

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

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

O Autorze	21
Przedmowa	23
Część I Język Java	27
Rozdział 1. Historia i ewolucja języka Java	29
Rodowód Javy	29
Narodziny nowoczesnego języka — C	30
Język C++ — następny krok	31
Podwaliny języka Java	32
Powstanie języka Java	32
Powiązanie z językiem C#	34
Dlaczego język Java jest tak ważny dla internetu	35
Aplety Javy	35
Bezpieczeństwo	36
Przenośność	36
Magia języka Java — kod bajtowy	36
Hasła języka Java	37
Prostota	38
Obiektowość	38
Solidność	38
Wielowątkowość	39
Neutralność architektury	39
Interpretowalność i wysoka wydajność	40
Rozproszenie	40
Dynamika	40
Ewolucja Javy	40
Rewolucja J2SE 5	41
Kultura innowacji	42
Rozdział 2. Podstawy języka Java	43
Programowanie obiektowe	43
Dwa paradygmaty	43
Abstrakcja	44
Trzy zasady programowania obiektowego	44
Pierwszy przykładowy program	49
Wpisanie kodu programu	50
Kompilacja programów	50
Bliższe spojrzenie na pierwszy przykładowy program	51

Drugi prosty program	53
Dwie instrukcje sterujące	55
Instrukcja if	55
Pętla for	57
Bloki kodu	58
Kwestie leksykalne	59
Znaki niedrukowane	60
Identyfikatory	60
Literał	60
Komentarze	60
Separatory	61
Słowa kluczowe języka Java	61
Biblioteki klas Javy	62
Rozdział 3. Typy danych, zmienne i tablice	63
Java to język ze ścisłą kontrolą typów	63
Typy proste	63
Typy całkowite	64
Typ byte	65
Typ short	65
Typ int	66
Typ long	66
Typy zmiennoprzecinkowe	67
Typ float	67
Typ double	67
Typ znakowy	68
Typ logiczny	69
Blizsze spojrzenie na literały	70
Literały będące liczbami całkowitymi	70
Literały zmiennoprzecinkowe	71
Literały logiczne	71
Literały znakowe	71
Literały tekstowe	72
Zmienne	72
Deklaracja zmiennej	73
Inicjalizacja dynamiczna	73
Zasięg i czas życia zmiennych	74
Konwersja typów i rzutowanie	76
Automatyczna konwersja typów	76
Rzutowanie dla typów niezgodnych	77
Automatyczne rozszerzanie typów w wyrażeniach	78
Zasady rozszerzania typu	79
Tablice	80
Tablice jednowymiarowe	80
Tablice wielowymiarowe	83
Alternatywna składnia deklaracji tablicy	86
Kilka słów na temat ciągów znaków	87
Uwaga dla programistów języka C lub C++ na temat wskaźników	87
Rozdział 4. Operatory	89
Operatory arytmetyczne	89
Podstawowe operatory arytmetyczne	89
Operator reszty z dzielenia	91
Operatory arytmetyczne z przypisaniem	91
Inkrementacja i dekrementacja	92

Operatory bitowe	94
Podstawowe operatory bitowe	95
Przesunięcie w lewo	97
Przesunięcie w prawo	99
Przesunięcie w prawo bez znaku	100
Operatory bitowe z przypisaniem	102
Operatory relacji	103
Operatory logiczne	104
Operatory logiczne ze skracaniem	105
Operator przypisania	106
Operator ?:	106
Kolejność wykonywania operatorów	107
Wykorzystanie nawiasów okrągłych	107
Rozdział 5. Struktury sterujące	109
Instrukcje wyboru	109
Konstrukcja if	109
Konstrukcja switch	112
Instrukcje iteracji	116
Pętla while	116
Pętla do-while	118
Pętla for	121
Wersja for-each pętli for	124
Pętle zagnieżdżone	129
Instrukcje skoku	130
Instrukcja break	130
Instrukcja continue	134
Instrukcja return	135
Rozdział 6. Wprowadzenie do klas	137
Klasy	137
Ogólna postać klasy	137
Prosta klasa	139
Tworzenie obiektów	141
Bliższe spojrzenie na klasę	142
Przypisywanie zmiennych referencyjnych do obiektów	143
Metody	144
Dodanie metody do klasy Box	145
Zwracanie wartości	146
Dodanie metody przyjmującej parametry	148
Konstruktor	150
Konstruktor sparametryzowany	152
Słowo kluczowe this	153
Ukrywanie zmiennych składowych	153
Mechanizm odzyskiwania pamięci	154
Metoda finalize()	154
Klasa stosu	155
Rozdział 7. Dokładniejsze omówienie metod i klas	159
Przeciążanie metod	159
Przeciążanie konstruktorów	162
Obiekty jako parametry	164
Dokładniejsze omówienie przekazywania argumentów	166
Zwracanie obiektów	168
Rekurencja	169

Wprowadzenie do sterowania dostępem	171
Składowe statyczne	175
Słowo kluczowe final	177
Powtórka z tablic	177
Klasy zagnieżdżone i klasy wewnętrzne	179
Omówienie klasy String	182
Wykorzystanie argumentów wiersza poleceń	184
Zmienna liczba argumentów	185
Przeciążanie metod o zmiennej liczbie argumentów	188
Zmienna liczba argumentów i niejednoznaczności	189
Rozdział 8. Dziedziczenie	191
Proste dziedziczenie	191
Dostęp do składowych a dziedziczenie	193
Bardziej praktyczny przykład	194
Zmienna klasy bazowej może zawierać referencję do obiektu podklasy	196
Słowo kluczowe super	197
Wykorzystanie słowa kluczowego super	
do wywołania konstruktora klasy bazowej	197
Drugie zastosowanie słowa kluczowego super	200
Tworzenie hierarchii wielopoziomowej	201
Kiedy dochodzi do wywołania konstruktorów?	204
Przesłanianie metod	205
Dynamiczne przydzielanie metod	208
Dlaczego warto przesłaniać metody?	209
Zastosowanie przesłaniania metod	210
Klasy abstrakcyjne	211
Słowo kluczowe final i dziedziczenie	214
Słowo kluczowe final zapobiega przesłanianiu	214
Słowo kluczowe final zapobiega dziedziczeniu	215
Klasa Object	215
Rozdział 9. Pakiety i interfejsy	217
Pakiety	217
Definiowanie pakietu	218
Znajdowanie pakietów i ścieżka CLASSPATH	219
Prosty przykład pakietu	219
Ochrona dostępu	220
Przykład dostępu	221
Import pakietów	224
Interfejsy	226
Definiowanie interfejsu	227
Implementacja interfejsu	227
Zastosowanie interfejsów	230
Zmienne w interfejsach	233
Interfejsy można rozszerzać	235
Rozdział 10. Obsługa wyjątków	237
Podstawy obsługi wyjątków	237
Typy wyjątków	238
Nieprzechwycone wyjątki	238
Wykorzystanie konstrukcji try i catch	240
Wyświetlenie opisu wyjątku	241
Wiele klauzul catch	241
Zagnieżdżone konstrukcje try	243

Instrukcja throw	245
Klauzula throws	246
Słowo kluczowe finally	248
Wyjątki wbudowane w język Java	249
Tworzenie własnej podklasy wyjątków	250
Łańcuch wyjątków	252
Wykorzystanie wyjątków	254
Rozdział 11. Programowanie wielowątkowe	255
Model wątków języka Java	256
Priorytety wątków	257
Synchronizacja	257
Przekazywanie komunikatów	258
Klasa Thread i interfejs Runnable	258
Wątek główny	259
Tworzenie wątku	261
Implementacja interfejsu Runnable	261
Rozszerzanie klasy Thread	263
Wybór odpowiedniego podejścia	264
Tworzenie wielu wątków	264
Wykorzystanie metod isAlive() oraz join()	266
Priorytety wątków	268
Synchronizacja	271
Synchronizacja metod	271
Konstrukcja synchronized	274
Komunikacja międzywątkowa	275
Blokada wzajemna	279
Zawieszanie, wznowianie i zatrzymywanie wątków	281
Zawieszanie, wznowianie i zatrzymywanie wątków w Java 1.1 lub starszej	282
Nowoczesny sposób zawieszania, wznowiania i zatrzymywania wątków	284
Korzystanie z wielowątkowości	286
Rozdział 12. Wyliczenia, automatyczne otaczanie typów prostych i metadane	287
Wyliczenia	287
Podstawy wyliczeń	288
Metody values() i valueOf()	290
Wyliczenia Javy jako typ klasy	291
Wyliczenia dziedziczą po klasie Enum	293
Inny przykład wyliczenia	295
Typy otoczkowe	296
Automatyczne otaczanie typów prostych	298
Automatyczne otaczanie i metody	299
Automatyczne otaczanie i rozpakowywanie w wyrażeniach	300
Automatyczne otaczanie dla typów znakowych i logicznych	302
Automatyczne otaczanie pomaga zapobiegać błędom	303
Słowo ostrzeżenia	303
Metadane (notatki)	304
Podstawy tworzenia notatek	304
Określenie strategii zachowania	305
Pobieranie notatek w trakcie działania programu dzięki refleksji	306
Interfejs AnnotatedElement	310
Wartości domyślne	311
Notatki znacznikowe	312
Notatki jednoelementowe	313
Wbudowane notatki	315
Ograniczenia	316

Rozdział 13. Wejście-wyjście, applety i inne tematy	317
Podstawowa obsługa wejścia i wyjścia	317
Strumienie	318
Strumienie znakowe i bajtowe	318
Predefiniowane strumienie	319
Odczyt danych z konsoli	321
Odczyt znaków	321
Odczyt ciągów znaków	322
Wyświetlanie informacji na konsoli	324
Klasa <code>PrintWriter</code>	324
Odczyt i zapis plików	325
Podstawy appletów	328
Modyfikatory <code>transient</code> i <code>volatile</code>	331
Operator <code>instanceof</code>	332
Modyfikator <code>strictfp</code>	334
Metody napisane w kodzie rdzennym	335
Problemy z metodami rdzennymi	338
Asercja	338
Opcje włączania i wyłączania asercji	341
Import statyczny	341
Rozdział 14. Typy sparametryzowane	345
Czym są typy sparametryzowane?	346
Prosty przykład zastosowania typów sparametryzowanych	346
Typy sparametryzowane działają tylko dla obiektów	350
Typy sparametryzowane różnią się, jeśli mają inny argument typu	350
W jaki sposób typy sparametryzowane zwiększają bezpieczeństwo?	350
Klasa sparametryzowana z dwoma parametrami typu	353
Ogólna postać klasy sparametryzowanej	354
Typy ograniczone	354
Zastosowanie argumentów wieloznacznych	357
Ograniczony argument wieloznaczny	359
Tworzenie metody sparametryzowanej	364
Konstruktory sparametryzowane	366
Interfejsy sparametryzowane	367
Typy surowe i dotychczasowy kod	369
Hierarchia klas sparametryzowanych	372
Zastosowanie sparametryzowanej klasy bazowej	372
Podklasa sparametryzowana	374
Porównywanie typów w trakcie działania programu	
dla hierarchii klas sparametryzowanych	375
Rzutowanie	378
Przesłanie metod w klasach sparametryzowanych	378
Znoszenie	379
Metody mostu	381
Błędy niejednoznaczności	383
Pewne ograniczenia typów sparametryzowanych	384
Nie można tworzyć egzemplarza parametru typu	384
Ograniczenia dla składowych statycznych	385
Ograniczenia tablic typów sparametryzowanych	385
Ograniczenia wyjątków typów sparametryzowanych	386
Ostatnie uwagi na temat typów sparametryzowanych	387

Część II Biblioteka języka Java389**Rozdział 15. Ciągi znaków 391**

Konstruktory klasy String	392
Konstruktory dodane w wydaniu J2SE 5	394
Długość ciągu znaków	394
Specjalne operacje na ciągach znaków	394
Literały tekstowe	395
Konkatenacja ciągów znaków	395
Konkatenacja ciągu znaków z innymi typami danych	396
Konwersja do ciągu znaków i metoda toString()	396
Wydobycie znaków	397
Metoda charAt()	398
Metoda getChars()	398
Metoda getBytes()	399
Metoda toCharArray()	399
Porównywanie ciągów znaków	399
Metody equals() i equalsIgnoreCase()	399
Metoda regionMatches()	400
Metody startsWith() i endsWith()	401
Metoda equals() a operator ==	401
Metoda compareTo()	402
Wyszukiwanie podciągów znaków	403
Modyfikacja ciągu znaków	405
Metoda substring()	405
Metoda concat()	406
Metoda replace()	406
Metoda trim()	407
Konwersja danych za pomocą metody valueOf()	408
Zmiana wielkości liter ciągu znaków	408
Dodatkowe metody klasy String	409
Klasa StringBuffer	409
Konstruktory klasy StringBuffer	411
Metody length() i capacity()	411
Metoda ensureCapacity()	412
Metoda setLength()	412
Metody charAt() i setCharAt()	412
Metoda getChars()	413
Metoda append()	413
Metoda insert()	414
Metoda reverse()	414
Metody delete() i deleteCharAt()	415
Metoda replace()	416
Metoda substring()	416
Dodatkowe metody klasy StringBuffer	416
Klasa StringBuilder	418

Rozdział 16. Pakiet java.lang 419

Otoczki typów prostych	420
Klasa Number	420
Klasy Double i Float	420
Klasy Byte, Short, Integer i Long	424
Klasa Character	432
Dodatki wprowadzone w celu obsługi rozszerzonych znaków Unicode	434
Klasa Boolean	435

Klasa Void	435
Klasa Process	436
Klasa Runtime	437
Zarządzanie pamięcią	438
Wykonywanie innych programów	440
Klasa ProcessBuilder	441
Klasa System	442
Wykorzystanie metody currentTimeMillis()	
do obliczania czasu wykonywania programu	444
Użycie metody arraycopy()	445
Właściwości środowiska	446
Klasa Object	446
Wykorzystanie metody clone() i interfejsu Cloneable	446
Klasa Class	449
Klasa ClassLoader	452
Klasa Math	452
Funkcje trygonometryczne	452
Funkcje wykładnicze	453
Funkcje zaokrążeń	453
Różnorodne metody klasy Math	454
Klasa StrictMath	455
Klasa Compiler	455
Klasy Thread i ThreadGroup oraz interfejs Runnable	455
Interfejs Runnable	456
Klasa Thread	456
Klasa ThreadGroup	458
Klasy ThreadLocal i InheritableThreadLocal	462
Klasa Package	463
Klasa RuntimePermission	463
Klasa Throwable	463
Klasa SecurityManager	463
Klasa StackTraceElement	464
Klasa Enum	465
Interfejs CharSequence	466
Interfejs Comparable	467
Interfejs Appendable	467
Interfejs Iterable	467
Interfejs Readable	468
Podpakiety pakietu java.lang	468
Podpakiet java.lang.annotation	468
Podpakiet java.lang.instrument	469
Podpakiet java.lang.management	469
Podpakiet java.lang.ref	469
Podpakiet java.lang.reflect	469

Rozdział 17. Pakiet java.util, część 1. — kolekcje 471

Wprowadzenie do kolekcji	472
Zmiany w kolekcjach spowodowane wydaniem J2SE 5	473
Typy sparametryzowane w znaczący sposób zmieniają kolekcje	473
Automatyczne otaczanie ułatwia korzystanie z typów prostych	474
Pętla for typu for-each	474
Interfejsy kolekcji	474
Interfejs Collection	475
Interfejs List	477

Interfejs Set	478
Interfejs SortedSet	479
Interfejs Queue	479
Klasy kolekcji	480
Klasa ArrayList	481
Klasa LinkedList	484
Klasa HashSet	486
Klasa LinkedHashSet	488
Klasa TreeSet	488
Klasa PriorityQueue	489
Klasa EnumSet	490
Dostęp do kolekcji za pomocą iteratora	490
Korzystanie z iteratora Iterator	492
Pętla typu for-each jako alternatywa dla iteratora	494
Przechowywanie w kolekcjach własnych klas	495
Interfejs RandomAccess	496
Korzystanie z map	496
Interfejsy map	497
Klasy map	499
Komparatory	505
Wykorzystanie komparatora	505
Algorytmy kolekcji	508
Klasa Arrays	513
Dlaczego kolekcje są sparametryzowane?	516
Starsze klasy i interfejsy	519
Interfejs wyliczeń	520
Klasa Vector	520
Klasa Stack	524
Klasa Dictionary	526
Klasa Hashtable	527
Klasa Properties	530
Wykorzystanie metod store() i load()	533
Ostatnie uwagi na temat kolekcji	535
Rozdział 18. Pakiet java.util, część 2. — pozostałe klasy użytkowe	537
Klasa StringTokenizer	537
Klasa BitSet	539
Klasa Date	542
Klasa Calendar	543
Klasa GregorianCalendar	546
Klasa TimeZone	548
Klasa SimpleTimeZone	548
Klasa Locale	550
Klasa Random	551
Klasa Observable	553
Interfejs Observer	554
Przykład użycia interfejsu Observer	555
Klasy Timer i TimerTask	557
Klasa Currency	560
Klasa Formatter	561
Konstruktory klasy Formatter	561
Metody klasy Formatter	562
Podstawy formatowania	562
Formatowanie tekstów i znaków	565

Formatowanie liczb	565
Formatowanie daty i godziny	567
Specyfikatory %n i %%	567
Określanie minimalnej szerokości pola	569
Określanie precyzji	570
Używanie znaczników formatów	571
Wyrównywanie danych wyjściowych	572
Znaczniki spacji, plusa, zera i nawiasów	573
Znacznik przecinka	574
Znacznik #	574
Opcja wielkich liter	574
Stosowanie indeksu argumentu	575
Metoda printf() w Javie	576
Klasa Scanner	577
Konstruktory klasy Scanner	577
Podstawy skanowania	578
Kilka przykładów użycia klasy Scanner	580
Ustawianie separatorów	585
Pozostałe elementy klasy Scanner	586
Podpakiety pakietu java.util	587
java.util.concurrent, java.util.concurrent.atomic oraz java.util.concurrent.locks ...	588
java.util.jar	588
java.util.logging	588
java.util.prefs	588
java.util.regex	588
java.util.zip	588

Rozdział 19. Operacje wejścia-wyjścia: analiza pakietu java.io 589

Dostępne w Javie klasy i interfejsy obsługujące operacje wejścia-wyjścia	590
Klasa File	590
Katalogi	593
Stosowanie interfejsu FilenameFilter	594
Alternatywna metoda listFiles()	595
Tworzenie katalogów	596
Interfejsy Closeable i Flushable	596
Klasy strumienia	597
Strumienie bajtów	597
Klasa InputStream	598
Klasa OutputStream	598
Klasa FileInputStream	599
Klasa FileOutputStream	601
Klasa ByteArrayInputStream	602
Klasa ByteArrayOutputStream	603
Filtrowane strumienie bajtów	605
Buforowane strumienie bajtów	605
Klasa SequenceInputStream	609
Klasa PrintStream	610
Klasy DataOutputStream i DataInputStream	613
Klasa RandomAccessFile	615
Strumienie znaków	616
Klasa Reader	616
Klasa Writer	616
Klasa FileReader	616
Klasa FileWriter	618

Klasa CharArrayReader	619
Klasa CharArrayWriter	620
Klasa BufferedReader	621
Klasa BufferedWriter	623
Klasa PushbackReader	623
Klasa PrintWriter	624
Stosowanie operacji wejścia-wyjścia na strumieniach	626
Usprawnienie metody wc() przez zastosowanie klasy StreamTokenizer	627
Serializacja	629
Interfejs Serializable	630
Interfejs Externalizable	630
Interfejs ObjectOutputStream	631
Klasa ObjectOutputStream	631
Interfejs ObjectInput	633
Klasa ObjectInputStream	633
Przykład serializacji	634
Korzyści wynikające ze stosowania strumieni	636
Rozdział 20. Obsługa sieci	637
Podstawy obsługi sieci	637
Przegląd gniazd	638
Klient-serwer	638
Gniazda zastrzeżone	639
Serwery pośredniczące	639
Obsługa adresów internetowych	640
Java i usługi sieciowe	641
Klasy i interfejsy obsługujące komunikację siecią	641
Klasa InetAddress	642
Metody fabryczne	642
Metody klasy	643
Klasy InetAddress oraz Inet6Address	644
Gniazda klientów TCP/IP	644
Przykład użycia usługi whois	645
URL	646
Format	647
Klasa URLConnection	648
Gniazda serwerów TCP/IP	651
Serwer pośredniczący protokołu HTTP z pamięcią podręczną	651
Kod źródłowy	652
Datagramy	672
Klasa DatagramPacket	672
Przesyłanie datagramów pomiędzy serwerem a klientem	673
Klasa URI	674
Nowe klasy środowiska J2SE 5	675
Rozdział 21. Klasa Applet	677
Podstawy appletów	677
Klasa Applet	678
Architektura appletu	680
Szkielet appletu	681
Inicjalizacja i przerywanie działania appletu	682
Przysyłanie metody update()	684
Proste metody wyświetlania składników appletów	684
Żądanie ponownego wyświetlenia	686
Prosty applet z paskiem reklamowym	688

Wykorzystywanie paska stanu	690
Znacznik APPLET języka HTML	691
Przekazywanie parametrów do apletów	692
Udoskonalenie apletu z paskiem reklamowym	694
Metody getDocumentBase() i getCodeBase()	695
Interfejs AppletContext i metoda showDocument()	696
Interfejs AudioClip	698
Interfejs AppletStub	698
Wyświetlanie danych wyjściowych na konsoli	698
Rozdział 22. Obsługa zdarzeń	699
Dwa mechanizmy obsługi zdarzeń	699
Model obsługi zdarzeń oparty na ich delegowaniu	700
Zdarzenia	700
Źródła zdarzeń	701
Obiekty nasłuchujące zdarzeń	702
Klasy zdarzeń	702
Klasa ActionEvent	704
Klasa AdjustmentEvent	704
Klasa ComponentEvent	705
Klasa ContainerEvent	706
Klasa FocusEvent	706
Klasa InputEvent	707
Klasa ItemEvent	708
Klasa KeyEvent	709
Klasa MouseEvent	710
Klasa MouseWheelEvent	712
Klasa TextEvent	713
Klasa WindowEvent	713
Źródła zdarzeń	714
Interfejsy nasłuchujące zdarzeń	715
Interfejs ActionListener	715
Interfejs AdjustmentListener	715
Interfejs ComponentListener	716
Interfejs ContainerListener	717
Interfejs FocusListener	717
Interfejs ItemListener	717
Interfejs KeyListener	717
Interfejs MouseListener	717
Interfejs MouseMotionListener	718
Interfejs MouseWheelListener	718
Interfejs TextListener	718
Interfejs WindowFocusListener	718
Interfejs WindowListener	719
Stosowanie modelu delegowania zdarzeń	719
Obsługa zdarzeń generowanych przez mysz	720
Obsługa zdarzeń generowanych przez klawiaturę	722
Klasy adapterów	726
Klasy wewnętrzne	728
Anonimowa klasa wewnętrzna	729
Rozdział 23. Wprowadzenie do AWT: praca z oknami, grafiką i tekstem	731
Klasy AWT	732
Podstawy okien	734
Klasa Component	734
Klasa Container	735

Klasa Panel	735
Klasa Window	735
Klasa Frame	736
Klasa Canvas	736
Praca z oknami typu Frame	736
Ustawianie wymiarów okna	737
Ukrywanie i wyświetlanie okna	737
Ustawianie tytułu okna	737
Zamykanie okna typu Frame	737
Tworzenie okna typu Frame z poziomu apletu	738
Obsługa zdarzeń w oknie typu Frame	740
Tworzenie programu wykorzystującego okna	743
Wyświetlanie informacji w oknie	745
Praca z grafiką	746
Rysowanie prostych	746
Rysowanie prostokątów	747
Rysowanie elips, kół i okręgów	748
Rysowanie łuków	749
Rysowanie wielokątów	750
Dostosowywanie rozmiarów obiektów graficznych	751
Praca z klasą Color	752
Metody klasy Color	753
Ustawianie bieżącego koloru kontekstu graficznego	754
Aplet demonstrujący zastosowanie klasy Color	754
Ustawianie trybu rysowania	755
Praca z czcionkami	757
Określanie dostępnych czcionek	757
Tworzenie i wybieranie czcionek	759
Uzyskiwanie informacji o czcionkach	761
Zarządzanie tekstowymi danymi wyjściowymi	
z wykorzystaniem klasy FontMetrics	762
Wyświetlanie tekstu w wielu wierszach	763
Wyśrodkowanie tekstu	766
Wyrównywanie wielowierszowych danych tekstowych	767

Rozdział 24. Stosowanie kontrolek AWT, menadżerów układu graficznego

oraz menu	771
Podstawy kontrolek	772
Dodawanie i usuwanie kontrolek	772
Odpowiadanie na zdarzenia kontrolek	772
Etykiety	773
Stosowanie przycisków	774
Obsługa zdarzeń przycisków	775
Stosowanie pól wyboru	777
Obsługa zdarzeń pól wyboru	778
CheckboxGroup	780
Kontrolki list rozwijanych	782
Obsługa zdarzeń list rozwijanych	783
Stosowanie list	784
Obsługa zdarzeń generowanych przez listy	786
Zarządzanie paskami przewijania	787
Obsługa zdarzeń generowanych przez paski przewijania	789
Stosowanie kontrolek typu TextField	790
Obsługa zdarzeń generowanych przez kontrolkę TextField	792
Stosowanie kontrolek typu TextArea	794

Wprowadzenie do menadżerów układu graficznego komponentów	795
FlowLayout	797
BorderLayout	799
Stosowanie obramowań	800
GridLayout	801
Klasa CardLayout	803
Klasa GridBagLayout	806
Menu i paski menu	811
Okna dialogowe	817
FileDialog	822
Obsługa zdarzeń przez rozszerzanie dostępnych komponentów AWT	823
Rozszerzanie kontrolki Button	824
Rozszerzanie kontrolki Checkbox	825
Rozszerzanie komponentu grupy pól wyboru	826
Rozszerzanie kontrolki Choice	827
Rozszerzanie kontrolki List	828
Rozszerzanie kontrolki Scrollbar	829
Poznanie środowiska kontrolek, menu i menadżerów układu	830
Rozdział 25. Obrazy	831
Formaty plików	832
Podstawy przetwarzania obrazów: tworzenie, wczytywanie i wyświetlanie	832
Tworzenie obiektu obrazu	832
Wczytywanie obrazu	833
Wyświetlanie obrazu	833
Interfejs ImageObserver	835
Podwójne buforowanie	836
Klasa MediaTracker	839
Interfejs ImageProducer	842
Klasa MemoryImageSource	843
Interfejs ImageConsumer	844
Klasa PixelGrabber	845
Klasa ImageFilter	847
Klasa CropImageFilter	848
Klasa RGBImageFilter	850
Animacja poklatkowa	861
Dodatkowe klasy obsługujące obrazy	863
Rozdział 26. Narzędzia współbieżności	865
Pakiety interfejsu Concurrent API	866
java.util.concurrent	866
java.util.concurrent.atomic	867
java.util.concurrent.locks	867
Korzystanie z obiektów służących do synchronizacji	868
Semaphore	868
Klasa CountdownLatch	874
CyclicBarrier	875
Klasa Exchanger	878
Korzystanie z egzekutorów	880
Przykład prostego egzekutora	881
Korzystanie z interfejsów Callable i Future	883
Obiekty typu TimeUnit	885
Kolekcje współbieżne	887
Blokady	887
Operacje atomowe	890
Pakiet Concurrency Utilities a tradycyjne metody języka Java	891

Rozdział 27. System NIO, wyrażenia regularne i inne pakiety	893
Pakiety głównego API języka Java	893
System NIO	895
Podstawy systemu NIO	896
Zestawy znaków i selektory	898
Korzystanie z systemu NIO	899
Czy system NIO jest przyszłością operacji wejścia-wyjścia?	905
Przetwarzanie wyrażen regularnych	905
Klasa Pattern	906
Klasa Matcher	906
Składnia wyrażen regularnych	907
Przykład wykorzystania dopasowywania do wzorca	908
Typy operacji dopasowywania do wzorca	913
Przegląd wyrażen regularnych	914
Refleksje	914
Zdalne wywoływanie metod (RMI)	917
Prosta aplikacja typu klient-serwer wykorzystująca RMI	918
Formatowanie tekstu	921
Klasa DateFormat	921
Klasa SimpleDateFormat	923
 Część III Pisanie oprogramowania w języku Java	927
Rozdział 28. Java Beans	929
Czym jest komponent typu Java Bean?	929
Zalety komponentów Java Beans	930
Introspekcja	930
Wzorce właściwości	931
Wzorce projektowe dla zdarzeń	932
Metody i wzorce projektowe	932
Korzystanie z interfejsu BeanInfo	933
Właściwości ograniczone	933
Trwałość	934
Interfejs Customizer	934
Interfejs Java Beans API	934
Klasa Introspector	936
KlasaPropertyDescriptor	937
Klasa EventSetDescriptor	937
Klasa MethodDescriptor	937
Przykład komponentu typu Bean	937
 Rozdział 29. Przewodnik po pakiecie Swing	941
Klasa JApplet	942
Klasy JFrame i JComponent	943
Ikony i etykiety	943
Problemy z wątkami	945
Pola tekstowe	947
Przyciski	948
Klasa JButton	949
Pola wyboru	951
Przyciski opcji.....	953
Listy kombinowane	955
Okna z zakładkami	957
Okna przewijane	959
Drzewa	961
Przegląd pakietu Swing	965

Rozdział 30. Serwlety	967
Podstawy	967
Cykl życia serwletu	968
Korzystanie ze środowiska Tomcat	969
Przykład prostego serwletu	970
Tworzenie i kompilacja kodu źródłowego serwletu	971
Uruchamianie środowiska Tomcat	971
Uruchamianie przeglądarki i generowanie zapytania	972
Interfejs Servlet API	972
Pakiet javax.servlet	972
Interfejs Servlet	973
Interfejs ServletConfig	974
Interfejs ServletContext	974
Interfejs ServletRequest	975
Interfejs ServletResponse	975
Klasa GenericServlet	976
Klasa ServletInputStream	976
Klasa ServletOutputStream	977
Klasy wyjątków związanych z serwletami	977
Odczytywanie parametrów serwletu	977
Pakiet javax.servlet.http	979
Interfejs HttpServletRequest	979
Interfejs HttpServletResponse	979
Interfejs HttpSession	980
Interfejs HttpSessionBindingListener	981
Klasa Cookie	981
Klasa HttpServlet	983
Klasa HttpSessionEvent	983
Klasa HttpSessionBindingEvent	984
Obsługa zapytań i odpowiedzi HTTP	985
Obsługa zapytań HTTP GET	985
Obsługa zapytań HTTP POST	986
Korzystanie ze znaczników kontekstu użytkownika	988
Śledzenie sesji	990
 Część IV Zastosowanie Javy w praktyce	 993
Rozdział 31. Aplet i serwlety finansowe	995
Znajdowanie raty pożyczki	996
Pola apletu	999
Metoda init()	1000
Metoda actionPerformed()	1002
Metoda paint()	1002
Metoda compute()	1003
Znajdowanie przyszłej wartości inwestycji	1004
Znajdowanie wkładu początkowego wymaganego	
do uzyskania przyszłej wartości inwestycji	1007
Znalezienie inwestycji początkowej wymaganej	
do uzyskania odpowiedniej emerytury	1011
Znajdowanie maksymalnej emerytury dla danej inwestycji	1015
Obliczenie pozostałej kwoty do spłaty kredytu	1019
Tworzenie serwletów finansowych	1022
Konwersja apletu RegPay do serwletu	1023
Serwlet RegPayS	1023
Możliwe rozszerzenia	1026

Rozdział 32. Wykonanie menedżera pobierania plików w Javie1027

Sposoby pobierania plików z internetu	1028
Omówienie programu	1028
Klasa Download	1029
Zmienne pobierania	1033
Konstruktor klasy	1033
Metoda download()	1033
Metoda run()	1033
Metoda stateChanged()	1037
Metody akcesorowe i działań	1037
Klasa ProgressRenderer	1037
Klasa DownloadsTableModel	1038
Metoda addDownload()	1040
Metoda clearDownload()	1041
Metoda getColumnClass()	1041
Metoda getValueAt()	1041
Metoda update()	1042
Klasa DownloadManager	1042
Zmienne klasy DownloadManager	1047
Konstruktor klasy	1048
Metoda verifyUrl()	1048
Metoda tableSelectionChanged()	1049
Metoda updateButtons()	1049
Obsługa zdarzeń akcji	1050
Kompilacja i uruchamianie programu	1051
Rozszerzanie możliwości programu	1051

Dodatki1053**Dodatek A Korzystanie z komentarzy dokumentacyjnych Javy1055**

Znaczniki komentarzy dokumentacyjnych	1055
Znacznik @author	1056
Znacznik {@code}	1057
Znacznik @deprecated	1057
Znacznik {@docRoot}	1057
Znacznik @exception	1057
Znacznik {@inheritDoc}	1057
Znacznik {@link}	1057
Znacznik {@linkplain}	1058
Znacznik {@literal}	1058
Znacznik @param	1058
Znacznik @return	1058
Znacznik @see	1058
Znacznik @serial	1059
Znacznik @serialData	1059
Znacznik @serialField	1059
Znacznik @since	1059
Znacznik @throws	1059
Znacznik {@value}	1059
Znacznik @version	1060
Ogólna postać komentarzy dokumentacyjnych	1060
Wynik działania narzędzia javadoc	1060
Przykład korzystający z komentarzy dokumentacyjnych	1061

Skorowidz1063

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.

Rozmiar 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 szczególnie 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 Intel 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
<code>double</code>	64	od $4.9\text{e}-324$ do $1.8\text{e}+308$
<code>float</code>	32	od $1.4\text{e}-045$ do $3.4\text{e}+038$

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 złotych i groszy.

Oto przykład deklaracji zmiennych typu `float`.

```
float hightemp, lowtemp;
```

Typ `double`

Zmienne o podwójnej precyzji zajmujące 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 potęgze 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
<code>\ddd</code>	Znak jako liczba ósemkowa (ddd)
<code>\uxxxx</code>	Znak Unicode jako liczba szesnastkowa (xxxx)
<code>\'</code>	Apostrof
<code>\"</code>	Cudzysłów
<code>\\</code>	Lewy ukośnik
<code>\r</code>	Powrót karetki
<code>\n</code>	Nowy wiersz (nazywany również przesunięciem papieru)
<code>\f</code>	Wysunięcie kartki
<code>\t</code>	Znak tabulacji
<code>\b</code>	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) odkładałem 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 niejawną 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 literalów 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 dokładnie 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

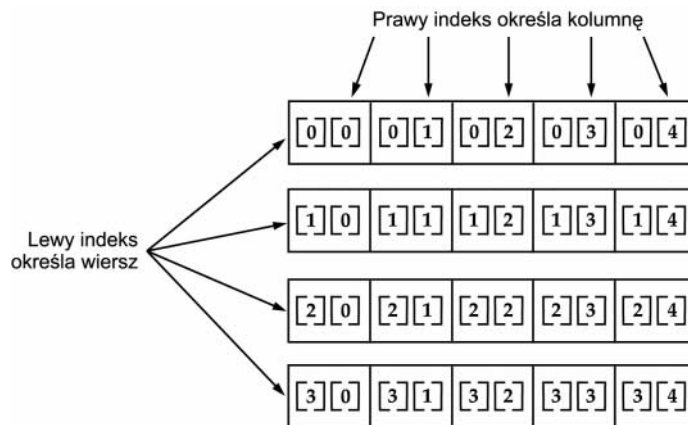
W Javie **tablice wielowymiarowe** są tak naprawdę tablicami tablic. Działają one dokładnie tak samo jak rzeczywiste tablice wielowymiarowe, choć występuje kilka subtelnych różnic. W celu zadeklarowania tablicy wielowymiarowej należy dodać dodatkowe indeksy, używając kolejnych par nawiasów kwadratowych.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

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

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

Rysunek 3.1.

*Widok koncepcyjny
tablicy dwuwymiarowej
o wymiarach 4 na 5*



Instrukcja: `int twoD [][] = new int [4] [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

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

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 wypełnia 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.

Przjrzyjmy się bardziej złożonemu przykładowi, który wykorzystuje tablicę wielowymiarową. Kolejny program tworzy tablicę trójwymiarową 3 na 4 na 5. Następnie wypełnia 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.