korzysta z tablicy jednowymiarowej, ponieważ oblicza średnią z kilku wartości.

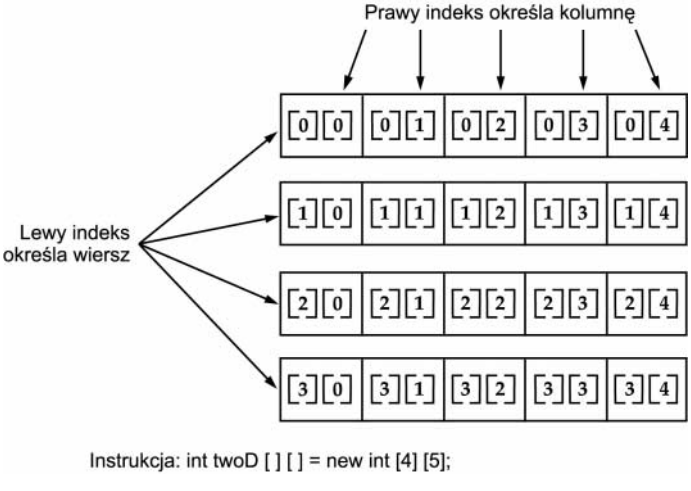
```
// Średnia wartości znajdujących się w tablicy.
class Average {
  public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;

    for(i=0; i<5; i++)
      result = result + nums[i];

    System.out.println("Średnia wynosi " + result / 5);
  }
}
```

Tablice wielowymiarowe

W Javie **tablice wielowymiarowe** są tak naprawdę tablicami tablic. Działają one dokładnie tak samo jak rzeczywiste tablice wielowymiarowe, choć występuje kilka subtelnych różnic. W celu zadeklarowania tablicy wielowymiarowej należy dodać dodatkowe indeksy, używając kolejnych par nawiasów kwadratowych. Poniższy przykład deklaruje tablicę dwuwymiarową o nazwie `twoD`.

```
int twoD[][] = new int[4][5];
```

Tablica ma rozmiar 4 na 5. Wewnętrznie macierz jest implementowana jako **tablica tablic** typu `int`. Koncepcyjnie wygląda ona mniej więcej tak, jak na rysunku 3.1.



Rysunek 3.1. *Widok koncepcyjny tablicy dwuwymiarowej o wymiarach 4 na 5*

Poniższy program numeruje poszczególne elementy tablicy, posuwając się od lewej do prawej i z góry na dół. Następnie wyświetla te wartości.

```
// Przykład tablicy dwuwymiarowej.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

Wykonanie programu powoduje uzyskanie następującego wyjścia.

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Gdy alokuje się pamięć dla tablicy wielowymiarowej, obowiązkowe jest podanie tylko pierwszego (najbardziej lewego) wymiaru. Pozostałe wymiary można deklarować osobno. Poniższy kod tworzy dokładnie tę samą dwuwymiarową tablicę co wcześniej, ale drugi wymiar jest ustalany ręcznie.

```
int twoD[][]= new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

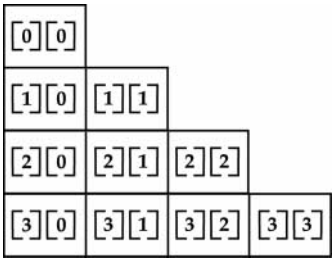
Choć w tej sytuacji sposób ręcznej alokacji tablicy nie ma żadnej przewagi nad alokacją automatyczną, są sytuacje, w których warto go stosować. Przykładem może być sytuacja, w której poszczególne wymiary mają posiadać różną liczbę elementów. Ponieważ tablica wielowymiarowa jest tak naprawdę tablicą tablic, mamy pełną swobodę w dobieraniu rozmiarów podtablic. Poniższy program obrazuje, w jaki sposób wykonać dwuwymiarową tablicę, w której liczba elementów w drugim wymiarze zmienia się.

```
// Ręczna alokacja różnych rozmiarów dla drugiego wymiaru.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

Wykonanie programu powoduje uzyskanie następującego wyjścia.

```
0
1 2
3 4 5
6 7 8 9
```

Rysunek 3.2 przedstawia tablicę utworzoną przez program.



Rysunek 3.2. *Dwuwymiarowa tablica z różnym rozmiarem drugiego wymiaru*

Wykorzystanie nieregularnych tablic wielowymiarowych w wielu zastosowaniach nie jest pożądane, ponieważ ludzie na ogół oczekują symetryczności takiej tablicy. Z drugiej strony są sytuacje, w których nieregularność zwiększa wydajność. Przykładem może być algorytm, który potrzebuje bardzo dużej tablicy dwuwymiarowej, ale zapełnia tylko niewielki jej fragment.

Tablice wielowymiarowe także można inicjalizować. W tym celu inicjalizację każdego z wymiarów trzeba zawrzeć wewnątrz własnego zestawu nawiasów klamrowych. Poniższy przykładowy program tworzy macierz, w której każdy element zawiera wartość mnożenia indeksu wiersza i kolumny. Zauważ, że przy inicjalizacji tablicy można stosować nie tylko literały, ale również wyrażenia.

```
// Inicjalizacja tablicy wielowymiarowej.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

Wyniki działania programu są następujące.

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Jak łatwo zauważyć, każdy element został zainicjalizowany zgodnie z listą inicjalizacji.

Przyjrzyjmy się bardziej złożonemu przykładowi, który wykorzystuje tablicę wielowymiarową. Kolejny program tworzy tablicę trójwymiarową 3 na 4 na 5. Następnie zapełnia elementy wartościami mnożenia ich indeksów. Na końcu wyświetla obliczone wartości.

```
// Przykład tablicy trójwymiarowej.
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
        threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

Wyniki działania programu są następujące.

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternatywna składnia deklaracji tablicy

Java obsługuje alternatywną postać deklaracji tablicy.

```
typ[] nazwa-zmiennej;
```

W tej postaci nawiasy kwadratowe pojawiają się po nazwie typu zamiast po nazwie zmiennej. Dwie poniższe deklaracje są sobie równoważne.

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

Poniższe deklaracje również są sobie równoważne.

```
char twod1[][] = new char[3][4];
char[][] twod1 = new char[3][4];
```

Alternatywna postać deklaracji przydaje się przede wszystkim wtedy, gdy w jednym wierszu deklaruje się wiele tablic. Oto przykład.

```
int[] nums, nums2, nums3; // powstają trzy tablice
```

W ten sposób powstają trzy tablice typu `int`. Gdyby zastosować podstawową postać deklaracji, trzeba by napisać następujący wiersz.

```
int nums[], nums2[], nums3[]; // powstają trzy tablice
```

Dodatkowo, postać alternatywna przydaje się także wtedy, gdy tablica ma być typem zwracanym przez metodę. W książce obie składnie pojawiają się zamiennie.

Kilka słów na temat ciągów znaków

W poprzednich opisach związanych z typami danych oraz tablicami praktycznie nie pojawiały się żadne informacje na temat ciągów znaków i związanych z nimi typów. Nie oznacza to, że Java nie obsługuje ciągów znaków — wręcz przeciwnie. Istnieje w Javie typ tekstowy o nazwie `String`, ale nie jest on typem prostym ani tablicą znaków. Typ `String` to klasa, więc jej pełny opis wymaga dobrej znajomości elementów obiektowych dostępnych w Javie. Z tego powodu zostanie opisana dopiero po dokładnym omówieniu obiektowości. Ponieważ jednak proste teksty pojawiają się w wielu przykładowych programach, oto krótkie wyjaśnienie.

Typ `String` służy do deklarowania ciągów znaków. Oczywiście można również deklarować tablice ciągów znaków. Do zmiennej typu `String` można przypisać dowolny tekst ujęty w cudzysłowy lub też przypisać zawartość jednej zmiennej typu `String` do innej zmiennej tego samego typu. Metoda `println()` jako argumenty przyjmuje obiekty typu `String`. Rozważmy następujący fragment kodu.

```
String str = "to jest test";
System.out.println(str);
```

W tym przykładzie `str` to obiekt typu `String`, któremu został przypisany tekst „to jest test”. Tekst zostaje wyświetlony za pomocą instrukcji `println()`.

Obiekty `String` posiadają wiele cech i atrybutów, które czynią je bardzo elastycznymi i łatwymi w użyciu. Jednak w kilku kolejnych rozdziałach będziemy korzystać tylko z ich najprostszej postaci.

Uwaga dla programistów języka C lub C++ na temat wskaźników

Jeśli jest się doświadczonym programistą języka C lub C++, z pewnością wielokrotnie stosowało się w nim wskaźniki. W tym rozdziale do tej pory ani razu nie padło słowo wskaźnik. Powód jest bardzo prosty: Java po prostu nie obsługuje wskaźników.

(W zasadzie to nie obsługuje wskaźników, które mogłyby być modyfikowane lub odczytywane przez programistę). Java nie może sobie pozwolić na udostępnienie wskaźników, gdyż takie rozwiązanie złamałoby ścianę dzielącą środowisko wykonawcze Javy od systemu operacyjnego komputera. (Pamiętaj, że wskaźnik może przyjąć dowolny adres — nawet taki, który znajduje się poza systemem wykonawczym Javy). Ponieważ języki C i C++ intensywnie korzystają ze wskaźników, może się wydawać, iż ich brak w języku Java jest znaczącym ograniczeniem. Na szczęście nie jest to prawda. Java została tak zaprojektowana, że wskaźniki w ogóle nie są potrzebne, jeśli pozostaje się wewnątrz środowiska wykonawczego.