helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

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

Thinking in Java
edycja polska



Autor: Bruce Eckel
Tłumaczenie: Adrian Nowak, Szymon Kobalczyk, Łukasz Fryz
ISBN: 83-7197-452-3
Tytuł oryginału: Thinking in Java. Second Edition.
Format: B5, stron: około 816
Przykłady na ftp: 328 kB

„Najlepsza książka na temat Javy", „Jeden ze zdecydowanie najlepszych kursów Javy, jaki kiedykolwiek widziałem, dla jakiegokolwiek języka" – to wybrane opinie o propozycji wydawnictwa Helion.
Książka zarówno dla początkujących, jak i ekspertów:

- Uczy języka Java, nie zaś mechanizmów zależnych od platformy systemowej.
- Poprzez omówienie podstaw wprowadza tematykę zaawansowaną.
- Omawia ponad 300 działających programów Javy, ponad 15 000 linii kodu.
- Dogłębnie objaśnienia zasady obiektowości oraz ich zastosowania w Javie.

Począwszy od podstaw składni Javy do jej najbardziej zaawansowanych właściwości (obliczenia rozproszone, zaawansowany potencjał obiektowy, wielowątkowość), książka ta została napisana przede wszystkim po to, by rozbudzić w początkującym programiście zainteresowanie Javą. Przystępny styl Bruce'a Eckela i ukierunkowane przykłady powodują, iż nawet najbardziej tajemnicze pojęcia stają się zrozumiałe. Bruce Eckel jest autorem książki „Thinking in C++", która zdobyła nagrodę Software Development Jolt Award dla najlepszej książki 1995 roku. Programowaniem zajmuje się profesjonalnie od 20 lat. Uczy ludzi na całym świecie, jak programować z zastosowaniem obiektów już od 1986 roku, najpierw jako konsultant C++ a teraz Javy. Był członkiem Komitetu Standardów C++, napisał 5 innych książek na temat programowania zorientowanego obiektowego, wydał ponad 150 artykułów i prowadził felietony w wielu magazynach informatycznych. Stworzył ścieżkę C++, Javy i Pythona na konferencji Software Development Conference. Zdobył tytuł naukowy z zakresu Zastosowań Fizyki oraz tytuł magistra z zakresu Inżynierii Oprogramowania.

- Nagroda dla najlepszej książki przyznana przez czytelników JavaWorld w 2000 roku.
- Nagroda dla najlepszej książki przyznana przez redakcję Java Developer Journal, 1999.
- Nagroda za twórczość od Software Development Magazine, 1999.

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

Spis treści

Od tłumaczy *19*

Od korektora merytorycznego *20*

Przedmowa *21*

Przedmowa do wydania drugiego 23

Java 2 24

Wprowadzenie *25*

Warunki wstępne 25

Nauka Javy 26

Cele 26

Dokumentacja on-line 27

Zawartość rozdziałów 27

Ćwiczenia 32

Kody źródłowe 32

Konwencje zapisu 33

Wersje Javy 34

Seminaria 34

Projekt okładki 35

Podziękowania 35

Współpracownicy internetowi 37

Rozdział 1. *Wprowadzenie w świat obiektów* *39*

Postępująca abstrakcja 40

Obiekt posiada interfejs 41

Ukrywanie implementacji 43

Wielokrotne wykorzystanie implementacji 44

Dziedziczenie: wielokrotne użycie interfejsu 45

Relacja „bycia czymś" a relacja „bycia podobnym do czegoś" 48

Wymienialność obiektów z użyciem polimorfizmu 49

Abstrakcyjne klasy bazowe i interfejsy 52

Obiekty, sposób przechowywania i czas życia ... 53
Kolekcje i iteratory ... 54
Hierarchia z pojedynczym korzeniem ... 55
Biblioteki kolekcji i ich stosowanie ... 56
Dylemat domowy: kto powinien posprzątać? ... 57
Obsługa wyjątków — eliminowanie błędów ... 58
Wielowątkowość ... 59
Trwałość ... 60
Java i Internet ... 60
Czym jest Internet? ... 60
Programowanie po stronie klienta ... 62
Programowanie po stronie serwera ... 67
Osobny obszar: aplikacje ... 67
Analiza i projektowanie ... 68
Etap 0. Zrób plan ... 70
Etap 1. Co mamy stworzyć? ... 70
Etap 2. Jak to zrobimy? ... 73
Etap 3. Zbuduj jądro ... 76
Etap 4. Przeglądaj przypadki użycia ... 76
Etap 5. Ewolucja ... 77
Planowanie popłaca ... 78
Programowanie ekstremalne (Extreme Programming) ... 79
Najpierw napisz testy ... 79
Programowanie w parach ... 80
Dlaczego Java odnosi sukcesy ... 81
Systemy jest łatwiej opisać i zrozumieć ... 81
Maksymalne zwiększenie wydajności dzięki bibliotekom ... 81
Obsługa błędów ... 82
Programowanie na wielką skalę ... 82
Strategie przejścia ... 82
Wskazówki ... 82
Problemy zarządzania ... 84
Java kontra C++ ... 85
Podsumowanie ... 86

Rozdział 2. *Wszystko jest obiektem* ... ***89***

Dostęp do obiektów poprzez referencje ... 89
Wszystkie obiekty trzeba stworzyć ... 90
Gdzie przechowujemy dane? ... 91
Przypadek specjalny: typy podstawowe ... 92
Tablice w Javie ... 93

Nigdy nie ma potrzeby niszczenia obiektu ... 94
Zasięg ... 94
Zasięg obiektów ... 94
Tworzenie nowych typów danych — klasa ... 95
Pola i metody ... 96
Metody, argumenty i wartości zwracane ... 97
Lista argumentów ... 98
Tworzenie programu w Javie ... 99
Widoczność nazw ... 99
Wykorzystanie innych komponentów ... 100
Słowo kluczowe static ... 101
Twój pierwszy program w Javie ... 102
Kompilacja i uruchomienie ... 103
Komentarze oraz dokumentowanie kodu ... 104
Dokumentacja w komentarzach ... 105
Składnia ... 105
Osadzony HTML ... 106
@see: odwołanie do innych klas ... 106
Znaczniki dokumentowania klas ... 107
Znaczniki dokumentacji zmiennych ... 107
Znaczniki dokumentacji metod ... 107
Przykład dokumentowania kodu ... 108
Styl programowania ... 109
Podsumowanie ... 110
Ćwiczenia ... 110

Rozdział 3. *Sterowanie kolejnością wykonania* ... ***111***

Używanie operatorów Javy ... 111
Kolejność ... 111
Przypisanie ... 112
Operatory matematyczne ... 114
Operatory zwiększania i zmniejszania ... 116
Operatory relacji ... 117
Operatory logiczne ... 118
Operatory bitowe ... 120
Operatory przesunięcia ... 121
Operator trójargumentowy if-else ... 124
Przecinek ... 125
Łańcuchowy operator + ... 125
Najczęstsze pułapki przy używaniu operatorów ... 126
Operatory rzutowania ... 126
W Javie nie ma „sizeof” ... 129

Powtórka z kolejności operatorów ... 129
Kompendium operatorów ... 130
Sterowanie wykonaniem ... 138
Prawda i fałsz ... 138
if-else ... 138
return ... 139
Iteracja ... 139
do-while ... 140
for ... 140
break i continue ... 142
switch ... 146
Podsumowanie ... 150
Ćwiczenia ... 150

Rozdział 4. *Inicjalizacja i sprzątanie* ... *151*

Gwarantowana inicjalizacja przez konstruktor ... 151
Przeciążanie metod ... 153
Rozróżnianie przeciążonych metod ... 155
Przeciążanie a typy podstawowe ... 155
Przeciążanie przez wartości zwracane ... 158
Konstruktory domyślne ... 159
Słowo kluczowe this ... 159
Sprzątanie: finalizacja i odśmiecanie pamięci ... 162
Do czego służy finalize()? ... 163
Musimy przeprowadzić sprzątanie ... 164
Warunek śmierci ... 167
Jak działa odśmiecacz pamięci ... 168
Inicjalizacja składowych ... 171
Określanie sposobu inicjalizacji ... 172
Inicjalizacja w konstruktorze ... 173
Inicjalizacja tablic ... 178
Tablice wielowymiarowe ... 182
Podsumowanie ... 185
Ćwiczenia ... 185

Rozdział 5. *Ukrywanie implementacji* ... *187*

Pakiet — jednostka biblioteczna ... 188
Tworzenie unikatowych nazw pakietów ... 189
Własna biblioteka narzędziowa ... 192
Wykorzystanie instrukcji import do zmiany zachowania ... 193
Pułapka związana z pakietami ... 195

Modyfikatory dostępu w Javie ... 195
Dostęp „przyjazny” ... 195
public: dostęp do interfejsu ... 196
private: nie dotykać! ... 198
protected: „pewien rodzaj przyjacielskości” ... 199
Interfejs i implementacja ... 200
Dostęp do klas ... 201
Podsumowanie ... 204
Ćwiczenia ... 205

Rozdział 6. *Wielokrotne wykorzystanie klas* ... ***207***

Składnia kompozycji ... 207
Składnia dziedziczenia ... 210
Inicjalizacja klasy bazowej ... 212
Łączenie kompozycji i dziedziczenia ... 214
Zapewnienie poprawnego sprzątania ... 215
Ukrywanie nazw ... 218
Wybór między kompozycją a dziedziczeniem ... 219
protected ... 220
Przyrostowe tworzenie oprogramowania ... 221
Rzutowanie w górę ... 221
Dlaczego „w górę”? ... 222
Słowo kluczowe final ... 223
Zmienne ostateczne ... 223
Metody ostateczne ... 227
Klasy ostateczne ... 228
Ostrożnie z ostatecznością ... 229
Inicjalizacja i ładowanie klas ... 230
Inicjalizacja w przypadku dziedziczenia ... 230
Podsumowanie ... 232
Ćwiczenia ... 232

Rozdział 7. *Polimorfizm* ... ***235***

Rzutowanie w górę raz jeszcze ... 235
Zapominanie o typie obiektu ... 236
Mały trik ... 238
Wiązanie wywołania metody ... 238
Uzyskiwanie poprawnego działania ... 239
Rozszerzalność ... 241

Przesłanianie kontra przeciążanie ... 244
Klasy i metody abstrakcyjne ... 245
Konstruktory a polimorfizm ... 248
Kolejność wywołań konstruktorów ... 248
Dziedziczenie a metoda finalize() ... 250
Zachowanie się metod polimorficznych wewnątrz konstruktorów ... 253
Projektowanie z użyciem dziedziczenia ... 255
Czyste dziedziczenie kontra rozszerzanie ... 256
Rzutowanie w dół a identyfikacja typu w czasie wykonania ... 257
Podsumowanie ... 259
Ćwiczenia ... 259

Rozdział 8. *Interfejsy i klasy wewnętrzne* ... *261*

Interfejsy ... 261
„Wielokrotne dziedziczenie" w Javie ... 264
Rozszerzanie interfejsu poprzez dziedziczenie ... 267
Grupowanie stałych ... 268
Inicjalizacja pól interfejsów ... 269
Zagnieżdżanie interfejsów ... 270
Klasy wewnętrzne ... 272
Klasy wewnętrzne a rzutowanie w górę ... 274
Klasy wewnętrzne w metodach i zakresach ... 276
Anonimowe klasy wewnętrzne ... 278
Połączenie z klasą zewnętrzną ... 280
Statyczne klasy wewnętrzne ... 282
Odwoływanie się do obiektu klasy zewnętrznej ... 284
Sięganie na zewnątrz z klasy wielokrotnie zagnieżdżonej ... 285
Dziedziczenie po klasach wewnętrznych ... 285
Czy klasy wewnętrzne mogą być przesłaniane? ... 286
Identyfikatory klas wewnętrznych ... 288
Dlaczego klasy wewnętrzne? ... 288
Klasy wewnętrzne a szkielety sterowania ... 293
Podsumowanie ... 299
Ćwiczenia ... 299

Rozdział 9. *Przechowywanie obiektów* ... *303*

Tablice ... 303
Tablice to obiekty ... 304
Tablice jako wartości zwracane ... 307
Klasa Arrays ... 308
Wypełnianie tablicy ... 317
Kopiowanie tablic ... 318

Porównywanie tablic ... 319
Porównywanie elementów tablic ... 320
Sortowanie tablic ... 323
Przeszukiwanie tablicy posortowanej ... 324
Podsumowanie wiadomości o tablicach ... 325
Wprowadzenie do kontenerów ... 326
Wypisanie zawartości kontenerów ... 327
Wypełnianie kontenerów ... 328
Wada kontenera: nieznany typ ... 333
Czasami mimo wszystko działa ... 335
Tworzenie świadomej typu klasy ArrayList ... 336
Iteratory ... 337
Rodzaje kontenerów ... 340
Interfejs Collection ... 343
Interfejs List ... 345
Stos na podstawie LinkedList ... 348
Kolejka na podstawie LinkedList ... 349
Interfejs Set ... 350
SortedSet ... 352
Interfejs Map ... 352
SortedMap ... 356
Haszowanie i kody haszujące ... 356
Przesłonięcie metody hashCode() ... 363
Przechowywanie referencji ... 365
WeakHashMap ... 367
Iteratory ponownie ... 369
Wybór implementacji ... 370
Wybór między listami ... 370
Wybór implementacji zbioru ... 373
Wybór implementacji odwzorowania ... 375
Sortowanie i przeszukiwanie list ... 377
Dodatkowe usługi ... 377
Niemodyfikowalne kontenery Collection i Map ... 378
Synchronizacja Collection i Map ... 379
Nie obsługiwane operacje ... 380
Kontenery Java 1.0 i 1.1 ... 382
Vector i Enumeration ... 383
Hashtable ... 384
Stack ... 384
BitSet ... 385
Podsumowanie ... 386
Ćwiczenia ... 387

Rozdział 10. *Obsługa błędów za pomocą wyjątków*.....391

Podstawy obsługi wyjątków.....392
Parametry wyjątków.....393
Przechwytywanie wyjątku.....393
Blok try.....394
Obsługa wyjątków.....394
Tworzenie własnych wyjątków.....395
Specyfikacja wyjątków.....398
Przechwytywanie dowolnego wyjątku.....399
Ponowne wyrzucanie wyjątków.....401
Standardowe wyjątki Javy.....404
Specjalny przypadek RuntimeException.....404
Robienie porządków w finally.....406
Do czego służy finally?.....407
Pułapka: zagubiony wyjątek.....409
Ograniczenia wyjątków.....410
Konstruktory.....413
Dopasowywanie wyjątków.....416
Wskazówki.....417
Podsumowanie.....418
Ćwiczenia.....418

Rozdział 11. *System wejścia-wyjścia w Javie*.....421

Klasa File.....421
Wypisywanie zawartości katalogu.....422
Operacje na katalogach.....425
Wejście i wyjście.....427
Typy InputStream.....427
Typy OutputStream.....429
Dodawanie atrybutów i użytecznych interfejsów.....430
Czytanie z InputStream za pomocą FilterInputStream.....430
Zapis do OutputStream za pomocą FilterOutputStream.....431
Czytelnicy i pisarze.....433
Źródła i ujścia danych.....433
Modyfikacja zachowania strumienia.....434
Klasy nie zmienione.....435
Osobna i samodzielna RandomAccessFile.....435
Typowe zastosowania strumieni I/O.....436
Strumienie wejścia.....438
Strumienie wyjścia.....440
Strumienie typu „pipe”.....441

Standardowe wejście-wyjście ... 442
Czytanie ze standardowego wejścia ... 442
Zamiana System.out na PrintWriter ... 443
Przekierowywanie standardowego wejścia-wyjścia ... 443
Kompresja ... 444
Prosta kompresja do formatu GZIP ... 445
Przechowywanie wielu plików w formacie Zip ... 446
Archiwa Javy (JAR) ... 448
Serializacja obiektów ... 450
Odnajdywanie klasy ... 453
Kontrola serializacji ... 454
Stosowanie trwałości ... 462
Atomizacja wejścia ... 468
StreamTokenizer ... 468
StringTokenizer ... 470
Sprawdzanie poprawności stylu ... 472
Podsumowanie ... 479
Ćwiczenia ... 480

Rozdział 12. *Identyfikacja typu w czasie wykonania* ... *483*

Potrzeba mechanizmu RTTI ... 483
Obiekt Class ... 485
Sprawdzanie przed rzutowaniem ... 488
Składnia RTTI ... 494
Odzwierciedlenia — informacja o klasie w czasie wykonania ... 496
Ekstraktor metod klasowych ... 497
Podsumowanie ... 501
Ćwiczenia ... 502

Rozdział 13. *Tworzenie okienek i apletów* ... *505*

Podstawy tworzenia apletów ... 507
Ograniczenia apletów ... 507
Zalety apletów ... 508
Szkielet aplikacji ... 508
Uruchamianie apletów w przeglądarce internetowej ... 509
Wykorzystanie programu Appletviewer ... 511
Testowanie apletów ... 511
Uruchamianie apletów z wiersza poleceń ... 512
Platforma prezentacyjna ... 514
Wykorzystanie Windows Explorer ... 516
Tworzenie przycisku ... 516

Przechwytywanie zdarzenia ... 517
Pola tekstowe ... 520
Rozmieszczenie komponentów ... 521
BorderLayout ... 521
FlowLayout ... 522
GridLayout ... 523
GridBagLayout ... 524
Bezpośrednie pozycjonowanie ... 524
BoxLayout ... 524
Najlepsze rozwiązanie? ... 527
Model zdarzeń w Swingu ... 528
Rodzaje zdarzeń i odbiorców ... 528
Śledzenie wielu zdarzeń ... 533
Katalog komponentów Swing ... 536
Przyciski ... 536
Ikony ... 539
Podpowiedzi ... 540
Pola tekstowe ... 540
Ramki ... 542
Panele z paskami przewijania ... 543
Miniedytor ... 545
Pola wyboru ... 546
Przyciski wyboru ... 547
Listy rozwijane ... 548
Listy ... 549
Zakładki ... 551
Okienka komunikatów ... 552
Menu ... 554
Menu kontekstowe ... 559
Rysowanie ... 560
Okienka dialogowe ... 563
Okienka dialogowe plików ... 566
HTML w komponentach Swing ... 568
Suwaki i paski postępu ... 569
Drzewa ... 570
Tabele ... 572
Zmiana wyglądu aplikacji ... 573
Schowek ... 575
Pakowanie apletu do pliku JAR ... 578
Techniki programowania ... 578
Dynamiczne dołączanie zdarzeń ... 579
Oddzielenie logiki biznesowej od interfejsu użytkownika ... 580
Postać kanoniczna ... 582

Programowanie wizualne i Beany ... 583
Czym jest Bean? ... 584
Wydobycie informacji o Beanie poprzez introspektor ... 586
Bardziej wyszukany Bean ... 591
Pakowanie Beana ... 594
Bardziej złożona obsługa Beanów ... 595
Więcej o Beanach ... 596
Podsumowanie ... 596
Ćwiczenia ... 597

Rozdział 14. *Wielowątkowość* ... *601*

Interaktywny interfejs użytkownika ... 601
Dziedziczenie z klasy Thread ... 603
Wielowątkowość do budowy interaktywnego interfejsu ... 605
Połączenie wątku z klasą główną ... 607
Tworzenie wielu wątków ... 609
Wątki demony ... 611
Współdzielenie ograniczonych zasobów ... 613
Niewłaściwy dostęp do zasobów ... 613
Jak Java współdzieli zasoby ... 617
JavaBeans w innym wydaniu ... 621
Blokowanie ... 625
Zablokowanie ... 625
Impas ... 634
Priorytety ... 638
Odczyt i ustawienie priorytetów ... 638
Grupy wątków ... 641
Runnable ... 647
Zbyt wiele wątków ... 650
Podsumowanie ... 652
Ćwiczenia ... 654

Rozdział 15. *Przetwarzanie rozproszone* ... *657*

Programowanie sieciowe ... 658
Identyfikowanie maszyny ... 658
Gniazda ... 661
Obsługa wielu klientów ... 666
Datagramy ... 671
Użycie adresów URL w apletach ... 671
Więcej o sieciach ... 673

Java DataBase Connectivity (JDBC) ... 673
Uruchamianie przykładu ... 676
Wersja programu wyszukującego z interfejsem graficznym ... 679
Dlaczego interfejs JDBC wydaje się tak skomplikowany ... 681
Bardziej wyrafinowany przykład ... 682
Serwlety ... 688
Podstawowy serwlet ... 689
Serwlety a wielowątkowość ... 692
Obsługiwanie sesji w serwletach ... 693
Uruchamianie przykładowych serwletów ... 697
Java Server Pages ... 697
Obiekty niejawne ... 698
Dyrektywy JSP ... 699
Elementy skryptowe JSP ... 700
Wydobywanie pól i ich wartości ... 702
Atrybuty strony JSP oraz zasięg ich ważności ... 703
Manipulowanie sesjami z poziomu stron JSP ... 704
Tworzenie i modyfikowanie cookies ... 705
Podsumowanie tematu JSP ... 706
RMI (Remote Method Invocation) ... 707
Odległe interfejsy ... 707
Implementacja odległego interfejsu ... 708
Tworzenie namiastki i szkieletu ... 710
Użycie odległego obiektu ... 711
CORBA ... 712
Podstawy CORBA ... 712
Przykład ... 714
Aplety Javy i CORBA ... 718
CORBA a RMI ... 718
Enterprise JavaBeans ... 719
JavaBeans a EJB ... 720
Specyfikacja EJB ... 720
Komponenty EJB ... 720
Części komponentu EJB ... 722
Działanie EJB ... 723
Rodzaje EJB ... 723
Tworzenie EJB ... 724
Podsumowanie EJB ... 728
Jini — serwisy rozproszone ... 728
Zadania Jini ... 728
Co to jest Jini? ... 729
Jak to działa ... 730
Proces odkrywania ... 730

Proces dołączenia ... 730
Proces lokalizacji ... 731
Rozdzielenie interfejsu od implementacji ... 732
Abstrakcja systemów rozproszonych ... 732
Podsumowanie ... 733
Ćwiczenia ... 733

Dodatek A *Przekazywanie i zwracanie obiektów* ... **735**

Przekazywanie referencji ... 735
Odnośniki ... 736
Tworzenie kopii lokalnych ... 738
Przekazywanie przez wartość ... 738
Klonowanie obiektów ... 739
Dodanie klonowalności do klasy ... 740
Udane klonowanie ... 741
Działanie Object.clone() ... 743
Klonowanie złożonego obiektu ... 745
Głęboka kopia ArrayList ... 747
Głęboka kopia poprzez serializację ... 748
Dodanie klonowalności w dół hierarchii ... 750
Dlaczego takie dziwne rozwiązanie? ... 751
Sterowanie klonowalnością ... 751
Konstruktor kopiujący ... 755
Klasy tylko do odczytu ... 759
Tworzenie klas tylko do odczytu ... 760
Wada obiektów odpornych na zmiany ... 761
Niezmienne obiekty String ... 763
Klasy String i StringBuffer ... 765
Łańcuchy są wyjątkowe ... 768
Podsumowanie ... 768
Ćwiczenia ... 769

Dodatek B *Java Native Interface (JNI)* ... **771**

Wywołanie metody rodzimej ... 772
Generator pliku nagłówkowego — javah ... 772
Maglowanie nazw i sygnatury funkcji ... 773
Implementacja biblioteki dynamicznej ... 773
Dostęp do funkcji JNI: argument JNIEnv ... 774
Dostęp do obiektów typu String ... 775
Przekazywanie i używanie obiektów Javy ... 775
JNI i wyjątki Javy ... 777

JNI a wątki ... 778
Wykorzystanie istniejącego kodu ... 778
Dodatkowe informacje ... 778

Dodatek C *Wskazówki dla programistów* ... ***779***
Projekt ... 779
Implementacja ... 783

Dodatek D *Zasoby* ... ***789***
Oprogramowanie ... 789
Książki ... 789
Analiza i projektowanie ... 790
Python ... 792
Lista moich książek ... 792

Skorowidz ... ***795***

14

Wielowątkowość

Obiekty zapewniają nam sposób podziału programu na niezależne części. Często jednak musimy również podzielić program na rozłączne i niezależnie działające podzadania.

Każde z takich niezależnych podzadań jest nazywane *wątkiem* (ang. *thread*) i programuje się go tak, jakby każdy wątek działał samodzielnie, posiadając procesor dla siebie. Pewien mechanizm wewnętrzny w rzeczywistości dzieli czas procesora, ale przeważnie nie musimy się tym martwić, dzięki czemu programowanie wielu wątków jest zadaniem znacznie łatwiejszym.

Proces jest wykonującym się programem z własną przestrzenia adresową. *Wielozadaniowy* system operacyjny jest w stanie uruchomić więcej niż jeden proces (program) równocześnie, chociaż z zewnątrz wygląda, jakby każdy działał samotnie, poprzez okresowe przydzielanie mu cykli procesora. Wątek z kolei jest pojedynczym sekwencyjnym przepływem sterowania, działającym w ramach procesu. Pojedynczy proces może zatem posiadać wiele jednocześnie wykonywanych wątków.

Istnieje wiele zastosowań wielowątkowości, ale zasadniczo będziemy ją wykorzystywać do wiązania pewnej części programu z konkretnym zdarzeniem lub zasobem, i z tego powodu nie będziemy chcieli pozwolić na wstrzymanie pracy reszty programu. Zatem wątek skojarzony ze zdarzeniem albo zasobem ma działać niezależnie od programu głównego. Dobrym przykładem jest przycisk „zamknij" — nie chcemy być zmuszani do badania stanu przycisku w każdym fragmencie napisanego kodu, ale mimo to chcemy, aby przycisk reagował na akcję tak, jakbyśmy *sprawdzali go* regularnie. Faktycznie jednym z najczęstszych powodów wykorzystania wielowątkowości jest stworzenie interfejsu użytkownika zdolnego do reagowania.

Interaktywny interfejs użytkownika

Na początek rozważmy program przeprowadzający działania intensywnie wykorzystujące procesor — w ten sposób ignoruje on działania użytkownika. Ten kombinowany aplet-aplikacja pokaże wyniki działającego licznika:

```
//: c14:Counter1.java
// Niereagujący interfejs użytkownika.
// <applet code=Counter1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
```

```
import java.awt.*;
import com.bruceeckel.swing.*;

public class Counter1 extends JApplet {
  private int count = 0;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Przełącz");
  private JTextField t = new JTextField(10);
  private boolean runFlag = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void go() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
      if (runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter1(), 300, 100);
  }
} ///:~
```

W tym miejscu książki Swing i budowa apletu powinny być już wystarczająco dobrze znane z rozdziału 13. Program jest ciągle zajęty wewnątrz pętli w metodzie `go()` — zamieszcza tam aktualną wartość zmiennej `counter` w polu `t` klasy `JTextField` i ciągle inkrementuje tę zmienną.

Część nieskończonej pętli wewnątrz `go()` zajmuje się wywoływaniem metody `sleep()`. Metoda ta musi być skojarzona z obiektem `Thread` i okazuje się, iż każda aplikacja ma *pewien* wątek skojarzony ze sobą (w istocie Java bazuje na wątkach i w aplikacji zawsze są jakieś działające). Zatem niezależnie od tego, czy otwarcie stosujemy wątki, możemy pozyskać aktualny wątek używany przez program dzięki klasie `Thread` i wywołać jej statyczną metodę `sleep()`.

Zwróć uwagę, iż `sleep()` może wyrzucić wyjątek `InterruptedException`, mimo że jest to uważane jest za nieprzyjazny sposób przerywania wątku i powinno się tego unikać (powtarzam jeszcze raz, wyjątki są dla sytuacji wyjątkowych, a nie zwykłego przepływu sterowania). Przerywanie uśpionego wątku zostało umożliwione, aby wspierać przyszłe właściwości języka.

Kiedy przycisk `start` zostanie naciśnięty, wywołana zostanie metoda `go()`. Analizując `go()`, można naiwnie pomyśleć (tak jak ja), iż powinna ona pozwolić na wielowątkowość, ponieważ usypia. Dokładniej: kiedy metoda jest uśpiona, to mogłoby się zdawać, że procesor mógłby zająć się monitorowaniem innych przycisków. Ale okazuje się, iż prawdziwy problem tkwi w tym, iż wyjście z `go()` nigdy nie nastąpi, jako że ma ona nieskończoną pętlę, a to oznacza, że metoda `actionPerformed()` również nie odda sterowania. Ponieważ utknęliśmy w metodzie `actionPerformed()` po pierwszym wciśnięciu przycisku, program nie może obsłużyć żadnych innych zdarzeń (aby mimo wszystko wyjść, trzeba w jakiś sposób zabić proces — najłatwiejszy to wciśnięcie Control-C w okienku konsoli, jeśli tam został uruchomiony. Jeżeli uruchomienie miało miejsce w przeglądarce, to trzeba zamknąć okno przeglądarki).

Podstawowym problemem jest to, iż `go()` powinna kontynuować swoje działanie, ale równocześnie wymagamy, aby się zakończyła, by również metoda `actionPerformed()` mogła się zakończyć i interfejs użytkownika mógł kontynuować reagowanie na działania użytkownika. W metodzie konwencjonalnej, takiej jak `go()`, nie da się kontynuować *i* w tym samym czasie zwrócić sterowania do reszty programu. Brzmi to jak rzecz niemożliwa do osiągnięcia, ponieważ procesor musiałby być w dwóch miejscach programu naraz, ale jest to właśnie iluzja, jaką dają wątki.

Model wielowątkowy (i jego obsługa w Javie) jest dogodnością programistyczną, upraszczającą zarządzanie kilkoma operacjami równocześnie w pojedynczym programie. W przypadku wątków procesor będzie „skakał" i przydzielał każdemu wątkowi trochę własnego czasu. Wątek jest świadomy przydzielenia procesora, a czas procesora jest w rzeczywistości podzielony między wszystkie takie wątki. Wyjątkiem od tej reguły jest przypadek, kiedy program działa na wielu procesorach. Jedną z najwspanialszych rzeczy dotyczących wątków jest to, iż można abstrahować od tej warstwy do tego stopnia, iż nie ma potrzeby, by kod wiedział, czy naprawdę będzie działał na jedynym czy na wielu procesorach. Zatem wątki są sposobem na tworzenie w sposób przezroczysty dla twórcy programów skalowalnych.

Zastosowanie wątków zmniejsza w pewnym stopniu wydajność, ale polepszenie obsługi sieci w projektowaniu programu, zarządzanie zasobami i wygoda użytkownika są bardzo często znacznie ważniejsze. Oczywiście, jeśli mamy więcej niż jeden procesor, to system operacyjny może dedykować każdy procesor dla zbioru wątków czy nawet pojedynczego wątku i cały program może działać znacznie szybciej[1]. Wielozadaniowość i wielowątkowość zdaje się być najbardziej rozsądnym sposobem użytkowania systemów wieloprocesorowych.

Dziedziczenie z klasy Thread

Najprostszym sposobem na stworzenie wątku jest dziedziczenie z klasy `Thread`, posiadającej wszystkie funkcje, konieczne do tworzenia i uruchomienia wątków. Najważniejszą metodą tej klasy jest metoda `run()`, którą należy przesłonić, aby wątek wypełniał nasze rozkazy. Tak więc metoda `run()` będzie uruchamiana „równocześnie" przez inne wątki programu.

[1] Jeżeli program jest napisany jako jednowątkowy, to niezależnie od liczby procesorów w komputerze będzie się wykonywał tylko na jednym — *przyp. red.*

Następny przykład tworzy pewną liczbę wątków, które nadzoruje poprzez przypisanie każdemu z nich unikatowego numeru generowanego dzięki zmiennej statycznej. Metoda `run()` została przesłonięta, aby odliczać w dół przebiegi zamieszczonej tam pętli, a kończy pracę, gdy licznik osiągnie zero (w miejscu wyjścia z `run()` wątek zostaje przerwany).

```
//: c14:SimpleThread.java
// Bardzo prosty przykład użycia wątków.

public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Tworzenie " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Wątek " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("Wszystkie wątki zostały uruchomione");
  }
} ///:~
```

Metoda `run()` praktycznie zawsze zawiera pewien rodzaj pętli, która wykonuje się, dopóki wątek nie będzie nam już potrzebny, a zatem trzeba ustalić warunek przerwania pętli (albo, jak w powyższym przykładzie, po prostu wyskoczyć, stosując `return`). Często metoda ta jest implementowana jako pętla nieskończona, co oznacza, iż o ile nie będzie jakichś zewnętrznych czynników, które zmuszą ją do przerwania, to będzie trwać bezustannie.

W metodzie `main` następuje stworzenie i uruchomienie grupy wątków. Wywoływana metoda `start()` klasy `Thread` dokonuje specjalnej inicjacji wątku i automatycznie wywołuje `run()`. Tak więc przebiega to w następujący sposób — aby zbudować obiekt, wywoływany jest jego konstruktor, metoda `start()` konfiguruje wątek i wywołuje metodę `run()`. Jeśli nie wywołamy `start()` (co można zrobić w konstruktorze, jeśli jest takowy), to wątek nigdy nie wystartuje.

Wynik programu po uruchomieniu prezentuje się na przykład tak (może różnić się przy kolejnych uruchomieniach):

```
Tworzenie 1
Tworzenie 2
Tworzenie 3
Tworzenie 4
Tworzenie 5
Wątek 1(5)
Wątek 1(4)
Wątek 1(3)
Wątek 1(2)
Wątek 2(5)
Wątek 2(4)
Wątek 2(3)
```

```
Wątek 2(2)
Wątek 2(1)
Wątek 1(1)
Wszystkie wątki zostały uruchomione
Wątek 3(5)
Wątek 4(5)
Wątek 4(4)
Wątek 4(3)
Wątek 4(2)
Wątek 4(1)
Wątek 5(5)
Wątek 5(4)
Wątek 5(3)
Wątek 5(2)
Wątek 5(1)
Wątek 3(4)
Wątek 3(3)
Wątek 3(2)
Wątek 3(1)
```

Jak pewnie zauważyłeś, nigdzie w przykładnie nie ma wywołania `sleep()`, no i wynik działania wskazuje na to, iż każdy wątek dostaje porcję czasu procesora, w którym jest uruchamiany. Pokazuje to, iż metoda `sleep()`, zakładająca istnienie wątku, aby mogła się wykonać, nie jest potrzebna ani przy aktywacji, ani dezaktywacji wielowątkowości. Jest po prostu kolejną metodą.

Widać także, iż wątki nie działają w takiej kolejności, w jakiej zostały utworzone. Faktycznie kolejność zajmowania procesora przez istniejący zestaw wątków pozostaje nieokreślona, dopóki nie ustalimy priorytetów, stosując metodę `setPriority()` z klasy `Thread`.

Kiedy w metodzie `main()` tworzone są obiekty `Thread`, referencje do któregokolwiek z nich nie są wyłapywane. Zwykły obiekt byłby poddany odśmiecaniu pamięci, ale nie wątek. Każdy obiekt `Thread` „rejestruje" się w taki sposób, iż w rzeczywistości gdzieś istnieje do niego referencja i odśmiecacz pamięci nie może się go pozbyć.

Wielowątkowość do budowy interaktywnego interfejsu

Teraz możemy już rozwiązać problem ujawniony w `Counter1.java`. Sztuczka polega na zamieszczeniu podzadania — to znaczy pętli znajdującej się wewnątrz `go()` — wewnątrz metody `run()` jakiegoś wątku. Kiedy użytkownik wciśnie przycisk `start`, wystartuje wątek, ale po chwili jego *tworzenie* zostanie zakończone, metoda obsługi zdarzeń odda sterowanie, więc główna praca programu może przebiegać dalej (obserwowanie i reakcja na zdarzenia interfejsu użytkownika). Oto rozwiązanie:

```
//: c14:Counter2.java
// Reagujący interfejs użytkownika wykorzystujący wątki.
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```
public class Counter2 extends JApplet {
  private class SeparateSubTask extends Thread {
    private int count = 0;
    private boolean runFlag = true;
    SeparateSubTask() { start(); }
    void invertFlag() { runFlag = !runFlag; }
    public void run() {
      while (true) {
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
       if(runFlag)
         t.setText(Integer.toString(count++));
      }
    }
  }
  private SeparateSubTask sp = null;
  private JTextField t = new JTextField(10);
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Przełącz");
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SeparateSubTask();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.invertFlag();
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Counter2 (), 300, 100);
  }
} ///:~
```

`Counter2` **jest prostym programem, którego jedynym zadaniem jest zbudowanie i obsługa interfejsu użytkownika. Teraz, kiedy użytkownik wciśnie przycisk** `start`**, kod obsługi zdarzenia nie wywołuje metody. Zamiast tego tworzony jest wątek klasy** `SeparateSubTask`**, a następnie pętla obsługi zdarzeń klasy** `Counter2` **wraca do pracy.**

Klasa `SeparateSubTask` **jest prostym rozszerzeniem** `Thread` **z konstruktorem uruchamiającym wątek poprzez wywołanie** `start()` **i przez to** `run()`**, która zasadniczo zawiera kod „**`go()`**" z przykładu** `Counter1.java`**.**

Ponieważ `SeparateSubTask` jest klasą wewnętrzną, to może bezpośrednio sięgać do pola `JTextField t` z klasy `Counter2` — można to zaobserwować w metodzie `run()`. Pole `t` w klasie zewnętrznej licznika jest rodzaju `private`, gdyż `SeparateSubTask` i tak ma do niego dostęp bez żadnego specjalnego pozwolenia — dlatego zawsze dobrze jest tworzyć składowe „prywatne na tyle, jak to tylko możliwe", by nie mogły być przypadkowo zmienione z zewnątrz.

Po wciśnięciu przycisku `onOff` nastąpi przełączenie znacznika `runFlag` należącego do obiektu `SeparateSubTask`. Nasz wątek (spoglądając na znacznik) może się następnie uruchomić lub zatrzymać. Wybór przycisku `onOff` daje najwyraźniej rezultat natychmiastowy. Oczywiście odpowiedź nie jest rzeczywiście natychmiastowa, tak jak w systemie działającym na przerwaniach. Licznik zatrzymuje się tylko wtedy, kiedy wątek ma przydzielony procesor i zauważy, że znacznik uległ zmianie.

Można zaobserwować, iż klasa wewnętrzna jest klasą prywatną, co oznacza, że jej pola i metody mogą mieć dostęp domyślny (poza `run()`, która musi być `public`, gdyż jest publiczna w klasie bazowej). Prywatna klasa wewnętrzna nie jest dostępna dla nikogo poza `Counter2` i obie klasy są mocno powiązane. Zawsze, kiedy tylko zauważysz klasy, które wyglądają na bardzo powiązane między sobą, weź pod uwagę poprawę kodowania i utrzymania, które zyskuje się, stosując klasy wewnętrzne.

Połączenie wątku z klasą główną

W poprzednim przykładzie klasa wątku była odseparowana od głównej klasy programu. Ma to sporo sensu i jest relatywnie łatwe do zrozumienia. Jednak istnieje alternatywna forma, którą często widać w zastosowaniu — nie jest tak prosta, ale zazwyczaj jest bardziej zwięzła (co prawdopodobnie przyczyniło się do jej popularności). Postać ta łączy główną klasę programu z klasą wątku poprzez uczynienie klasy głównej wątkiem. Ponieważ dla programu GUI główna klasa programu musi być dziedziczona z klasy `Frame` lub `Applet`, to dla dołożenia dodatkowych funkcji musi zostać wykorzystany interfejs. Interfejs ten nosi nazwę `Runnable` i zawiera tę samą podstawową metodę, którą jest w klasie `Thread`. Tak naprawdę to `Thread` również implementuje interfejs `Runnable`, który oznacza jedynie posiadanie przez klasę metody `run()`.

Użycie połączonego programu-wątku nie jest tak oczywiste. Kiedy uruchamiamy program, powstaje obiekt będący `Runnable`, ale nie uruchamia on wątku — to już trzeba wykonać jawnie. Widać to w następnym programie, który powiela funkcję klasy `Counter2`:

```
//: c14:Counter3.java
// Wykorzystanie interfejsu Runnable do zamiany
// głównej klasy w wątek.
// <applet code=Counter3 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter3
    extends JApplet implements Runnable {
  private int count = 0;
  private boolean runFlag = true;
  private Thread selfThread = null;
```

```
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Przełącz");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
        selfThread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(selfThread == null) {
        selfThread = new Thread(Counter3.this);
        selfThread.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Counter3(), 300, 100);
  }
} ///:~
```

Tym razem metoda `run()` jest wewnątrz naszej klasy, ale wciąż oczekuje zakończenia metody `init()`. Po wciśnięciu przycisku `start` wątek zostanie utworzony (jeśli jeszcze nie istniał) za pomocą pewnej mało jasnej instrukcji:

```
new Thread(Counter3.this);
```[2]

Kiedy coś implementuje interfejs `Runnable`, to oznacza to tylko tyle, iż zawiera metodę `run()`, ale nie ma w tym nic szczególnego — nie daje to żadnych wrodzonych zdolności wątków, jak te w klasie dziedziczącej z `Thread`. Zatem aby uzyskać wątek z obiektu `Runnable`, trzeba, jak to widać powyżej, stworzyć osobny wątek `Thread`, podający obiekt `Runnable` do jego konstruktora. Później można już dla takiego wątku wywołać `start()`:

```
selfThread.start();
```

[2] Możesz nie pamiętać, że konstrukcja `Counter3.this` oznacza: „odwołaj się do referencji this klasy zewnętrznej". Ta możliwość jest zagwarantowana przez specyfikację klas wewnętrznych — *przyp. red.*

Przeprowadzona zostanie zwyczajna inicjacja i wywołana metoda `run()`.

Wygoda interfejsu `Runnable` polega na tym, iż wszystko należy do tej samej klasy. Jeżeli trzeba do czegoś sięgnąć, to po prostu robi się to bez przechodzenia przez jakiś oddzielny obiekt. Jednakże, jak było widać w przykładzie, ten dostęp jest tak prosty, gdy używa się klasy wewnętrznej[3].

Tworzenie wielu wątków

Rozważmy stworzenie wielu różnych wątków. Można to zrobić na poprzednim przykładzie — trzeba wrócić do wersji z oddzielną klasą dziedziczącą z `Thread` hermetyzującą metodę `run()`. Jest to rozwiązanie bardziej uniwersalne i łatwiejsze do zrozumienia — zatem choć poprzedni przykład pokazywał styl często używany, nie mogę go polecać, gdyż jest po prostu odrobinę bardziej mylący i mniej elastyczny.

Następny przykład ma postać poprzednich programów z licznikami i przyciskami przełączającymi. Teraz jednak wszystkie informacje dla konkretnego licznika, włączając w to przycisk i pole tekstowe, są wewnątrz swojego własnego obiektu, wywodzącego się z `Thread`. Wszystkie pola klasy `Ticker` są prywatne, dzięki czemu można później zmieniać implementacje zgodnie z życzeniem, włączając w to ilość i typ danych, aby zdobyć i wyświetlić informację. Podczas tworzenia obiektu `Ticker` konstruktor dodaje swoje wizualne składniki do panelu obiektu zewnętrznego:

```
//: c14:Counter4.java
// Poprzez utrzymanie wątku jako odrębnej klasy
// można utworzyć tyle wątków, ile dusza zapragnie.
// <applet code=Counter4 width=200 height=600>
// <param name="rozmiar" value="12"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size = 12;
  class Ticker extends Thread {
    private JButton b = new JButton("Przełącz");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = true;
    public Ticker() {
      b.addActionListener(new ToggleL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Wywołanie JApplet.getContentPane().add();
```

[3] `Runnable` **był już w Java 1.0, podczas gdy klasy wewnętrzne nie były wprowadzone przed Javą 1.1, co mogło się częściowo przyczynić do zaistnienia** `Runnable`. **Także tradycyjne architektury wielowątkowe skupiają się na uruchamianiu funkcji, a nie obiektu. Osobiście zawsze preferuję dziedziczenie z klasy** `Thread`, **jeśli tylko mogę — wydaje mi się to prostsze i bardziej elastyczne.**

```
        getContentPane().add(p);
      }
      class ToggleL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
          runFlag = !runFlag;
        }
      }
      public void run() {
        while (true) {
          if (runFlag)
            t.setText(Integer.toString(count++));
          try {
            sleep(100);
          } catch(InterruptedException e) {
            System.err.println("Przerwany");
          }
        }
      }
    }
    class StartL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(!started) {
          started = true;
          for (int i = 0; i < s.length; i++)
            s[i].start();
        }
      }
    }
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      // Pobierz parametr "rozmiar" ze strony HTML:
      if (isApplet) {
        String sz = getParameter("rozmiar");
        if(sz != null)
          size = Integer.parseInt(sz);
      }
      s = new Ticker[size];
      for (int i = 0; i < s.length; i++)
        s[i] = new Ticker();
      start.addActionListener(new StartL());
      cp.add(start);
    }
    public static void main(String[] args) {
      Counter4 applet = new Counter4();
      // To nie jest aplet, zatem ustawia znacznik
      // i pobiera wartość parametru z args:
      applet.isApplet = false;
      if(args.length != 0)
        applet.size = Integer.parseInt(args[0]);
      Console.run(applet, 200, applet.size * 50);
    }
} ///:~
```

`Ticker` zawiera nie tylko wyposażenie wątku, ale także sposób jego kontroli i wyświetlania. Możemy stworzyć tyle wątków, ile chcemy, bez konieczności bezpośredniego tworzenia komponentów okienkowych.

W klasie `Couner4` mamy tablicę obiektów `Ticker` o nazwie `s`. Aby zapewnić elastyczność, rozmiar tej tablicy jest inicjowany przez sięgnięcie do strony HTML z parametrami apletu. Tak wygląda parametr rozmiaru na stronie, osadzony wewnątrz znacznika apletu:

```
<param name=rozmiar value="20">
```

Wszystkie słowa `param`, `name` i `value` są słowami kluczowymi HTML. `name` jest nazwą, do której odwołujemy się w programie, natomiast `value` może być dowolnym łańcuchem tekstowym, nie tylko liczbą.

Ustalenie rozmiaru tablicy `s` ma miejsce nie podczas deklaracji tablicy, a w metodzie `init()`. Znaczy to, iż *nie można* w części definicji kasy (poza jakąkolwiek metodą) robić takich rzeczy:

```
int size = Integer.parseInt(getParameter("rozmiar"));
Ticker[] s = new Ticker[size];
```

Da się to skompilować, ale dostaniemy dziwny wyjątek („null-pointer exception") podczas uruchomienia. Działa lepiej, jeżeli przeniesiemy inicjacje poprzez `getParameter()` do wnętrza `init()`. Aplet dokona bowiem koniecznego przygotowania do pobrania parametrów przed wejściem do `init()`.

Ponadto nasz kod jest przygotowany zarówno do pracy jako aplet, jak i jako aplikacja. Jeżeli jest aplikacją, to parametr `size` jest pobierany z wiersza poleceń (albo używana jest wartość domyślna).

Po ustaleniu rozmiaru tablicy tworzony jest nowy obiekt typu `Ticker`; w konstruktorze dodawane są do apletu m.in. przycisk i pole tekstowe każdego obiektu `Ticker`.

Wciśnięcie przycisku `start` oznacza przejście całej tablicy obiektów `Ticker` i wywołanie wobec każdego metody `start()`. Jak pamiętamy, `start()` przeprowadza całą potrzebną inicjację i wywołuje `run()` dla danego wątku.

Klasa obsługi zdarzenia `ToggleL` jedynie zmienia znacznik obiektu `Ticker` na przeciwny i jeżeli skojarzony wątek po raz kolejny ją sprawdzi, to może odpowiednio zareagować.

Wartość przykładu polega na tym, iż pozwala on na łatwe stworzenie dużych zbiorów niezależnych podzadań i monitorowanie ich zachowania. W tym przypadku, jeżeli liczba podzadań wzrośnie, to Twój komputer prawdopodobnie wykaże więcej rozbieżności w prezentowanych wartościach liczbowych, z powodu sposobu, w jaki wątki są obsługiwane.

Radzę również poeksperymentować, aby odkryć, jak ważne jest wywołanie `sleep(100)` zamieszczone wewnątrz `Tickler.run()`. Jeżeli się go usunie, to wszystko działa w porządku, dopóki nie wciśnie się przycisku przełącznika. Wtedy wybrany wątek uzyskuje wartość fałszu dla `runFlag` i `run()` jest po prostu związana w nieskończonej pętli, co okazuje się być trudne do przerwania ze względu na wielowątkowość — więc program przestanie reagować i po prostu stanie.

Wątki demony

Wątek „demon" to wątek, który powinien zapewnić ogólne usługi w tle programu w czasie jego działania, ale nie jest bezpośrednio związany z główną częścią programu. Więc kiedy wszystkie wątki nie będące demonami zakończą pracę, program również. Odwrotnie: jeżeli wciąż są jakieś działające wątki nie będące demonami, to program się nie zakończy (istnieje na przykład wątek, który uruchamia `main()`).

Aby się przekonać, czy wątek jest demonem, wystarczy wywołać jego metodę `isDeamon()`. Można też włączyć lub wyłączyć „demoniczność" wątku dzięki `setDeamon()`. Jeśli wątek jest demonem, to każdy wątek, który stworzy, stanie się również automatycznie demonem.

Kolejny przykład pokazuje wątki demony:

```
//: c14:Daemons.java
// Zachowanie demoniczne.
import java.io.*;

class Daemon extends Thread {
  private static final int SIZE = 10;
  private Thread[] t = new Thread[SIZE];
  public Daemon() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < SIZE; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < SIZE; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}

class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    System.out.println(
      "DaemonSpawn " + i + " wystartowany");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Daemons {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Daemon();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Wątki demony mogą
    // zakończyć pracę:
    System.out.println("Naciśnij jakiś klawisz");
    System.in.read();
  }
} ///:~
```

Wątek `Deamon` ustawia swój znacznik „demoniczności" na „true" i tworzy grupę innych wątków, by pokazać, iż są one również demonami. Następnie wchodzi do nieskończonej pętli wywołującej metodę `yield()`, aby oddać sterowanie innym procesom. W poprzednich wersjach tego programu nieskończone pętle inkrementowały licznik typu `int`, ale to prowadzi do zatrzymania całego programu. Zastosowanie `yield()` pozwala mu natomiast na normalną pracę.

Nie ma w przykładzie nic, co powstrzymałoby przed zakończeniem programu, kiedy swoje zadanie zakończy metoda `main()`, ponieważ uruchomiliśmy tylko wątki demony. Zatem można zaobserwować wynik uruchomienia wszystkich wątków demonów. `System.in` jest przygotowany do odczytu, stąd program będzie czekał na wciśnięcie klawisza, zanim zakończy pracę. Bez tego triku zobaczylibyśmy tylko parę wyników tworzenia wątków demonów (spróbuj zamienić kod `read()` przez wywołanie `sleep()` o różnym czasie uśpienia, aby zaobserwować działanie).

Współdzielenie ograniczonych zasobów

Można myśleć o programie jednowątkowym, jak o samotnej jednostce przemieszczającej się po przestrzeni zadań i wykonującej jedną rzecz naraz. Ponieważ istnieje tylko jedna jednostka, nigdy nie trzeba się martwić sytuacją, że dwie w tym samym czasie spróbują wykorzystać ten sam zasób, jak dwoje ludzi próbujących zaparkować w tym samym miejscu lub przejść przez drzwi w tej samej chwili albo nawet jednocześnie mówić.

W przypadku wielowątkowości samotność już nie istnieje, ale zachodzi możliwość, iż dwa lub więcej wątków będzie usiłowało użyć tego samego ograniczonego zasobu równocześnie. Musimy zapobiec kolizji przy dostępie do zasobu albo będziemy mieli dwa wątki próbujące sięgać do tego samego konta bankowego równocześnie, drukować na tej samej drukarce albo modyfikować tę samą zmienną itp.

Niewłaściwy dostęp do zasobów

Rozważmy zmianę liczników, których używaliśmy do tej pory w niniejszym rozdziale. W naszym kolejnym przykładzie każdy wątek składa się z dwóch liczników inkrementowanych i wyświetlanych w metodzie `run()`. Ponadto istnieje inny wątek klasy `Watcher`, który obserwuje liczniki i sprawdza, czy ich wartości są zawsze sobie równe. Zdaje się to być działaniem zbędnym, gdyż patrząc w kod, wydaje się oczywiste, że zawsze będą takie same. Ale tu właśnie pojawia się niespodzianka. Oto pierwsza wersja programu:

```
//: c14:Sharing1.java
// Problemy ze współdzieleniem dostępu do zasobów w przypadku wątków.
// <applet code=Sharing1 width=350 height=500>
// <param name=rozmiar value="12">
// <param name=obserwatorzy value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing1 extends JApplet {
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
```

```
private JButton
  start = new JButton("Start"),
  watcher = new JButton("Sprawdź");
private boolean isApplet = true;
private int numCounters = 12;
private int numWatchers = 15;
private TwoCounter[] s;
class TwoCounter extends Thread {
  private boolean started = false;
  private JTextField
    t1 = new JTextField(5),
    t2 = new JTextField(5);
  private JLabel l =
    new JLabel("count1 == count2");
  private int count1 = 0, count2 = 0;
  // Umieść komponenty prezentujące na panelu:
  public TwoCounter() {
    JPanel p = new JPanel();
    p.add(t1);
    p.add(t2);
    p.add(l);
    getContentPane().add(p);
  }
  public void start() {
    if(!started) {
      started = true;
      super.start();
    }
  }
  public void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
  public void synchTest() {
    incrementAccess();
    if(count1 != count2)
      l.setText("Różne");
  }
}
class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
}
```

```
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("rozmiar");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("obserwatorzy");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new JLabel("Licznik sprawdzeń"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing1 applet = new Sharing1();
    // To nie jest aplet, zatem ustawia znacznik
    // i pobiera wartości parametrów z args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

Jak poprzednio, każdy licznik zawiera własne komponenty wyświetlające — dwa pola tekstowe i etykietę, która początkowo wskazuje, iż oba liczniki są równe. Komponenty te są dodawane do panelu obiektu klasy zewnętrznej w konstruktorze `TwoCounter`.

Ponieważ wątek jest uruchamiany poprzez naciśnięcie przycisku, to możliwe jest, że metoda `start()` zostanie wywołana więcej niż raz. Jednakże wielokrotne uruchomienie `Thread.start()` jest niedopuszczalne (zgłaszany wyjątek). Tak więc został dodany mechanizm zapobiegający tej sytuacji — znacznik `started` i przesłonięcie metody `start()`.

Wewnątrz `run()` następuje inkrementacja i wyświetlenie obu liczników, `count1` i `count2`, w sposób, który zdaje się zachowywać ich identyczność. Następnie wywoływana jest metoda `sleep()` — bez tego program się zatrzyma, ponieważ procesor będzie miał trudności z przełączaniem zadań.

Metoda `synchTest()` przeprowadza najwidoczniej bezużyteczne sprawdzenie równości `count1` i `count2`; jeżeli ich wartości są różne, tekst etykiety ustawiany jest na „Różne". Wcześniej jednak wywoływana jest statyczna metoda klasy `Sharing1`, inkrementująca i wyświetlająca licznik sprawdzeń, aby pokazać, ile razy doszło do sprawdzenia (przyczyna takiego podejścia stanie się oczywista w dalszych odmianach tego przykładu).

Klasa `Watcher` jest wątkiem, którego zadaniem jest wywołanie metody `synchTest()` dla każdego z aktywnych obiektów klasy `TwoCounter`. Robi to, przechodząc przez tablicę trzymaną w obiekcie `Sharing1`. Można wyobrazić sobie, iż `Watcher` stale zerka przez ramię obiektom `TwoCounter`.

`Sharing1` zawiera tablicę obiektów `TwoCounter`, inicjowaną w metodzie `init()` i startującą zawierające wątki, kiedy wciśnie się przycisk „Start". Później, kiedy wybierze się przycisk „Sprawdź", zostanie stworzony jeden (lub więcej) obserwator i wypuszczony na nic nie podejrzewające wątki `TwoCounter`.

Aby uruchomić aplet w przeglądarce, znacznik apletu powinien zawierać następujące wiersze:

```
<param name=rozmiar value="20">
<param name=obserwatorzy value="1">
```

Można poeksperymentować, zmieniając szerokość, wysokość i parametry wedle własnego gustu. Zmieniając parametry `rozmiar` i `obserwatorzy`, zmienisz zachowanie programu. Program jest także przystosowany do działania jako samodzielna aplikacja poprzez podanie parametrów z wiersza poleceń (albo zdanie się na domyślne).

A teraz niespodzianka. Nieskończona pętla w metodzie `TwoCounter.run()`zwyczajnie powtarza podobne instrukcje:

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

(czasem zostaje uśpiony, ale to nie jest tu istotne). Jeżeli uruchomisz program, to zauważysz, iż `coint1` i `count2` zostaną czasami rozpoznane (przez wątki `Watcher`) jako różne! Dzieje się tak ze względu na właściwości wątków — mogą one zostać zawieszone podczas pracy w dowolnej chwili. Stąd czasami zawieszenie to występuje pomiędzy wykonaniem dwóch powyższych wierszy, a wątek `Watcher` przychodzi i dokonuje sprawdzenia dokładnie w tym momencie, odkrywając, iż liczniki są różne.

Mamy tu zatem podstawowy problem z wykorzystaniem wątków. Nigdy nie wiadomo, kiedy wątek może działać. Wyobraź sobie, że siedzisz przy stole z widelcem, sięgając po ostatni kawałek pokarmu na talerzu i wtedy jedzenie nagle znika (ponieważ Twój wątek został zawieszony, inny przyszedł i ukradł pokarm). Tak właśnie wygląda problem, z którym mamy tu do czynienia.

Czasami nie dbamy o to, czy dany zasób jest używany w tym samym czasie, kiedy chcemy również go użyć (jedzenie jest na jakimś innym talerzu). Jednak aby wielowątkowość działała poprawnie, trzeba w jakiś sposób uniemożliwić dwóm wątkom sięganie do tego samego zasobu, choćby w okresach krytycznych.

Zapobieganie tego rodzaju kolizjom jest po prostu kwestią blokowania (zamknięcia) zasobu, kiedy jakiś wątek już go używa. Pierwszy wątek, który sięgnie do zasobu, zamyka go i wtedy inne wątki nie mają dostępu, póki nie zostanie odblokowany, kiedy to znowu inny wątek zablokuje i wykorzysta go itd. Jeśli przednie siedzenie w samochodzie byłoby naszym zasobem, to dziecko, które krzyknie: „ja!", zablokuje zasób.

Jak Java współdzieli zasoby

Java posiada wbudowane funkcje, zapobiegające kolizjom przy dostępie do jednego z rodzajów zasobów — a mianowicie pamięci przydzielonej dla obiektu. Ponieważ przeważnie dane klasy tworzy się jako prywatne i sięga do pamięci tylko poprzez metody, można uniknąć kolizji, czyniąc metodę synchronizowaną. Tylko jeden wątek w tym samym czasie może wywołać metodę synchronizowaną konkretnego obiektu (choć może wywołać więcej niż jedną metodę synchronizowaną obiektu). Przykładowe metody deklarowane z modyfikatorem `synchronized`:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Każdy obiekt zawiera pojedynczą blokadę (zwaną także *monitorem*), która jest automatycznie częścią obiektu (nie trzeba pisać żadnego dodatkowego kodu). Wywołując metodę typu `synchronized`, powodujemy, że obiekt jest blokowany i żadna inna jego metoda `synchronized` nie może zostać wywołana, zanim pierwsza się nie zakończy i nie zwolni blokady. W powyższym przykładzie jeżeli `f()` zostanie wywołana wobec jakiegoś obiektu, to `g()` nie może być wywołana wobec tego samego obiektu, dopóki `f()` nie zakończy pracy i nie zwolni blokady. Tak więc mamy pojedynczą blokadę, współdzieloną przez wszystkie metody `synchronized` danego obiektu i to ona zapobiega zapisowi do wspólnej pamięci przez więcej niż jedną metodę równocześnie (tj. nie więcej niż przez jeden wątek równocześnie).

Istnieje także jedna taka blokada dla klasy (część obiektu `Class` odpowiadającego danej klasie), a zatem metody `synchronized static` mogą blokować się wzajemnie przed równoczesnym dostępem do statycznych danych klasowych.

Zauważ, iż jeśli chcemy strzec inne zasoby przed jednoczesnym dostępem przez różne wątki, można to uzyskać, wymuszając dostęp do takich zasobów przez metody synchronizowane.

Synchronizowanie liczników

Jesteśmy uzbrojeni w nowe słowo kluczowe i wydaje nam się, iż rozwiązanie synchronizacji liczników jest w zasięgu ręki — po prostu użyjemy słowa kluczowego `synchronized` do metod w klasie `TwoCounter`. Kolejny przykład jest taki sam jak poprzedni, dodajemy jednak nasze nowe słowo kluczowe:

```
//: c14:Sharing2.java
// Wykorzystanie słowa kluczowego synchronized, aby zapobiec
// równoczesnemu dostępowi do jakiegoś zasobu.
```

```
// <applet code=Sharing2 width=350 height=500>
// <param name=rozmiar value="12">
// <param name=obserwatorzy value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing2 extends JApplet {
  TwoCounter[] s;
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Sprawdź");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;

  class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    public TwoCounter() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!started) {
        started = true;
        super.start();
      }
    }
    public synchronized void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Przerwany");
        }
      }
    }
```

```
  public synchronized void synchTest() {
    incrementAccess();
    if(count1 != count2)
      l.setText("Różne");
  }
}

class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
}
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < s.length; i++)
      s[i].start();
  }
}
class WatcherL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    for(int i = 0; i < numWatchers; i++)
      new Watcher();
  }
}
public void init() {
  if(isApplet) {
    String counters = getParameter("rozmiar");
    if(counters != null)
      numCounters = Integer.parseInt(counters);
    String watchers = getParameter("obserwatorzy");
    if(watchers != null)
      numWatchers = Integer.parseInt(watchers);
  }
  s = new TwoCounter[numCounters];
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  for(int i = 0; i < s.length; i++)
    s[i] = new TwoCounter();
  JPanel p = new JPanel();
  start.addActionListener(new StartL());
  p.add(start);
  watcher.addActionListener(new WatcherL());
  p.add(watcher);
  p.add(new Label("Licznik sprawdzeń"));
  p.add(aCount);
  cp.add(p);
}
```

```
  public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // To nie jest aplet, zatem ustawia znacznik
    // i pobiera wartości parametrów z args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

Obie metody `run()` i `synchTest()` są teraz deklarowane jako `synchronized`. Jeżeli zsynchronizowana zostałaby tylko jedna z nich, to druga mogłaby swobodnie ignorować blokadę obiektu i być bezkarnie wywoływana. Jest to ważne — każda metoda, która sięga do współdzielonego zasobu, musi byś synchronizowana albo nie będzie działała poprawnie.

Pojawia się teraz jednak nowa kwestia: `Watcher` może nigdy nie móc sprawdzić, co się dzieje, ponieważ została zsynchronizowana cała metoda `run()`i ponieważ działa ona cały czas dla każdego obiektu, to blokada jest ciągle do któregoś przypisana i metoda `synchTest()` może nigdy nie zostać wywołana. Można to zaobserwować, gdyż `accessCount` nigdy nie ulega zmianie.

To, co chcielibyśmy uzyskać w naszym przykładzie, to sposób na separację tylko *części* kodu wewnątrz `run()`. Część kodu, którą chcemy izolować, jest nazywana *sekcją krytyczną* i aby ją uzyskać, stosuje się słowo kluczowe `synchronized` w trochę inny sposób. Java zapewnia sekcje krytyczne poprzez *bloki synchronizowane*; tym razem słowo `synchronized` jest stosowane do określenia obiektu, którego blokada ma być używana do synchronizacji otoczonego blokiem kodu:

```
synchronized(syncObject) {
  // Ten kod może być dostępny tylko
  // dla jednego wątku równocześnie
}
```

Przed wejściem do bloku synchronizowanego oczywiście blokada obiektu `syncObject` musi być dostępna. Jeżeli jakiś inny wątek już ją zablokował, to zawartość bloku synchronizowanego nie może zostać wykonana, zanim blokada nie zostanie zdjęta.

Przykład `Sharing2` można zmodyfikować, usuwając słowo kluczowe `synchronized` z całej metody `run()`, a zamiast tego zamieścić `synchronized`, obejmujące dwa krytyczne wiersze. Ale jaki obiekt powinien być wybrany jako właściciel zamka? Ten, który został już uwzględniony przez metodę `synchTest()`, czyli aktualny obiekt (`this`)! Zatem po zmianach metoda `run()` prezentuje się tak:

```
public void run() {
    while (true) {
      synchronized(this) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
      }
      try {
        sleep(500);
      } catch(InterruptedException e) {
```

```
            System.err.println("Przerwany");
          }
        }
      }
```

To jest jedyna zmiana, którą trzeba przeprowadzić w przykładzie `Sharing2.java`. Przekonasz się, iż chociaż oba liczniki prawdopodobnie nigdy nie zostaną rozsynchronizowane (w momentach kiedy obserwator `Watcher` będzie miał pozwolenie na zerknięcie do nich), to będziemy mieć również zapewniony właściwy dostęp przez obserwatorów podczas wykonania `run()`.

Oczywiście właściwa synchronizacja zależy od świadomości programisty — każdy fragment kodu, który może sięgać współdzielonych zasobów, musi być opakowany w stosowny blok synchronizujący.

Wydajność synchronizacji

Chociaż posiadanie dwóch metod napisanych dla tego samego kawałka danych *nigdy* nie wygląda na szczególnie dobry pomysł, to może mieć sens wobec wszystkich metod będących automatycznie `synchronized` i generalnie eliminuje słowo kluczowe `synchronized` (oczywiście przykład z `synchronized run()` pokazuje, iż to również nie zadziała). Okazuje się jednak, iż uzyskanie blokady obiektu nie jest operacją tanią — zwielokrotnia koszt wywołania metody (czyli wejście i wyjście z metody, nie zaś wykonanie ciała metody) minimum czterokrotnie i może być znacznie bardziej zależne od implementacji. Zatem jeżeli wiadomo, że metoda nie będzie powodować problemów kolizji, to celowe jest pominięcie słowa kluczowego `synchronized`. Z drugiej strony zaniechanie synchronizacji — ponieważ wydaje się, że jest wąskim gardłem i mamy nadzieję, że nie będzie kolizji — jest proszeniem się o katastrofę.

JavaBeans w innym wydaniu

Po zrozumieniu kwestii synchronizacji możemy ponownie spojrzeć na JavaBeans. Kiedykolwiek tworzy się Bean, trzeba zakładać, iż będzie uruchamiany w środowisku wielowątkowym. Oznacza to, że:

1. Kiedy tylko jest to możliwe, wszystkie metody publiczne Beana powinny być synchronizowane. Oczywiście wywołuje to narzut związany z taką synchronizacją w czasie wykonania. Jeżeli jest to problemem, to metody nie powodujące kłopotów w sekcjach krytycznych mogą pozostać niesynchronizowane, ale trzeba pamiętać, że nie zawsze jest to oczywiste. Pozostawienie niesynchronizowanymi metod, które można zaliczyć do niewielkich (tak jak `getCircleSize()` w kolejnym przykładzie) i „atomowych", czyli takich, które wykonują na tyle niewielką ilość kodu, iż obiekt nie może ulec zmianie podczas ich działania — może nie mieć znaczącego wpływu na czas wykonania programu. Można uczynić wszystkie publiczne metody Beana synchronizowanymi i usunąć słowo kluczowe `synchronized` tylko wtedy, jeżeli na pewno wiadomo, że nie będzie konieczne i że spowoduje to jakąś różnicę.
2. Podczas wysyłania zdarzeń do grupy odbiorców nimi zainteresowanych trzeba brać pod uwagę, iż odbiorcy mogą być dodawani lub usuwani podczas przechodzenia po liście.

Pierwszy punkt jest stosunkowo łatwy do spełnienia, ale drugi wymaga trochę więcej przemyślenia. Rozważmy przykład `BangBean.java` przedstawiony w poprzednim rozdziale. Tam uchylamy się od kwestii wielowątkowości, ignorując słowo kluczowe `synchronized` (które wtedy jeszcze nie

zostało przedstawione) i pozwalając tylko na pojedynczego odbiorcę zdarzenia. Oto tamten przykład zmodyfikowany do pracy w środowisku wielowątkowym:

```
//: c14:BangBean2.java
// Powinieneś pisać swoje Beany w ten sposób, by
// mogły działać w środowisku wielowątkowym.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class BangBean2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Rozmiar okręgu
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.red;
  private ArrayList actionListeners =
    new ArrayList();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() {
    return cSize;
  }
  public synchronized void
  setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() {
    return text;
  }
  public synchronized void
  setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize() {
    return fontSize;
  }
  public synchronized void
  setFontSize(int newSize) {
    fontSize = newSize;
  }
  public synchronized Color getTextColor() {
    return tColor;
  }
  public synchronized void
  setTextColor(Color newColor) {
    tColor = newColor;
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
```

```
    g.drawOval(xm - cSize/2, ym - cSize/2,
      cSize, cSize);
  }
  // To jest wieloodbiorca zdarzeń, który jest
  // wykorzystywany bardziej typowo niż rozwiązanie
  // jednego odbiorcy przedstawione w BangBean.java:
  public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
  }
  public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
  }
  // Zwróć uwagę na brak synchronizacji:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Stwórz płytką kopię listy na wypadek,
    // gdyby ktoś dodał nowego odbiorcę,
    // kiedy my wywołujemy odbiorców:
    synchronized(this) {
      lv = (ArrayList)actionListeners.clone();
    }
    // Wywołaj wszystkie metody odbiorcy:
    for(int i = 0; i < lv.size(); i++)
      ((ActionListener)lv.get(i))
        .actionPerformed(a);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, fontSize));
      int width =
        g.getFontMetrics().stringWidth(text);
      g.drawString(text,
        (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      notifyListeners();
    }
  }

  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BangBean2 bb = new BangBean2();
    bb.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e){
          System.out.println("ActionEvent " + e);
        }
      });
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("Odbiorca BangBean2");
        }
      });
      bb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          System.out.println("Więcej odbiorców");
        }
      });
      Console.run(bb, 300, 300);
    }
} ///:~
```

Dodanie słowa `synchronized` do modyfikatorów metod jest prostą zmianą. Jednak zauważ, iż w metodach `addActionListener()` i `removeActionListener()` odbiorcy są teraz dodawani i usuwani z listy `ArrayList`, a zatem można ich mieć tylu, ilu tylko zechcemy.

Jak widać, metoda `notifyListeners()` *nie* jest synchronizowana. Można ją wywoływać z więcej niż jednego wątku jednocześnie. Jest również możliwe, by `addActionListener()` lub `removeActionListener()` były wywołane w czasie wywołania `notifyListeners()`, co jest kłopotliwe, ponieważ iteruje on po liście odbiorców — obiekcie `actionListeners` klasy `ArrayList`. Aby załagodzić sprawę, lista odbiorców jest klonowana wewnątrz klauzuli `synchronized` i przeglądany jest klon (zajrzyj do dodatku A po szczegóły na temat klonowania). Tym sposobem można modyfikować oryginalną listę `ArrayList` bez wpływu na informowanie odbiorców w `notifyListeners()`.

Metoda `paintComponent()` również nie jest synchronizowana. Decyzja o synchronizacji metod przesłoniętych nie jest tak łatwa, jak w przypadku własnych metod dodanych. W naszym przykładzie okazuje się, iż `paint()` zdaje się działać dobrze bez względu na to, czy jest synchronizowana, czy też nie. Ale kwestie, które trzeba brać pod uwagę, są następujące:

1. Czy metoda modyfikuje stan zmiennych „krytycznych" obiektu? Aby odkryć, czy dana zmienna jest „krytyczną", trzeba określić, czy będzie czytana lub ustawiana przez inne wątki programu (w tym przypadku odczyt i zmiana są praktycznie zawsze przeprowadzane poprzez metody `synchronized`, a zatem można po prostu to zbadać). W przypadku metody `paint()` żadna modyfikacja nie ma miejsca.
2. Czy metoda zależy od stanu tych zmiennych „krytycznych"? Jeśli jakaś metoda synchronizowana modyfikuje zmienną, którą nasza metoda wykorzystuje, to bardzo prawdopodobnie będziemy chcieć uczynić i naszą metodę `synchronized`. Wykorzystując to, można zaobserwować, iż `cSize` jest zmieniana przez metodę synchronizowaną i dlatego `paint()` też taką powinno być. Jednak można zapytać: „co najgorszego może się przytrafić, jeśli `cSize` zmieni się w trakcie działania `paint()`?". Kiedy przekonasz się, iż nie jest to nic strasznego i powoduje tylko przelotny efekt, to możesz zdecydować się na pozostawienie tej metody jako niesynchronizowanej, aby uniknąć dodatkowego narzutu związanego z wywoływaniem.

3. Trzecia wskazówka to zwrócenie uwagi na to, czy wersja metody `paint()` z klasy bazowej jest synchronizowana, co nie ma miejsca. Nie jest to żaden poważny argument, a zwykła wskazówka. W tym przypadku na przykład pole, które *jest* zmieniane przez metody synchronizowane (czyli `cSize`), zostało wmieszane w formułę `paint()` i mogłoby to zmienić sytuację. Zauważ jednak, iż synchronizacja nie jest dziedziczona — a zatem jeśli metoda jest deklarowana jako `synchronized` w klasie bazowej, to *nie jest* ona automatycznie synchronizowana w wersji przesłoniętej zamieszczonej w klasie potomnej.

Kod próbny w funkcji `main`, testujący `BangBean2` został zmodyfikowany w stosunku do tego z poprzedniego rozdziału, aby pokazać zdolność pracy `BangBean2` z wieloma odbiorcami.

Blokowanie

Każdy wątek może się znajdować w jednym z czterech stanów:

1. *Nowy*. Obiekt wątku został stworzony, ale wątek nie jest jeszcze uruchomiony, a więc nie może działać.
2. *Uruchamialny*. Oznacza, iż wątek *może* działać, kiedy tylko mechanizm podziału czasu udostępni mu procesor. Tak więc wątek może być lub też nie być uruchomiony, ale nie ma nic, co by go powstrzymywało, jeśli zarządca może zorganizować dostęp do procesora; nie jest to stan śmierci ani blokady.
3. *Uśmiercony*. Normalny sposób śmierci wątku to zakończenie jego metody `run()`. Można także wywołać metodę `stop()`, ale wtedy wyrzucany jest wyjątek będący podklasą klasy `Error` (co oznacza, iż nie trzeba zamieszczać tego wywołania w bloku `try`). Pamiętajmy, iż zgłoszenie wyjątku powinno być zdarzeniem specjalnym, a nie częścią normalnego wykonania programu; stąd użycie `stop()`nie jest w Java 2 zalecane. Istnieje także metoda `destroy()` (nigdy nie została zaimplementowana), której nigdy nie powinno się wywoływać, jeśli można tego uniknąć, gdyż jest drastyczna i nie zwalnia blokady obiektu.
4. *Zablokowany*. Wątek mógłby działać, ale jest coś, co temu przeszkadza. Gdy wątek jest w stanie zablokowania, to zarządca podziału czasu po prostu będzie go pomijał i nie da mu żadnego dostępu do procesora. Dopóki wątek ponownie nie wejdzie w stan uruchomienia, to nie będzie mógł wykonać żadnej operacji.

Zablokowanie

Stan zablokowania jest najbardziej interesującym i wartym szerszego omówienia. Wątek może zostać zablokowany z pięciu powodów:

1. „Położyliśmy wątek spać", wywołując `sleep(milisekundy)`, a wtedy nie będzie działał przez określony czas.
2. Zawiesiliśmy wykonanie wątku poprzez wywołanie metody `suspend()`. Nie będzie uruchamialny, aż nie uzyska komunikatu `resume()` (te metody nie są zalecane w Java 2 i przeanalizujemy je dalej).

3. Zawiesiliśmy wykonanie wątku metodą `wait()`. Nie będzie uruchamialny ponownie, dopóki nie uzyska komunikatu `notify()` albo `notifyAll()` (tak, to wygląda jak punkt 2., ale jest tu wyraźna różnica, którą pokażę dalej).
4. Wątek oczekuje na zakończenie jakieś operacji obsługi wejścia-wyjścia.
5. Wątek usiłuje wywołać metodę synchronizowana na innym obiekcie, który jest właśnie zablokowany.

Można również wywołać metodę `yield()` (metoda z klasy `Thread`), aby dobrowolnie zrezygnować z procesora, a więc pozwolić innym wątkom na działanie. Jednak to samo zachodzi, jeśli zarządca zdecyduje, że nasz wątek miał wystarczająco czasu i przekaże pracę do innego wątku. Nic nie powstrzyma zarządcy przed przeniesieniem naszego wątku i oddaniem czasu jakiemuś innemu. Kiedy wątek jest zablokowany, to są powody, by nie mógł kontynuować działania.

Kolejny przykład pokazuje wszystkie pięć sposobów zablokowania. Wszystkie zostały zamieszczone w pojedynczym pliku o nazwie `Blocking.java`, ale zostaną przebadane w oddzielnych fragmentach (pojawią się znaczniki „Continued" i „Continuing", pozwalające mojemu narzędziu do wydobywania kodu na połączenie kawałków).

Ponieważ ten przykład zawiera pewne metody przestarzałe i nie zalecane, to podczas kompilacji pojawią się komunikaty „deprecated".

Najpierw podstawowy szkielet:

```
//: c14:Blocking.java
// Pokaz różnych sposobów
// blokowania wątku.
// <applet code=Blocking width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

////////////// Podstawowy szkielet ///////////
class Blockable extends Thread {
  private Peeker peeker;
  protected JTextField state = new JTextField(30);
  protected int i;
  public Blockable(Container c) {
    c.add(state);
    peeker = new Peeker(this, c);
  }
  public synchronized int read() { return i; }
  protected synchronized void update() {
    state.setText(getClass().getName()
      + " stan: i = " + i);
  }
  public void stopPeeker() {
    // peeker.stop(); Nie zalecana w Java 1.2
    peeker.terminate(); // Preferowane podejście
  }
}
```

```
class Peeker extends Thread {
  private Blockable b;
  private int session;
  private JTextField status = new JTextField(30);
  private boolean stop = false;
  public Peeker(Blockable b, Container c) {
    c.add(status);
    this.b = b;
    start();
  }
  public void terminate() { stop = true; }
  public void run() {
    while (!stop) {
      status.setText(b.getClass().getName()
        + " Peeker " + (++session)
        + "; wartość = " + b.read());
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
} ///:Continued
```

Klasa `Blockable` ma być klasą bazową dla wszystkich klas tego przykładu. Obiekt `Blockable` zawiera pole `JTextField` o nazwie `state`, które jest wykorzystywane do wyświetlania informacji o obiekcie. Metodą, która wyświetla te wiadomości, jest `update()`. Jak widać, używa ona `getClass().getName()` do uzyskania nazwy klasy, zamiast po prostu ją wypisywać — jest tak, gdyż `update()` nie może znać dokładnej nazwy klasy, wobec której zostanie wykonana, jako że będzie to klasa dziedzicząca z `Blockable`.

Wskaźnikiem zmian w `Blockable` jest zmienna `int i`, która będzie inkrementowana przez metodę `run()` klasy potomnej.

Jest tu wątek klasy `Peeker`, który jest uruchamiany dla każdego obiektu `Blockable`. Jego zadaniem jest obserwowanie skojarzonego obiektu `Blockable`, aby zauważyć zmianę wartości `i`, i wywołując `read()`, informować o niej w polu `status JTextField`. To jest ważne: obie metody `read()` i `update()` są synchronizowane, co oznacza, iż wymagają, by blokada obiektu była wolna.

Uśpienie

Pierwszy sprawdzian w naszym programie dotyczy metody `sleep()`:

```
///:Continuing
////////////// Blokowanie poprzez sleep() ///////////
class Sleeper1 extends Blockable {
  public Sleeper1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        sleep(1000);
```

```
        } catch(InterruptedException e) {
          System.err.println("Przerwany");
        }
      }
    }
  }

  class Sleeper2 extends Blockable {
    public Sleeper2(Container c) { super(c); }
    public void run() {
      while(true) {
        change();
         try {
          sleep(1000);
        } catch(InterruptedException e) {
          System.err.println("Przerwany");
        }
      }
    }
    public synchronized void change() {
        i++;
        update();
    }
  } ///:Continued
```

W klasie `Sleeper1` synchronizowana jest cała metoda `run()`. Zobaczymy, iż `Peeker` skojarzony z tym obiektem będzie wesoło działał, dopóki nie wystartujemy wątku i wtedy `Peeker` nagle „zamarznie". Jest to jedna z form blokowania: ponieważ `Sleeper1.run()` jest synchronizowana i po tym jak wątek wystartuje, jest on zawsze wewnątrz `run()`, metoda ta nigdy nie oddaje blokady obiektu i `Peeker` jest zablokowany.

`Sleeper2` dostarcza więc rozwiązanie — usuwa synchronizację z metody `run()`. Tylko metoda `change()` jest synchronizowana, stąd kiedy `run()` jest w `sleep()`, to `Peeker` może sięgać do dowolnej metody synchronizowanej, jakiej potrzebuje. Teraz widać, iż `Peeker` kontynuuje działanie po wystartowaniu wątku `Sleeper2`.

Zawieszenie i wznowienie

Następna część naszego przykładu wprowadza pojęcie zawieszenia działania. Klasa `Thread` zawiera bowiem metodę `suspend()` do tymczasowego zatrzymania wątku oraz `resume()`, która ponownie startuje z miejsca, w którym został zatrzymany. Metoda `resume()` musi być wywołana przez jakiś wątek zewnętrzny wobec tego zawieszanego — rozwiązuje to oddzielna klasa `Resumer`. Każda z klas pokazujących zawieszenie lub wznowienie ma skojarzony obiekt klasy `Reasumer`:

```
  ///:Continuing
  /////////// Blokowanie poprzez suspend() ///////////
  class SuspendResume extends Blockable {
    public SuspendResume(Container c) {
      super(c);
      new Resumer(this);
    }
  }
```

```
class SuspendResume1 extends SuspendResume {
  public SuspendResume1(Container c) { super(c):}
  public synchronized void run() {
    while(true) {
      i++:
      update():
      suspend(): // Nie zalecana w Java 1.2
    }
  }
}

class SuspendResume2 extends SuspendResume {
  public SuspendResume2(Container c) { super(c):}
  public void run() {
    while(true) {
      change():
      suspend(): // Nie zalecana w Java 1.2
    }
  }
  public synchronized void change() {
      i++:
      update():
  }
}

class Resumer extends Thread {
  private SuspendResume sr:
  public Resumer(SuspendResume sr) {
    this.sr = sr:
    start():
  }
  public void run() {
    while(true) {
       try {
        sleep(1000):
      } catch(InterruptedException e) {
        System.err.println("Przerwany"):
      }
      sr.resume(): // Nie zalecana w Java 1.2
    }
  }
} ///:Continued
```

Klasa `SuspendResume1` **także posiada metodę** `synchronized run()`. **Powtórzę — jeśli wystartujesz ten wątek, zobaczysz, iż skojarzony** `Peeker` **zostanie zablokowany w oczekiwaniu na zwolnienie blokady, co nigdy nie nastąpi. Zostało to poprawione w** `SuspendResume2`**, która nie synchronizuje całej metody** `run()`**, ale zamiast tego stosuje oddzielną metodę** `synchronized change()`.

Powinieneś pamiętać, iż w Java 2 stosowanie metod `suspend()` **i** `resume()` `nie jest zalecane`**, ponieważ przetrzymują obiekt i tym samym mogą prowadzić do impasu (ang. deadlock). Innymi słowy, można łatwo uzyskać wiele obiektów zablokowanych, oczekujących na siebie wzajemnie, i to powoduje, że program zostaje „zamrożony". Chociaż możesz zaobserwować wykorzystanie tych metod w starszych programach, to nie powinieneś tego robić. Właściwe rozwiązanie jest opisane nieco dalej w tym rozdziale.**

Wait i notify

W dwóch pierwszych przykładach ważne jest, aby zrozumieć, iż obie metody `sleep()` i `suspend()` *nie* zwalniają blokady po wywołaniu. Trzeba o tym pamiętać podczas pracy z użyciem synchronizacji. Z kolei metoda `wait()` *zwalnia* blokadę po wywołaniu, co oznacza, iż inne metody synchronizowane w obiekcie wątku mogą być wywoływane podczas oczekiwania. W kolejnych dwóch klasach zobaczymy, że `run()` będzie w pełni synchronizowana, a jednak `Peeker` nadal będzie miał dostęp do metod synchronizowanych podczas oczekiwania `wait()`. Jest tak, ponieważ `wait()` zwalnia blokadę obiektu podczas zawieszenia metody, z której jest wywołana.

Mamy dwie postacie `wait()`. Pierwsza pobiera argument w milisekundach o tym samym znaczeniu, co w przypadku metody `sleep()` — wstrzymanie na wskazany okres czasu. Różnica polega na tym, iż w `wait()` blokada obiektu jest zwalniana *i* można wyjść z `wait()` poprzez `notify()` albo zakończenie odliczania zegara.

Druga postać nie pobiera parametrów, a jej wywołanie powoduje, iż oczekiwanie będzie trwać, dopóki nie pojawi się `notify()` i tego nie zakończy.

Unikatowym aspektem metod `wait()` i `notify()` jest to, że obie pochodzą z klasy bazowej `Object`, a nie są częścią `Thread`, tak jak: `sleep()`, `suspend()` i `resume()`. Chociaż wydaje się to początkowo trochę dziwne — posiadanie czegoś, co jest wyłącznie dla wątków jako część uniwersalnej klasy bazowej — to jest niezmiernie istotne, ponieważ operują one blokadą, która także jest częścią każdego obiektu. W rezultacie można zamieścić `wait()` wewnątrz dowolnej metody synchronizowanej, bez względu na to, czy jakiekolwiek motywy wielowątkowości pojawiają się wewnątrz tej konkretnej klasy. Faktycznie *jedynym* miejscem, gdzie można wywołać `wait()`, jest wnętrze metody synchronizowanej albo synchronizowanego bloku. Jeśli wywoła się `wait()` albo `notify()` w metodzie nie będącej synchronizowaną, to program się skompiluje, ale podczas uruchomienia pojawi się wyjątek `IllegalMonitorStateException` z trochę nieintuicyjnym komunikatem „current thread not owner". Zauważ, iż wszystkie metody: `sleep()`, `suspend()` oraz `resume()` mogą być wywołane z metod niesynchronizowanych, gdyż nie manipulują blokadą obiektu.

Metody `wait()` i `notify()` mogą być wywołane tylko dla swojej własnej blokady. Jeszcze raz powtórzę — można skompilować kod usiłujący wykorzystać niewłaściwą blokadę, ale pojawi się ten sam komunikat `IllegalMonitorStateException`, co poprzednio. Nie można dostać się do blokady należącej do kogoś innego, ale zawsze można poprosić taki obiekt o wykonanie operacji manipulującej jego własną blokadą. Zatem jednym z rozwiązań jest stworzenie metody `synchronized`, która wywołuje `notify()` dla swego własnego obiektu. W klasie `Notifier` widać wywołanie `notify()` wewnątrz bloku `synchronized`:

```
synchronized(wn2) {
  wn2.notify();
}
```

gdzie `wn2` jest obiektem typu `WaitNotify2`. Metoda ta, nie będąca częścią `WaitNotify2`, nabywa dostęp do blokady obiektu `wn2` i staje się legalnym wywołanie `notify()` dla `wn2` — wyjątek `IllegalMonitorStateException` się nie pojawi.

```
///:Continuing
////////////// Blokowanie poprzez wait() //////////////
class WaitNotify1 extends Blockable {
  public WaitNotify1(Container c) { super(c); }
  public synchronized void run() {
```

```
    while(true) {
      i++;
      update();
       try {
        wait(1000);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
}

class WaitNotify2 extends Blockable {
  public WaitNotify2(Container c) {
    super(c);
    new Notifier(this);
  }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait();
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
}

class Notifier extends Thread {
  private WaitNotify2 wn2;
  public Notifier(WaitNotify2 wn2) {
    this.wn2 = wn2;
    start();
  }
  public void run() {
    while(true) {
       try {
        sleep(2000);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
      synchronized(wn2) {
        wn2.notify();
      }
    }
  }
} ///:Continued
```

Zazwyczaj stosuje się `wait()`, gdy dojdziemy do miejsca, w którym musimy poczekać na zmianę jakiś innych warunków, zależnych od sił zewnętrznych, i chcemy po prostu bezczynnie czekać wewnątrz wątku. Zatem `wait()` pozwala na uśpienie obiektów podczas oczekiwania na zmianę warunków zewnętrznych i tylko kiedy wystąpi `notify()` albo `notifyAll()`, wątek obudzi się i sprawdzi zmiany. Tak więc zapewnia to sposób na synchronizację pomiędzy wątkami.

Blokowanie na operacjach we-wy

Jeśli strumień oczekuje na pewna aktywność wejścia-wyjścia, to zostanie zablokowany automatycznie. W następnej części naszego przykładu dwie klasy pracują z obiektami `Reader` **i** `Writer`**, ale w szkielecie testowym zostanie ustawiony potok, aby pozwolić tym dwóm wątkom na bezpieczne przekazanie danych między sobą (co jest celem potoku).**

`Sender` **zamieszcza dane w obiekcie** `Writer` **i zostaje uśpiony na losowo wybrany okres czasu. Jednak odbiorca** `Receiver` **nie zawiera** `sleep()`, `suspend()` **ani** `wait()`**. Jednak podczas odczytu, kiedy wykonuje** `read()`**, jest automatycznie blokowany, jeśli musi czekać na dane.**

```
///:Continuing
class Sender extends Blockable { // wysyła
  private Writer out;
  public Sender(Container c, Writer out) {
    super(c);
    this.out = out;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          out.write(c);
          state.setText("Sender wysyła: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Przerwany");
        } catch(IOException e) {
          System.err.println("Problem wej/wyj");
        }
      }
    }
  }
}

class Receiver extends Blockable {
  private Reader in;
  public Receiver(Container c, Reader in) {
    super(c);
    this.in = in;
  }
  public void run() {
    try {
      while(true) {
        i++; // Pokazanie, że nadal żyje
        // Blokowanie aż pojawią się znaki:
        state.setText("Receiver odczytał: "
          + (char)in.read());
      }
    } catch(IOException e) {
      System.err.println("Problem wej/wyj");
    }
  }
} ///:Continued
```

Obie klasy zamieszczają informację w ich polach `state` i zmieniają wartość `i` tak, by `Peeker` mógł zobaczyć, że wątek działa.

Testowanie

Główna klasa apletu jest zdumiewająco prosta, ponieważ większość pracy została umieszczona w szkielecie `Blockable`. Tworzona jest tablica obiektów `Blockable` i ponieważ każdy jest wątkiem, to wykonuje swoje własne zajęcia od momentu wciśnięcia przycisku „Start". Są także przyciski i klauzula `actionPerformed()` do zatrzymania wszystkich obiektów `Peeker`, które w swojej konstrukcji pokazują alternatywę do nie zalecanej (w Java 2) metody `stop()` klasy `Thread`.

Aby ustanowić połączenie pomiędzy obiektami `Sender` i `Receiver`, tworzone są `PipedWriter` i `PipedReader`. `PipedReader in` musi być połączony z `PipedWriter out` poprzez argumenty konstruktora. Po tym cokolwiek, co jest zamieszczone w `out`, może być następnie wydobyte z `in`, jakby było przesyłane przez potok (stąd nazwa). Obiekty `in` i `out` są przekazywane do konstruktorów klas `Receiver` i `Sender`, które odpowiednio traktują je jak obiekty `Reader` i `Writer` dowolnego typu (to znaczy, że są one rzutowane w górę).

Tablica `b` odwołań `Blockable` nie jest inicjowana w miejscu swej definicji, ponieważ strumienie potokowe nie mogą być ustawione przed wystąpieniem tej definicji (potrzeba istnienia bloku `try` na to nie pozwala).

```
///:Continuing
////////// Testowanie wszystkiego //////////
public class Blocking extends JApplet {
  private JButton
    start = new JButton("Start"),
    stopPeekers = new JButton("Zatrzymaj wątki sprawdzające Peeker");
  private boolean started = false;
  private Blockable[] b;
  private PipedWriter out;
  private PipedReader in;
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Demonstracja preferowanej
      // alternatywy dla Thread.stop();
      for(int i = 0; i < b.length; i++)
        b[i].stopPeeker();
    }
  }
  public void init() {
     Container cp = getContentPane();
     cp.setLayout(new FlowLayout());
     out = new PipedWriter();
    try {
```

```
        in = new PipedReader(out);
      } catch(IOException e) {
        System.err.println("Problem z PipedReader");
      }
      b = new Blockable[] {
        new Sleeper1(cp),
        new Sleeper2(cp),
        new SuspendResume1(cp),
        new SuspendResume2(cp),
        new WaitNotify1(cp),
        new WaitNotify2(cp),
        new Sender(cp, out),
        new Receiver(cp, in)
      };
      start.addActionListener(new StartL());
      cp.add(start);
      stopPeekers.addActionListener(
        new StopPeekersL());
      cp.add(stopPeekers);
    }
    public static void main(String[] args) {
      Console.run(new Blocking(), 350, 550);
    }
  } ///:~
```

W metodzie `init()` mamy pętlę przechodzącą przez całą tablicę i dodającą stan `state` oraz pole tekstowe `peeker.status` na stronę.

Kiedy tworzone są wątki `Blockable`, to każdy z nich automatycznie tworzy i uruchamia swój własny `Peeker`. Zatem wątki `Peeker` będą działać, zanim wystartuje wątek `Blockable`. Jest to istotne, ponieważ niektóre z obiektów `Peeker` zostaną zablokowane i zatrzymane, kiedy wątki `Blockable` wystartują. Konieczne trzeba to zobaczyć, aby zrozumieć ten szczególny aspekt blokowania.

Impas

Ponieważ wątki mogą być blokowane *i* ponieważ obiekty mogą posiadać metody synchronizowane, nie pozwalające wątkom na dostęp do tych obiektów, dopóki nie zostanie zwolniony zamek synchronizacji, to jest możliwe, iż wątek utknie, oczekując na inny wątek, który czeka na inny wątek itd., aż łańcuch powróci do wątku czekającego na ten pierwszy. Dostajemy zamkniętą pętlę wątków, czekających na siebie wzajemnie, a żaden nie może się ruszyć — nazywa się to *impasem* (ang. *deadlock*). Nadzieja, że taka trudna sytuacja zdarza się bardzo rzadko, nie pomaga, kiedy przyjdzie nam testować i wyszukiwać błędy w kodzie, który wywołał impas.

Nie mamy wsparcia ze strony języka pomagającego unikać impasu — wszystko zależy od nas, czy unikniemy go przez ostrożne projektowanie. Nie są to słowa pocieszające dla osoby, która usiłuje usuwać błędy w programie, w którym wystąpił impas.

Nie zalecane metody: stop(), suspend(), resume() i destroy() w Java 2

Jedną ze zmian, dokonaną w Java 2, aby zmniejszyć możliwość impasu, jest niezalecanie następujących metod klasy Thread: `stop()`, `suspend()`, `resume()` oraz `destroy()`.

Przyczyną tego, iż metoda `stop()` stała się nie zalecana jest fakt, że nie zwalnia blokad, które wątek posiada, i jeżeli obiekty są w stanie niespójnym („uszkodzone"), inne wątki mogą je w takim stanie podglądać i modyfikować. Wynikające problemy mogą być subtelne i trudne do wykrycia. Zamiast używać `stop()`, powinieneś stosować się do tego, co zostało pokazane w przykładzie `Blocking.java` i używać znaczników do informowania wątku, kiedy ma zakończyć się przez wyjście ze swej metody `run()`.

Są sytuacje, kiedy wątek się blokuje — tak jak wtedy, gdy oczekuje na dane wejściowe — i nie może sprawdzić znacznika, jak było to w `Blocking.java`. W takich przypadkach nadal nie powinno się używać `stop()`, a zamiast tego można użyć metody `interrupt()` z klasy `Thread`, aby wydostać się z zablokowanego kodu:

```
//: c14:Interrupt.java
// Alternatywne podejście do stosowania
// metody stop() kiedy wątek jest zablokowany.
// <applet code=Interrupt width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Blocked extends Thread {
  public synchronized void run() {
    try {
      wait(); // Blokowanie
    } catch(InterruptedException e) {
      System.err.println("Przerwany");
    }
    System.out.println("Wyjście z run()");
  }
}

public class Interrupt extends JApplet {
  private JButton
    interrupt = new JButton("Przerwij");
  private Blocked blocked = new Blocked();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrupt);
    interrupt.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Przycisk wciśnięty");
          if(blocked == null) return;
          Thread remove = blocked;
          blocked = null; // aby go zwolnić
          remove.interrupt();
        }
      });
    blocked.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrupt(), 200, 100);
  }
} ///:~
```

Wywołanie `wait()` wewnątrz `Blocked.run()` powoduje zablokowanie wątku. Po wciśnięciu przycisku referencja `blocked` jest ustawiana na `null`, a zatem odśmiecacz pamięci ją sprzątnie i zostaje wywołana metoda `interrupt()` tego obiektu. Przy pierwszym wciśnięciu przycisku wątek zostanie zakończony, ale po tym nie będzie już żadnego wątku do zabicia.

Metody `suspend()` i `resume()` mogą łatwo spowodować impas. Po wywołaniu `suspend()` wątek zostanie zatrzymany, ale ciągle będzie przetrzymywał jakieś blokady, które nabył do tej pory. Tak więc żaden inny wątek nie może dostać się do zamkniętych zasobów, dopóki zawieszony wątek nie wznowi pracy. Każdy wątek, usiłujący wznowić wątek zawieszony i próbujący wykorzystać któryś z zablokowanych zasobów, powoduje impas. Nie powinno się stosować `suspend()` i `resume()`, ale zamiast tego dodać do swojej klasy `Thread` znacznik, aby sygnalizował, czy wątek powinien być aktywny, czy zawieszony. Jeśli znacznik wskazywałaby na zawieszenie wątku, to powinien on przejść do stanu oczekiwania poprzez `wait()`. Jeśli zaś wskazywałby na odwieszenie, to wątek byłby wznawiany poprzez `notify()`. Zmodyfikujmy `Counter2.java`. Chociaż wynik jest podobny, to organizacja kodu jest inna — wewnętrzne klasy anonimowe są używane dla wszystkich odbiorców, podobnie też `Thread` jest klasą wewnętrzną, co czyni programowanie odrobinę wygodniejszym, eliminując pewien dodatkowy kod w `Counter2.java`:

```
//: c14:Suspend.java
// Alternatywne podejście do stosowania metod suspend()
// i resume(), które nie są zalecane w Java 2.
// <applet code=Suspend width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspend extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspend = new JButton("Zawieś"),
    resume = new JButton("Wznów");
  private Suspendable ss = new Suspendable();
  class Suspendable extends Thread {
    private int count = 0;
    private boolean suspended = false;
    public Suspendable() { start(); }
    public void fauxSuspend() {
      suspended = true;
    }
    public synchronized void fauxResume() {
      suspended = false;
      notify();
    }
    public void run() {
      while (true) {
        try {
          sleep(100);
          synchronized(this) {
            while(suspended)
              wait();
          }
        } catch(InterruptedException e) {
          System.err.println("Przerwany");
```

```
          }
          t.setText(Integer.toString(count++));
        }
      }
    }
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(t);
      suspend.addActionListener(
        new ActionListener() {
          public
          void actionPerformed(ActionEvent e) {
            ss.fauxSuspend();
          }
        });
      cp.add(suspend);
      resume.addActionListener(
        new ActionListener() {
          public
          void actionPerformed(ActionEvent e) {
            ss.fauxResume();
          }
        });
      cp.add(resume);
    }
    public static void main(String[] args) {
      Console.run(new Suspend(), 300, 100);
    }
} ///:~
```

Znacznik `suspended` wewnątrz `Suspendable` jest wykorzystywany do przełączania zawieszenia na włączone i nie. Aby zawiesić wątek, znacznik jest ustawiany na `true` poprzez wywołanie `fauxSuspend()`, co jest wykrywane wewnątrz `run()`. Metoda `wait()`, jak to zostało opisane wcześniej w tym rozdziale, musi zostać synchronizowana, a więc ma już blokadę obiektu. W `fauxResume()` następuje ustawianie `suspended` na `false` i wywoływanie metody `notify()` — ponieważ to pobudza `wait()` wewnątrz klauzuli `synchronized`, to `fauxResume()` musi również być synchronizowana, by przejąć blokadę przed wywołaniem `notify()` (zatem blokada jest dostępna dla `wait()`, aby ją wzbudzić). Jeżeli będziesz naśladował styl pokazany w tym programie, to uda Ci się uniknąć stosowania metod `suspend()` i `resume()`.

Metoda `destroy()` klasy `Thread` nigdy nie została zaimplementowana; jest jak `suspend()`, która nie może zlikwidować blokady, zatem występuje ta sama kwestia impasu co przy `suspend()`. Jednak nie jest to metoda nie zalecana i może być zaimplementowana w przyszłej wersji Javy (po 2) dla sytuacji specjalnych, w których ryzyko impasu jest akceptowalne.

Możesz się dziwić, czemu te metody, aktualnie nie zalecane, były włączone od początku do Javy. Bezapelacyjne ich usunięcie wyglądałoby jak przyznanie się do raczej znaczącego błędu (i pokazało kolejną dziurę w argumentacji o wyjątkowości projektu Javy i nieomylności przedstawianej przez ludzi od marketingu z Suna). Zasadniczą sprawą tej zmiany jest to, iż wyraźnie wskazuje, że kierują tym ludzie od spraw technicznych, a nie od marketingu — odkryli problem i go naprawiają. Uważam to za dużo bardziej obiecujące i budzące nadzieję niż pozostawienie takiego problemu, ponieważ „naprawa przyznaje o błędzie". Oznacza to, że Java będzie nadal się rozwijać, nawet jeśli powoduje to pewien dyskomfort ze strony programistów. Raczej wolałbym zgodzić się z tymi niedogodnościami niż patrzeć na stagnację języka.

Priorytety

Priorytet wątku mówi zarządcy podziału czasu procesora, jak ważny jest dany wątek. Jeśli jest wiele wątków zablokowanych i oczekujących na uruchomienie, to zarządca najpierw uruchomi ten z najwyższym priorytetem. Nie oznacza to jednak, iż wątki o niższych priorytetach nie będą działać (czyli nie można uzyskać impasu z powodu priorytetów). Wątki z niższymi priorytetami po prostu mają zwyczaj działać rzadziej. Chociaż priorytety są ciekawe do poznania i do zabawy, to w praktyce prawie nigdy nie będziesz potrzebował ich samemu ustawiać. Zatem spokojnie możesz przeskoczyć resztę tego podrozdziału, jeżeli priorytety nie są dla Ciebie niczym interesującym.

Odczyt i ustawienie priorytetów

Priorytet wątku można odczytać metodą `getPriority()` i zmienić metodą `setPriority()`. Wykorzystajmy uprzednie przykłady „liczników", aby pokazać efekt zmiany priorytetów. W niniejszym aplecie będzie widoczne spowolnienie liczników, w miarę jak skojarzone wątki będą miały mniejsze priorytety:

```
//: c14:Counter5.java
// Regulacja priorytetów wątków.
// <applet code=Counter5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Ticker2 extends Thread {
  private JButton
    b = new JButton("Przełącz"),
    incPriority = new JButton("zwiększ"),
    decPriority = new JButton("zmniejsz");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Pokaż priorytet
  private int count = 0;
  private boolean runFlag = true;
  public Ticker2(Container c) {
    b.addActionListener(new ToggleL());
    incPriority.addActionListener(new UpL());
    decPriority.addActionListener(new DownL());
    JPanel p = new JPanel();
    p.add(t);
    p.add(pr);
    p.add(b);
    p.add(incPriority);
    p.add(decPriority);
    c.add(p);
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  class UpL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() + 1;
      if(newPriority > Thread.MAX_PRIORITY)
        newPriority = Thread.MAX_PRIORITY;
      setPriority(newPriority);
    }
  }
  class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() - 1;
      if(newPriority < Thread.MIN_PRIORITY)
        newPriority = Thread.MIN_PRIORITY;
      setPriority(newPriority);
    }
  }
  public void run() {
    while (true) {
      if(runFlag) {
        t.setText(Integer.toString(count++));
        pr.setText(
          Integer.toString(getPriority()));
      }
      yield();
    }
  }
}

public class Counter5 extends JApplet {
  private JButton
    start = new JButton("Start"),
    upMax = new JButton("Zwiększ Max Priorytet"),
    downMax = new JButton("Zmniejsz Max Priorytet");
  private boolean started = false;
  private static final int SIZE = 10;
  private Ticker2[] s = new Ticker2[SIZE];
  private JTextField mp = new JTextField(3);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new Ticker2(cp);
    cp.add(new JLabel(
      "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
    cp.add(new JLabel("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    cp.add(new JLabel("Max Priorytet Grupy = "));
    cp.add(mp);
    cp.add(start);
    cp.add(upMax);
    cp.add(downMax);
    start.addActionListener(new StartL());
    upMax.addActionListener(new UpMaxL());
    downMax.addActionListener(new DownMaxL());
    showMaxPriority();
    // Rekursywnie pokaż dla rodziców grupy wątków:
    ThreadGroup parent =
      s[0].getThreadGroup().getParent();
    while(parent != null) {
      cp.add(new Label(
```

```
          "Max priorytet rodzica grupy wątków = "
          + parent.getMaxPriority()));
        parent = parent.getParent();
      }
    }
    public void showMaxPriority() {
      mp.setText(Integer.toString(
        s[0].getThreadGroup().getMaxPriority()));
    }
    class StartL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(!started) {
          started = true;
          for(int i = 0; i < s.length; i++)
            s[i].start();
        }
      }
    }
    class UpMaxL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int maxp =
          s[0].getThreadGroup().getMaxPriority();
        if(++maxp > Thread.MAX_PRIORITY)
          maxp = Thread.MAX_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
      }
    }
    class DownMaxL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        int maxp =
          s[0].getThreadGroup().getMaxPriority();
        if(--maxp < Thread.MIN_PRIORITY)
          maxp = Thread.MIN_PRIORITY;
        s[0].getThreadGroup().setMaxPriority(maxp);
        showMaxPriority();
      }
    }
    public static void main(String[] args) {
      Console.run(new Counter5(), 450, 600);
    }
  } ///:~
```

`Ticker2` naśladuje postać wcześniej prezentowaną w tym rozdziale, ale dodano tu pole `JTextField` do wyświetlania priorytetu wątku i dwa przyciski do zwiększenia i zmniejszenia priorytetu o jeden.

Wykorzystaliśmy także metodę `yield()`, która przekazuje kontrolę z powrotem do zarządcy. Bez tego mechanizm wielowątkowości nadal działa, ale jak się przekonasz, działa wolniej (spróbuj usunąć wywołanie `yield()`, aby się przekonać). Możesz także wywołać `sleep()`, ale wtedy tempo zliczania będzie sterowane przez czas uśpienia zamiast przez priorytet.

Metoda `init()` w klasie `Counter5` tworzy tablicę dziesięciu obiektów `Ticker2`; ich przyciski i pola zamieszczane są na formatce przez konstruktor `Ticker2`. `Counter5` dodaje przyciski do wystartowania wszystkiego oraz inkrementacji i dekrementacji maksymalnego priorytetu grupy wątków. Są także etykiety prezentujące maksymalne i minimalne priorytety możliwe dla wątku i pole `JTextField`, pokazujące maksymalny priorytet grupy wątku (grupy wątków zostaną przedstawione w następnym podrozdziale). Priorytety nadrzędnych grup wątków też są wyświetlane jako etykiety.

Po wciśnięciu przycisku „zwiększ" lub „zmniejsz" priorytet `Ticker2` jest odpowiednio inkrementowany lub dekrementowany.

Podczas uruchomieniu programu zauważysz kilka rzeczy. Przede wszystkim priorytet domyślny dla wątku grupy ma wartość pięć. Nawet jeśli zmniejszysz maksymalny priorytet poniżej piątki przed uruchomieniem wątku (albo przed stworzeniem wątków, co wymaga zmiany w kodzie), to każdy wątek będzie nadal miał domyślny priorytet pięć.

Prosta próba może polegać na zmniejszeniu priorytetu jednego z liczników do jedynki i obserwowaniu, iż zlicza znacznie wolniej. Spróbuj zwiększyć go ponownie. Można przywrócić wartość do wielkości priorytetu wątku grupy, ale nie więcej. Teraz zmniejsz parokrotnie priorytet grupy wątków. Priorytety wątku nie zmienią się, ale jeśli spróbujesz je zmodyfikować w górę bądź w dół, to zobaczysz, iż automatycznie skoczą na wartość priorytetu grupy wątku. Również nowe wątki będą nadal dostawać domyślny priorytet, nawet jeśli będzie większy niż priorytet grupy (zatem priorytet grupy nie jest metodą na zapobieganie przed przyznaniem nowym wątkom wyższych priorytetów niż istniejące).

Spróbuj jeszcze zwiększyć maksymalny priorytet grupy. Nie można tego zrobić. Daje się tylko zredukować maksymalne priorytety grupy wątku, ale nie zwiększyć.

Grupy wątków

Wszystkie wątki należą do jakiejś a grupy. Może to być albo domyślna grupa wątku, albo grupa, którą się określi podczas tworzenia wątku. Podczas tworzenia wątek zostaje przywiązany do jakiejś grupy i później nie może się przepiąć do innej. Każda aplikacja posiada przynajmniej jeden wątek należący do systemowej grupy wątków. Jeśli utworzy się więcej wątków bez określania grupy, to one również będą należeć do grupy systemowej.

Grupy wątków również muszą należeć do jakichś innych grup. Grupa wątku, do której jakaś nowa grupa należy, powinna być określona w konstruktorze. Jeśli stworzysz grupę wątku bez określenia grupy nadrzędnej, to zostanie zamieszczona w grupie systemowej. Tak więc wszystkie grupy wątków w aplikacji będą ostatecznie posiadać systemową grupę wątku jako rodzica.

Powód istnienia grup wątków jest trudny do zrozumienia na podstawie literatury, która zazwyczaj jeszcze bardziej gmatwa ten temat. Często jest wymieniany jako „względy bezpieczeństwa". Według Arnolda i Goslinga[4]: „wątki w grupie wątków mogą modyfikować inne wątki w tej grupie, włączając w to te dalej w hierarchii. Wątek nie może modyfikować wątków spoza swojej grupy albo z grup podległych". Trudno uwierzyć, iż „modyfikować" może tu mieć znaczenie dosłownie. Zamieszczony poniżej przykład pokazuje, iż wątek w podgrupie będącej „liściem" zmienia priorytety wszystkich wątków w swoim drzewie grup oraz wywołuje metody dla wszystkich tych wątków w jej drzewie.

```
//: c14:TestAccess.java
// Jak wątki mogą uzyskać dostęp do innych wątków
// z grupy wątku rodzica.

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

[4] *The Java Programming Language* — Ken Arnold i James Gosling (Addison-Wesley 1996).

```
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      one = new TestThread1(x, "jeden"),
      two = new TestThread2(z, "dwa");
  }
}

class TestThread1 extends Thread {
  private int i;
  TestThread1(ThreadGroup g, String name) {
    super(g, name);
  }
  void f() {
    i++; // modyfikacja wątku
    System.out.println(getName() + " f()");
  }
}

class TestThread2 extends TestThread1 {
  TestThread2(ThreadGroup g, String name) {
    super(g, name);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gAll = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gAll.length; i++) {
      gAll[i].setPriority(Thread.MIN_PRIORITY);
      ((TestThread1)gAll[i]).f();
    }
    g.list();
  }
} ///:~
```

W metodzie `main()` tworzonych jest kilka obiektów `ThreadGroup` wywiedzionych z siebie wzajemnie: `x` nie ma parametrów, ale nazwę (`String`), zatem jest automatycznie umieszczany w grupie „systemowej" wątków, z kolei `y` jest pod `x` oraz `z` jest pod `y`. Zauważ, iż inicjalizacja przebiega w kolejności zapisu, a więc kod jest poprawny.

Dalej tworzone są dwa wątki i zamieszczane w różnych grupach. `TestThread1` nie posiada metody `run()`, ale zawiera `f()`, która modyfikuje wątek i wypisuje coś, aby było widać, że została wywołana. `TestThread2` jest podklasą i ma dosyć złożona metodę `run()`. Najpierw pobiera grupę aktualnego wątku, następnie wspina się po drzewie dziedzictwa o dwa poziomy, stosując `getParent()` (jest to umyślne, gdyż celowo zamieściłem obiekt `TestThread2` dwa poziomy niżej w hierarchii). Następnie — z wykorzystaniem metody `activeCount()`, aby zapytać, ile wątków znajduje się w grupie wątku i wszystkich grupach podrzędnych — tworzona jest tablica referencji do `Thread`. Metoda `enumerate()` zamieszcza odwołania do wszystkich tych wątków w tablicy `gAll`, a później po prostu przechodzimy przez całą tablicę, wywołując metodę `f()` wobec każdego wątku oraz zmieniamy mu priorytet. Tak więc wątek w grupie będącej „liściem" zmienia wątki w grupach nadrzędnych.

Metoda `list()` wypisuje wszystkie informacje o grupie wątku na standardowe wyjście i jest pomocna do śledzenia zachowania grupy wątku. Oto wyjście programu:

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[jeden,5,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[dwa,5,z]
dwa f()
java.lang.NullPointerException
    at TestThread2.run(TestAccess.java:40)
```

Metoda `list()` nie tylko wypisuje nazwę klasy `ThreadGroup` lub `Thread`, ale również podaje nazwę grupy wątku i jej maksymalny priorytet. W przypadku wątków wypisywana jest nazwa wątku, a po niej priorytet i grupa, do której należy. Jak widać, `list()` zaczyna wypisywać wątki i grupy wątków od akapitu, aby sygnalizować podleganie pod grupę.

Metoda `f()` jest wywoływana przez metodę `run()` klasy `TestThread2`, a zatem jest oczywiste, iż wszystkie wątki w grupie są narażone. Jednak można uzyskać dostęp tylko do wątków, które rozgałęziają się z naszego własnego drzewa grupy systemowej i prawdopodobnie to właśnie oznacza „bezpieczeństwo". Nie można dostać się do drzewa grupy systemowej wątków kogoś innego.

Kontrolowanie grup wątków

Odkładając kwestię bezpieczeństwa, jedyną rzeczą, do której grupy wątków zdają się być przydatne, jest kontrola — można przeprowadzić pewne dziania na całej grupie wątków poprzez pojedyncze polecenie. Kolejny przykład właśnie to pokazuje, podobnie jak ograniczenia na priorytety wewnątrz grup wątków. Umieszczone w komentarzach liczby w nawiasach odpowiadają odwołaniom do wyniku działania, aby można było łatwiej porównać.

```
//: c14:ThreadGroup1.java
// Jak grupy wątku kontrolują priorytety
// wątków w nich zamieszczonych.

public class ThreadGroup1 {
  public static void main(String[] args) {
    // Pobierz wątek systemowy i wypisz informacje o nim:
    ThreadGroup sys =
      Thread.currentThread().getThreadGroup();
    sys.list(); // (1)
    // Ograniczenie priorytetu grupy systemowej:
    sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Zwiększenie priorytetu głównego wątku:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sys.list(); // (2)
    // Próba ustawienia priorytetu nowej grupy na maksymalny:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Próba ustawienia priorytetu nowej grupy na maksymalny:
    Thread t = new Thread(g1, "A");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Ograniczenie maksymalnego priorytetu dla g1,
    // a następnie próba jego inkrementacji:
```

```
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Próba ustawienia priorytetu nowej grupy na maksymalny:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Obniżenie maksymalnego priorytetu poniżej domyślnego
    // priorytetu wątku:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Spojrzenie na priorytet nowego wątku przed
    // i po jego zmianie:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Zamieszczenie g2 jako podgrupy g1 i próba
    // zwiększenia jej priorytetu:
    ThreadGroup g2 = new ThreadGroup(g1, "g2");
    g2.list(); // (8)
    g2.setMaxPriority(Thread.MAX_PRIORITY);
    g2.list(); // (9)
    // Dodaj grupę nowych wątków do g2:
    for (int i = 0; i < 5; i++)
      new Thread(g2, Integer.toString(i));
    // Pokaż informacje o wszystkich grupach
    // wątków i o wątkach:
    sys.list(); // (10)
    System.out.println("Uruchomienie wątków:");
    Thread[] all = new Thread[sys.activeCount()];
    sys.enumerate(all);
    for(int i = 0; i < all.length; i++)
      if(!all[i].isAlive())
        all[i].start();
    // Zawieś i zatrzymaj wszystkie wątki
    // w tej grupie i jej podgrupach:
    System.out.println("Wszystkie wątki uruchomione");
    sys.suspend(); // Nie zalecane w Java 2
    // Program nigdy tu nie dojdzie...
    System.out.println("Wszystkie wątki zawieszone");
    sys.stop(); // Nie zalecane w Java 2
    System.out.println("Wszystkie wątki zatrzymane");
  }
} ///:~
```

Wynik programu przedstawiony poniżej został przeedytowany, aby mógł się zmieścić na stronie (napis `java.lang.` został usunięty) i były widoczne liczby odpowiadające skomentowanym liczbom w kodzie programu.

```
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
java.lang.ThreadGroup[name=main,maxpri=9]
    Thread[main,6,main]
java.lang.ThreadGroup[name=g1,maxpri=9]
    Thread[A,9,g1]
java.lang.ThreadGroup[name=g1,maxpri=8]
    Thread[A,9,g1]
java.lang.ThreadGroup[name=g1,maxpri=8]
```

```
    Thread[A,9,g1]
    Thread[B,8,g1]
java.lang.ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,3,g1]
java.lang.ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,2,g1]
java.lang.ThreadGroup[name=g2,maxpri=3]
java.lang.ThreadGroup[name=g2,maxpri=3]
java.lang.ThreadGroup[name=main,maxpri=9]
    Thread[main,6,main]
    java.lang.ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,2,g1]
        java.lang.ThreadGroup[name=g2,maxpri=3]
            Thread[0,3,g2]
            Thread[1,3,g2]
            Thread[2,3,g2]
            Thread[3,3,g2]
            Thread[4,3,g2]
Uruchomienie wątków:
Wszystkie wątki uruchomione
```

Każdy program ma działający przynajmniej jeden wątek i pierwszym działaniem w `main()` jest wywołanie statycznej metody klasy `Thread` o nazwie `currentThread()`. Z tego wątku uzyskiwana jest grupa wątku i dla niego wywoływana metoda `list()`. Na wyjściu dostajemy:

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
```

Jak widać, nazwa głównej grupy wątku to `system`, a nazwa głównego wątku to `main` i należy on do grupy wątku `system`.

Drugi manewr pokazuje, iż maksymalny priorytet grupy system może zostać zmniejszony, a priorytet wątku `main` może być zwiększony:

```
(2) ThreadGroup[name=system,maxpri=9]
     Thread[main,6,system]
```

Trzeci manewr tworzy nową grupę wątku `g1`, która automatycznie należy do systemowej grupy wątku — ponieważ nic innego nie zostało określone. Umieszczamy nowy wątek `A` w grupie `g1`. Po próbie ustawienia priorytetu maksymalnego tej grupy na wyższy poziom i priorytetu `A` tak samo, wynik jest taki:

```
(3) ThreadGroup[name=g1,maxpri=9]
     Thread[A,9,g1]
```

Zatem nie jest możliwe zmienienie maksymalnego priorytetu grupy wątku na wyższy niż w grupie rodzica.

Kolejne rozwiązanie to zmniejszenie maksymalnego priorytetu `g1` o dwa, a następnie zwiększenie w górę do wartości `Thread.MAX_PRIORITY`. Wynik jest taki:

```
(4) ThreadGroup[name=g1,maxpri=8]
    Thread[A,9,g1]
```

Jak widać, zwiększenie maksymalnego priorytetu nie działa. Można tylko zmniejszyć maksymalny priorytet grupy, ale nie zwiększyć. Priorytet wątku `A` natomiast nie ulega zmianie i jest teraz większy niż maksymalny priorytet grupy. Zmiana maksymalnego priorytetu grupy nie wpływa zatem na istniejące wątki.

Piąta zmiana to próba ustawienia nowego wątku na maksymalny priorytet:

```
(5) ThreadGroup[name=g1,maxpri=8]
    Thread[A,9,g1]
    Thread[B,8,g1]
```

Nowy wątek nie może uzyskać nic więcej niż maksymalny priorytet jego grupy.

Domyślny priorytet wątku w tym programie to sześć — jest to priorytet, z jakim zostanie stworzony nowy wątek i jaki pozostanie, jeśli nie będziemy go modyfikować. Fragment szósty zmniejsza maksymalny priorytet grupy wątku poniżej wartości domyślnej, by można zaobserwować, co się wtedy stanie:

```
(6) ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,6,g1]
```

Nawet jeśli maksymalny priorytet grupy jest równy trzy, to nowy wątek wciąż będzie tworzony z domyślną wartością priorytetu równą sześć. Tak więc maksymalny priorytet grupy wątku nie wpływa na priorytet domyślny (okazuje się, że nie ma sposobu na ustawienie priorytetu domyślnego wątku na nową wartość).

Po zmianie priorytetu próba zmniejszenia go o jeden daje następujący wynik:

```
(7) ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,3,g1]
```

Tylko podczas próby zmiany priorytetu wprowadzany jest maksymalny priorytet jego grupy.

Podobny eksperyment ma miejsce w punktach ósmym i dziewiątym, kiedy to powstaje nowa grupa wątku `g2` jako dziecko `g1` i zmieniany jest jej maksymalny priorytet. Jak widać, nie jest możliwe zwiększenie maksimum `g2` ponad to, co w `g1`:

```
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
```

Podczas tworzenia `g2` automatycznie ustawiana jest na maksymalny priorytet grupy `g1`.

Po tych wszystkich eksperymentach cały system grup wątków i wątków wygląda tak:

```
(10)ThreadGroup[name=system,maxpri=9]
    Thread[main,6,system]
    ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,3,g1]
      ThreadGroup[name=g2,maxpri=3]
        Thread[0,6,g2]
        Thread[1,6,g2]
        Thread[2,6,g2]
        Thread[3,6,g2]
        Thread[4,6,g2]
```

Zatem z uwagi na zasady obowiązujące grupy wątków, grupa potomna zawsze musi posiadać priorytet maksymalny o wartości mniejszej lub równej maksymalnemu priorytetowi jej rodzica.

Ostatnia część programu pokazuje metody operujące na całych grupach wątków. Najpierw program przechodzi po całym drzewie wątków i startuje każdy wątek, który jeszcze nie był wystartowany. Grupa system jest wtedy zawieszana i w końcu zatrzymywana (chociaż wydaje się interesujące obejrzenie działania `suspend()` i `stop()` na całej grupie wątków, to trzeba mieć na uwadze, iż użycie tych metod nie jest w Java 2 zalecane). Ale jeśli zawiesimy grupę system, to także zawiesimy wątek `main` i cały program zostanie zamknięty, zatem nigdy nie dojdzie do miejsca, w którym wątki zostaną zatrzymywane. Jeśli zatrzyma się wątek `main`, to wyskoczy wyjątek `ThreadDeath`, a więc nie jest to rzecz, którą należy robić. Ponieważ klasa `ThreadGroup` jest wywiedziana z `Object`, która zawiera metodę `wait()`, to można również wybrać zawieszenie programu na dowolny czas określony liczbą sekund poprzez wywołanie `wait(seconds*1000)`. Oczywiście musimy być w posiadaniu monitora obiektu (zamka), a więc wewnątrz bloku synchronizowanego.

Klasa `ThreadGroup` posiada także metody `suspend()` i `resume()`, toteż można zatrzymać i uruchomić całą grupę, wszystkie wątki i podgrupy jednym poleceniem (powtarzam, iż nie jest to zalecane w Java 2).

Grupy wątków mogą początkowo wyglądać odrobinę tajemniczo, ale pamiętaj o tym, że prawdopodobnie zbyt często nie będziesz ich bezpośrednio używał.

Runnable

Wcześniej w tym rozdziale sugerowałem, aby dobrze się zastanowić przed stworzeniem apletu lub głównej klasy `Frame` jako implementacji `Runnable`. Oczywiście jeśli dziedziczymy po jakiejś klasie *i* jeśli chcemy dodać do klasy zachowania właściwe dla wątku, to użycie interfejsu `Runnable` jest koniecznym i poprawnym rozwiązaniem. Końcowy przykład tego rozdziału wykorzystuje to rozwiązanie poprzez uczynienie klasy panelu `JPanel` odrysowującej różne kolory jako `Runnable`. Aplikacja ta pobiera wartość z wiersza poleceń, aby określić, jak duża jest siatka kolorów i jak długo ma działać `sleep()` wywoływane pomiędzy zmianą koloru. Poprzez zabawę z tymi wartościami odkryjesz parę interesujących i być może niewytłumaczalnych właściwości wątków:

```
//: c14:ColorBoxes.java
```

```
// Wykorzystanie interfejsu Runnable.
// <applet code=ColorBoxes width=500 height=400>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class CBox extends JPanel implements Runnable {
  private Thread t;
  private int pause;
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) {
    this.pause = pause;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
      cColor = newColor();
      repaint();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
}

public class ColorBoxes extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  private int pause = 50;
  public void init() {
    // Pobranie parametrów ze strony HTML:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
```

```
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for (int i = 0; i < grid * grid; i++)
      cp.add(new CBox(pause));
  }
  public static void main(String[] args) {
    ColorBoxes applet = new ColorBoxes();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

Klasa `ColorBoxes` jest zwykłym apletem-aplikacją z metodą `init()`, która ustawia interfejs graficzny użytkownika. Ustawiany jest menedżer ułożenia komponentów `GridLayout`, tak więc ma on komórki `grid` w każdym wymiarze. Następnie dodawana jest odpowiednia liczba obiektów `CBox`, aby wypełnić siatkę, oraz przekazywana jest wartość `pause` do każdego z nich. W metodzie `main()` widać, jak zmienne `pause` i `grid` zyskują wartości domyślne, które można zmienić poprzez przekazanie argumentów z wiersza poleceń albo poprzez wykorzystanie parametrów apletu.

W klasie `CBox` odbywa się cała „brudna robota". Dziedziczy ona z `JPanel` i implementuje interfejs `Runnable`, a więc każdy panel może być również wątkiem. Pamiętaj o tym, że jeśli implementuje się `Runnable`, to nie uzyskujemy obiektu `Thread`, a jedynie klasę, która posiada metodę `run()`. Tym sposobem trzeba jawnie stworzyć obiekt `Thread` i podać obiekt `Runnable` do jego konstruktora, a następnie wywołać `start()` (dzieje się to w konstruktorze). W kasie `CBox` wątek ten zyskał nazwę `t`.

Tablica `colors` obejmuje wszystkie kolory z klasy `Color`. Jest to wykorzystywane w metodzie `newColor()` do uzyskiwania losowo wybieranych kolorów. Aktualny kolor komórki to `cColor`.

Metoda `paintComponent()` jest dosyć prosta — ustawia kolor na `cColor` i wypełnia tym kolorem cały `JPanel`.

W `run()` widnieje nieskończona pęta, która nastawia `cColor` na nowy kolor losowo wybrany i wywołuje `repaint()`, żeby go pokazać. Następnie wątek udaje się na spoczynek na czas określony w wierszu poleceń.

Z uwagi na to, iż projekt jest elastyczny i wątki są związane z każdym elementem `JPanel`, to można poeksperymentować, tworząc tak wiele wątków, jak tylko chcemy (w rzeczywistości istnieje ograniczenie nadużywania liczby wątków w JVM, z którymi może sobie spokojnie radzić).

Program ten także udostępnia interesujące kryterium porównawcze, może bowiem pokazać dramatyczne różnice wydajności pomiędzy implementacjami wielowątkowości w różnych maszynach wirtualnych Javy.

Zbyt wiele wątków

W pewnych sytuacjach zauważysz, iż kolorowe klocki zwyczajnie są niewydolne. Na moim sprzęcie miało to miejsce, gdy siatka osiągnęła rozmiar około 10×10. Czemu się tak dzieje? Jesteś naturalnie zdziwiony, iż Swing może mieć coś z tym wspólnego, a zatem teraz będzie przykład sprawdzający tę przesłankę poprzez tworzenie mniejszej liczby wątków. Kod został zreorganizowany tak, że lista `ArrayList` będzie implementować `Runnable` i przechowywać kolorowe klocki oraz losowo wybierać jeden do aktualizacji. Następnie tworzonych będzie wiele takich list, zależnie od rozmiaru siatki, który wybierzemy. W rezultacie dostajemy znacznie mniej wątków niż kolorowych klocków, a zatem jeżeli to przyspiesza, to będziemy wiedzieli, iż niewydolność w poprzednim przykładzie było spowodowana zbyt dużą liczbą wątków:

```
//: c14:ColorBoxes2.java
// Zrównoważenie wykorzystywania wątków.
// <applet code=ColorBoxes2 width=600 height=500>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CBox2 extends JPanel {
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  void nextColor() {
    cColor = newColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class CBoxList
  extends ArrayList implements Runnable {
  private Thread t;
  private int pause;
```

```
  public CBoxList(int pause) {
    this.pause = pause;
    t = new Thread(this);
  }
  public void go() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CBox2)get(i)).nextColor();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Przerwany");
      }
    }
  }
  public Object last() { return get(size() - 1);}
}

public class ColorBoxes2 extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  // Krótsze domyślne opóźnienie niż w ColorBoxes:
  private int pause = 50;
  private CBoxList[] v;
  public void init() {
    // Pobierz parametry ze strony HTML:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    v = new CBoxList[grid];
    for(int i = 0; i < grid; i++)
      v[i] = new CBoxList(pause);
    for (int i = 0; i < grid * grid; i++) {
      v[i % grid].add(new CBox2());
      cp.add((CBox2)v[i % grid].last());
    }
    for(int i = 0; i < grid; i++)
      v[i].go();
  }
  public static void main(String[] args) {
    ColorBoxes2 applet = new ColorBoxes2();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

W kasie `ColorBoxes2` tworzona i inicjowana jest tablica elementów `CBoxList` w celu przechowywania elementów `grid` `CBoxList` poinformowanych o czasie uśpienia. Jednakowa liczba obiektów `CBox2` jest następnie dodawana do każdej listy `CBoxList` i każdej takiej liście mówimy `go()`, co powoduje wystartowanie jej wątku.

Klas `CBox2` jest podobna do `CBox` — odrysowuje się z wybranym losowym kolorem. Ale to *wszystko*, co robi `CBox2`. Wszystko inne dotyczące wątku zostało przeniesione do `CBoxList`.

`CBoxList` również mogła być dziedziczona z `Thread` i posiadać obiekt składowy typu `ArrayList`. Taki projekt ma tę zaletę, iż metody `add()` i `get()` mogą wtedy dostawać określony argument i zwracać wartości stosownych typów zamiast ogólnego `Object` (ich nazwy również mogłyby być zmienione na jakieś krótsze). Jednak rozwiązanie zastosowane tutaj wydaje się na pierwszy rzut oka wymagać mniej kodu. Dodatkowo automatycznie zachowuje wszystkie inne właściwości `ArrayList`. Z wszystkimi rzutowaniami i nawiasowaniami koniecznymi dla `get()` może się okazać, że kod jednak rośnie.

Tak jak poprzednio, jeśli implementujemy `Runnable`, nie zyskujemy całego wyposażenia, które przychodzi z `Thread`, a zatem trzeba stworzyć nowy wątek i przekazać do konstruktora, aby mieć coś do wystartowania — jak to widać w konstruktorze `CBoxList` i metodzie `go()`. Metoda `run()` po prostu wybiera losową liczbę spośród odpowiadających elementów listy i wywołuje `nextColor()` dla elementu, aby spowodować, by wybrał sobie nowy kolor.

Podczas uruchomienia programu widać, iż istotnie działa szybciej i szybciej reaguje (jeśli go przerwiemy, to szybciej się zatrzyma) i zdaje się grzęznąć w takim samym stopniu dopiero przy większym rozmiarze siatki. Tak więc dochodzi nowy czynnik do równania wątkowego — trzeba uważać, aby nie mieć „zbyt wielu wątków" (cokolwiek to oznacza wobec konkretnego programu i platformy — tu spowolnienie w `ColorBoxes` wydaje się być spowodowane faktem, iż jest tu tylko jeden wątek odpowiedzialny za wszystkie odrysowania i grzęźnie przy zbyt wielu żądaniach). Jeżeli masz zbyt wiele wątków, to musisz spróbować zastosować techniki podobne jak ta powyżej, żeby „zrównoważyć" liczbę wątków w programie. Jeżeli dostrzegasz problemy z wydajnością w programach wielowątkowych, to przyjrzyj się kilku istotnym sprawom:

1. Czy nasz wystarczającą liczbę wywołań: `sleep()`, `yield()` i (lub) `wait()`?
2. Czy wywołania `sleep()` odpowiednio długo usypiają wątki?
3. Czy nie uruchomiłeś zbyt wielu wątków?
4. Czy próbowałeś różnych platform i maszyn wirtualnych Javy?

Temu podobne kwestie są jednym z powodów, iż programowanie wielowątkowe jest często uważane za sztukę.

Podsumowanie

Zrozumienie tego, kiedy stosować wielowątkowość, a kiedy jej unikać, jest dość istotne. Głównym powodem by ją wykorzystywać jest zarządzanie wieloma zadaniami, które przyczynią się do znacznie efektywniejszego wykorzystania komputera (włączając zdolność do przezroczystego rozproszenia zadań między wiele procesorów) albo wzrostu wygody dla użytkownika. Klasyczny

przykład równoważenia dostępu do zasobów to wykorzystywanie procesora podczas odczekania na operacje wejścia-wyjścia. Klasyczny przykład wygody użytkowania to monitorowanie przycisku „stop" podczas długiego ściągania plików z sieci.

Główne wady wielowątkowości to:

1. spowolnienie podczas oczekiwania na współdzielone zasoby;
2. dodatkowy narzut na pracę procesora konieczny do zarządzania wątkami;
3. niepotrzebna złożoność, tak jak głupia idea posiadania osobnych wątków do uaktualniania każdego elementu tablicy;
4. patologie, takie jak zagłodzenia, wyścigi czy impas.

Na korzyść użycia wątków przemawia dodatkowo to, iż zastępują one wykonanie „ciężkiej" wymiany w kontekście procesów (rzędu 1000 instrukcji) przez wymianę „lekką" (rzędu 100 instrukcji). Ponieważ wszystkie wątki w danym procesie dzielą tę samą przestrzeń adresową pamięci, to wymiana kontekstowo lekka zmienia tylko wykonanie programu i zmienne lokalne. Z drugiej strony zmiana w procesie — ciężkie przełączenie kontekstowe — musi wymieniać pełną przestrzeń adresową.

Zastosowanie wątków to jak wkroczenie w całkowicie nowy świat i nauka zupełnie nowego języka programowania albo przynajmniej nowego zestawu pojęć języka. Wraz z pojawieniem się obsługi wielowątkowości w większości mikrokomputerowych systemów operacyjnych, rozszerzenia dla wielowątkowości zaczęły się pojawiać także w językach programowania czy też bibliotekach. W każdym razie programowanie z użyciem wątków zdaje się tajemnicze i wymaga zmiany sposobu myślenia, i wygląda podobnie do rozwiązania wielowątkowości w innych językach, tak więc wraz ze zrozumieniem wątków, zrozumiesz pewien wspólny język. Chociaż obsługa wielowątkowości może czynić Javę bardziej skomplikowanym językiem, ale nie można jej za to winić. Wątki po prostu są skomplikowane.

Jedna z największych trudności związanych z wątkami pojawia się z tego powodu, iż więcej niż jeden wątek może współdzielić jakiś zasób — jak na przykład pamięć w obiekcie — i musimy się upewnić, czy wiele wątków nie usiłuje czytać i zmieniać tego zasobu jednocześnie. Wymaga to rozważnego zastosowania słowa kluczowego `synchronized`, które pomaga, ale musi być dobrze zrozumiane, ponieważ może po cichu wprowadzić sytuację impasu.

Ponadto z pewnością stworzenie aplikacji opartej na wątkach jest sztuką. Java została zaprojektowana, aby pozwalać na stworzenie tak wielu obiektów, ile tylko jest potrzebne do rozwiązania zadania — przynajmniej teoretycznie (na przykład stworzenie milionów obiektów dla inżynierskiej analizy elementów skończonych może nie być możliwe w praktyce do uzyskania). Jednak wydaje się, iż istnieje górna granica liczby wątków, które chcemy utworzyć, ponieważ z pewnych względów duża liczba wątków zdaje się nieporęczna. Ten punkt krytyczny nie znajduje się w wielu tysiącach, jak może być z obiektami, ale raczej w małych setkach, czasami nawet wynosi mniej niż 100. Ponieważ najczęściej będziesz tworzył tylko garść wątków do pokonania problemu, to nie jest to spora niedogodność, mimo to w bardziej ogólnych projektach staje się to ograniczeniem.

Znaczącą nieintuicyjną kwestią użycia wątków jest to, iż ze względu na harmonogramowanie można przeważnie uczynić aplikację *szybszą* poprzez wstawienie wywołań `sleep()` wewnątrz głównej pętli metody `run()`. To zdecydowanie czyni takie programowanie sztuką, szczególnie gdy dłuższe

uśpienia mogą spowodować zwiększenie wydajności. Oczywiście przyczyną takiego zachowania jest to, iż krótsze opóźnienia mogą powodować przerwania pracy zarządcy podziału czasu, spowodowane obudzeniem, kiedy akurat działający wątek jest gotowy do uśpienia, zmuszając zarządcę do zatrzymania go i ponownego startu później, by mógł zakończyć to, co robił i dopiero został uśpiony. Dochodzi dodatkowa kwestia zdania sobie sprawy, jakie to może stać się niechlujne.

Jak może zauważyłeś, brakuje w tym rozdziale przykładu animacji, która jest jedną z najbardziej popularnych rzeczy w przypadku apletów. Jednak pełne rozwiązanie (z dźwiękiem) tego zadania jest dostępne wraz z JDK w grupie przykładów demo. Można się spodziewać lepszego wsparcia dla animacji w przyszłych wersjach Javy, choć pojawiają się kompletnie odmienne, nie w Javie, nieprogramistyczne rozwiązania animacji dla Internetu, które prawdopodobnie będą wypierały takie tradycyjne podejście. Po szczegółowe wyjaśnienia na temat działania animacji w Javie zajrzyj do książki *Core Java 2* — Horstmann i Cornell, Prentice-Hall, 1997. Natomiast bardziej zaawansowany opis wątków znajdziesz w *Concurrent Programming in Java* — Doug Lea, Addison-Wesley, 1997 oraz *Java Threads* — Oaks i Wong, O'Reilly, 1997.

Ćwiczenia

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.BruceEckel.com*.

1. Odziedzicz nową klasę z klasy `Thread` i przesłoń w niej metodę `run()`. W jej wnętrzu wypisz komunikat, a następnie wywołaj `sleep()`. Powtórz to trzykrotnie i wyjdź z `run()`. Zamieść komunikat początkowy w konstruktorze i przesłoń `finalize()`, aby wypisywała komunikat kończący. Napisz osobną klasę wątku, wywołującą `System.gc()` i `System.runFinalization()` wewnątrz metody `run()` oraz wypisującą komunikaty, jak poprzednio. Utwórz kilka obiektów wątków obu typów i uruchom je, aby zobaczyć, co się stanie.
2. Zmodyfikuj `Sharing2.java`, dodając blok synchronizowany wewnątrz metody `run()` klasy `TwoCounter` zamiast synchronizacji całej metody.
3. Stwórz dwie podklasy `Thread` — jedną z metodą `run()`, która uruchamia i przejmuje referencje do drugiego obiektu `Thread`, a następnie woła `wait()`. Metoda `run()` drugiej klasy powinna wywoływać `notifyAll()` wobec pierwszego wątku po upływie pewnego czasu, by pierwszy wątek mógł wyświetlić komunikat.
4. W przykładzie `Counter5.java` z wnętrza klasy `Ticker2` usuń metodę `yield()` i objaśnij rezultaty. Zastąp `yield()` metodą `sleep()` i też wyjaśnij, co się dzieje.
5. W `ThreadGroup1.java` zastąp wywołanie `sys.suspend()` wywołaniem `wait()` dla grupy wątków, powodujące oczekiwanie przez dwie sekundy. Aby to działało poprawnie, musisz uzyskać blokadę dla `sys` wewnątrz bloku `synchronized`.
6. Zmień `Daemons.java`, by metoda `main()` zawierała `sleep()` zamiast `readLine()`. Przeprowadź eksperymenty z różnymi czasami uśpienia, aby zaobserwować, co się dzieje.
7. Zlokalizuj przykład `GreenhouseControls.java` z rozdziału 8., składający się z trzech plików. W `Event.java` klasa `Event` opiera się na obserwacji czasu. Zmień `Event` tak, by była wątkiem `Thread` i zmień resztę projektu, aby działał z tą nową klasą.

8. Zmodyfikuj ćwiczenie 7. tak, by klasa `java.util.Timer` zamieszczona w JDK 1.3 została wykorzystana do uruchomienia systemu.
9. Zaczynając od przykładu `SineWave.java` z rozdziału 13., napisz program (aplet-aplikację wykorzystującą klasę `Console`), który rysuje animowane sinusoidalne fale wydające się przesuwać po okienku podglądu jak na oscyloskopie, działający na podstawie `Thread`. Prędkość animacji powinna być sterowana przez kontrolkę `java.swing.JSlider`.
10. Zmodyfikuj ćwiczenie 9. tak, aby w ramach aplikacji było tworzonych wiele paneli z sinusoidami. Liczba paneli powinna być kontrolowana przez znacznik HTML lub parametry wiersza poleceń.
11. Zmodyfikuj ćwiczenie 9. tak, aby wykorzystać klasę `java.swing.Timer` do działania animacji. Zauważ różnice pomiędzy tym a `java.util.Timer`.
12. Zmień `SimpleThread.java` tak, aby wszystkie wątki były wątkami demonami i sprawdź, iż pogram kończy się, kiedy tylko `main()` jest gotowa do wyjścia.