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

Java. Współbieżność dla praktyków

Autor: Zespół autorów
Tłumaczenie: Rafał Jońca
ISBN: 978-83-246-0921-5
Tytuł oryginału: Java Concurrency in Practice
Format: B5, stron: 376

Twórz bezpieczne i wydajne aplikacje wielowątkowe

Chcesz podnieść wydajność swoich aplikacji? Planujesz stworzenie systemu, który będzie uruchamiany na maszynach wyposażonych w procesory wielordzeniowe? A może próbowałeś już tworzyć aplikacje wielowątkowe, ale zniechęciłeś się po wielogodzinnych poszukiwaniach przyczyn błędów, które pojawiają się przy wysokich obciążeniach? Java niemal od początku swego istnienia jest wyposażona w mechanizmy umożliwiające tworzenie aplikacji wielowątkowych, lecz dopiero wersja 5. wniosła zupełnie nową jakość, dzięki wielu poprawkom zwiększającym wydajność maszyny wirtualnej oraz dodatkowym klasom ułatwiającym osiąganie lepszej współbieżności.

W książce „Java. Współbieżność dla praktyków” znajdziesz wyczerpujący opis metod projektowania i tworzenia aplikacji wielowątkowych. Przeczytasz nie tylko o klasach, ich działaniu i sposobach wykorzystania, ale również poznasz wzorce projektowe, praktyki programistyczne i modele, dzięki którym programowanie współbieżne jest łatwiejsze. Znajdziesz tu praktyczne aspekty oraz przykłady tworzenia pewnych, skalowalnych i łatwych w konserwacji aplikacji współbieżnych. Dowiesz się także, w jaki sposób testować aplikacje wielowątkowe, wynajdywać w nich błędy i usuwać je.

Dzięki książce poznasz:

- Możliwości wykorzystania wątków
- Podstawy stosowania wątków
- Współdzielenie obiektów
- Struktura aplikacji wielowątkowych
- Zarządzanie wątkami i zadaniami
- Zastosowania pul wątków
- Optymalizowanie wydajności
- Skalowalność aplikacji wielowątkowych
- Testowanie aplikacji współbieżnych
- Model pamięci Javy
- Tworzenie własnych synchronizatorów

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa **9**

Rozdział 1. Wprowadzenie **13**

1.1. (Bardzo) krótka historia współbieżności 13
1.2. Zalety wątków 15
1.3. Ryzyka związane z wątkami 18
1.4. Wątki są wszędzie 21

Część I Podstawy **25**

Rozdział 2. Wątki i bezpieczeństwo **27**

2.1. Czym jest bezpieczeństwo wątkowe? 29
2.2. Niepodzielność 31
2.3. Blokady 35
2.4. Ochrona stanu za pomocą blokad 39
2.5. Żywotność i wydajność 41

Rozdział 3. Współdzielenie obiektów **45**

3.1. Widoczność 45
3.2. Publikacja i ucieczka 51
3.3. Odosobnienie w wątku 54
3.4. Niezmienność 58
3.5. Bezpieczna publikacja 61

Rozdział 4. Kompozycja obiektów **67**

4.1. Projektowanie klasy bezpiecznej wątkowo 67
4.2. Odosobnienie egzemplarza 71
4.3. Delegacja bezpieczeństwa wątkowego 76
4.4. Dodawanie funkcjonalności do istniejących klas bezpiecznych wątkowo 82
4.5. Dokumentowanie strategii synchronizacji 86

Rozdział 5. Bloki budowania aplikacji **89**

5.1. Kolekcje synchronizowane 89
5.2. Kolekcje współbieżne 94
5.3. Kolejki blokujące oraz wzorzec producenta i konsumenta 97
5.4. Metody blokujące i przerywane 102
5.5. Synchronizatory 104
5.6. Tworzenie wydajnego, skalowalnego bufora wyników 112
Podsumowanie części I 117

Część II Struktura aplikacji współbieżnej 119

Rozdział 6. Wykonywanie zadań 121

6.1. Wykonywanie zadań w wątkach 121
6.2. Szkielet Executor 125
6.3. Znajdowanie sensownego zrównoleglenia 132
Podsumowanie 141

Rozdział 7. Anulowanie i wyłączanie zadań 143

7.1. Anulowanie zadań 144
7.2. Zatrzymanie usługi wykorzystującej wątki 158
7.3. Obsługa nietypowego zakończenia wątku 167
7.4. Wyłączanie maszyny wirtualnej 170
Podsumowanie 173

Rozdział 8. Zastosowania pul wątków 175

8.1. Niejawnie splecione zadania i strategie wykonania 175
8.2. Określanie rozmiaru puli wątków 178
8.3. Konfiguracja klasy ThreadPoolExecutor 179
8.4. Rozszerzanie klasy ThreadPoolExecutor 187
8.5. Zrównoleglenie algorytmów rekurencyjnych 188
Podsumowanie 195

Rozdział 9. Aplikacje z graficznym interfejsem użytkownika 197

9.1. Dlaczego graficzne interfejsy użytkownika są jednowątkowe? 197
9.2. Krótkie zadanie interfejsu graficznego 201
9.3. Długie czasowo zadania interfejsu graficznego 203
9.4. Współdzielone modele danych 208
9.5. Inne postacie podsystemów jednowątkowych 209
Podsumowanie 210

Część III Żywotność, wydajność i testowanie 211

Rozdział 10. Unikanie hazardu żywotności 213

10.1. Blokada wzajemna 213
10.2. Unikanie i diagnostyka blokad wzajemnych 223
Podsumowanie 228

Rozdział 11. Wydajność i skalowalność 229

11.1. Myślenie na temat wydajności 229
11.2. Prawo Amdahla 233
11.3. Koszta wprowadzane przez wątki 237
11.4. Zmniejszanie rywalizacji o blokadę 240
11.5. Przykład — porównanie wydajności obiektów Map 250
11.6. Redukcja narzutu przełączania kontekstu 251
Podsumowanie 253

Rozdział 12. Testowanie programów współbieżnych 255

12.1. Testy sprawdzające poprawność 256
12.2. Testowanie wydajności 268
12.3. Unikanie pomyłek w testach wydajności 273
12.4. Testy uzupełniające 278
Podsumowanie 281

Część IV Techniki zaawansowane 283

Rozdział 13. Blokady jawne 285

13.1. Interfejs Lock i klasa ReentrantLock 285
13.2. Rozważania na temat wydajności 290
13.3. Uczciwość 291
13.4. Wybór między synchronized i ReentrantLock 293
13.5. Blokady odczyt-zapis 294
Podsumowanie 297

Rozdział 14. Tworzenie własnych synchronizatorów 299

14.1. Zarządzanie zależnością od stanu 299
14.2. Wykorzystanie kolejek warunków 306
14.3. Jawne obiekty warunków 314
14.4. Anatomia synchronizatora 316
14.5. Klasa AbstractQueuedSynchronizer 318
14.6. AQS w klasach synchronizatorów pakietu java.util.concurrent 321
Podsumowanie 324

Rozdział 15. Zmienne niepodzielne i synchronizacja nieblokująca 325

15.1. Wady blokowania 326
15.2. Sprzętowa obsługa współbieżności 327
15.3. Klasy zmiennych niepodzielnych 331
15.4. Algorytmy nieblokujące 335
Podsumowanie 342

Rozdział 16. Model pamięci Javy 343

16.1. Czym jest model pamięci i dlaczego ma mnie interesować? 343
16.2. Publikacja 350
16.3. Bezpieczeństwo inicjalizacji 355
Podsumowanie 356

Dodatki 357

Dodatek A Adnotacje związane ze współbieżnością 359

A.1. Adnotacje dla klas 359
A.2. Adnotacje pól i metod 360

Dodatek B Bibliografia 361

Skorowidz 365

Rozdział 3.

Współdzielenie obiektów

Na początku rozdziału 2. pojawiło się stwierdzenie, że poprawne programy współbieżne muszą przede wszystkim właściwie zarządzać dostępem do współdzielonego, zmiennego stanu. Tamten rozdział dotyczył użycia synchronizacji do zabezpieczenia się przed wieloma wątkami korzystającymi z tych samych danych w tym samym momencie. Ten prezentuje techniki współdzielenia i publikacji obiektów, by były bezpieczne do stosowania w wielu wątkach. Razem oba elementy stanowią podstawę tworzenia klas bezpiecznych wątkowo i poprawnej konstrukcji współbieżnych aplikacji za pomocą klas biblioteki `java.util.concurrent`.

W poprzednim rozdziale przedstawiliśmy, w jaki sposób bloki i metody `synchronized` zapewniają niepodzielność operacji. Wielu osobom wydaje się, że te bloki dotyczą **tylko** niepodzielności i oznaczania sekcji krytycznych. Synchronizacja ma także inny istotny, choć subtelny, aspekt — **widoczność pamięci**. Chcemy nie tylko zapewnić, by gdy jeden wątek modyfikuje stan obiektu, inne mu w tym nie przeszkadzały, ale i to, by inne wątki rzeczywiście **widziały** dokonaną zmianę. Nie można tego osiągnąć bez synchronizacji. Obiekty są bezpiecznie publikowane czy to za pomocą jawnej synchronizacji, czy przez zastosowanie synchronizacji wybudowanej w klasy biblioteki.

3.1. Widoczność

Widoczność to subtelny temat, bo zadania, które mogą się nie udać, są mało intuicyjne. W środowisku jednowątkowym, gdy zapisujemy wartość do zmiennej i później ją odczytujemy (w międzyczasie nie było innych zapisów), możemy się spodziewać otrzymania tej samej wartości. Wydaje się to w miarę naturalne. Z tego względu początkowo trudno zaakceptować, że w przypadku wielu odczytów i zapisów z wielu wątków **przedstawione założenie może nie zaistnieć**. Ogólnie nie ma gwarancji, iż wątek odczytujący zobaczy wartość zapisaną przez inny wątek w odpowiednim czasie, a nawet w ogóle. Aby zapewnić widoczność zapisów do pamięci w różnych wątkach, należy użyć synchronizacji.

Listing 3.1 przedstawia klasę NoVisibility wskazującą, co może pójść nie tak, jeśli wątki współdzielą dane bez synchronizacji. Dwa wątki, główny i odczytujący, korzystają ze współdzielonych zmiennych ready i number. Główny wątek uruchamia wątek odczytujący, a następnie ustawia number na 42 i ready na true. Wątek odczytujący czeka, aż ready będzie równe true, i dopiero wtedy wyświetla wartość number. Choć wydawałoby się oczywiste, że NoVisibility zawsze wyświetli 42, w praktyce może wyświetlić 0 lub w ogóle nie wyjść z pętli! Z powodu braku odpowiedniej synchronizacji nie mamy żadnej gwarancji, że wartości ready i number zapisane przez główny wątek zobaczy wątek odczytujący.

Listing 3.1. *Współdzielenie zmiennych bez synchronizacji. Nie rób tak*

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Kod klasy NoVisibility może przebywać w pętli nieskończenie długo, bo wartość ready może nigdy nie zostać zauważona przez wątek odczytujący. Co jeszcze dziwniejsze, kod może wyświetlić wartość 0, gdy zapis ready będzie widoczny **wcześniej** niż zapis number (tak zwana **zmiana kolejności**). Nie ma gwarancji, że operacje jednego wątku wykonają się w kolejności podanej przez program, jeśli tylko zmiana kolejności będzie niezauważalna przez wątek dokonujący modyfikacji — **nawet jeśli oznacza to zmianę kolejności zmian w innych wątkach**[1]. Choć główny wątek w kodzie źródłowym najpierw zapisuje number, a później ready, bez synchronizacji inny wątek może zauważyć te operacje w odwrotnej kolejności (lub nawet wcale ich nie widzieć).

> Przy braku synchronizacji kompilator, procesor i system wykonawczy mogą wykonywać dziwne „przemeblowania” operacji, które mają wykonać. Próby wcześniejszego logicznego wskazywania kolejności wykonania w pamięci określonych działań przy braku synchronizacji wielowątkowej niemal na pewno będą niepoprawne.

[1] Mogłoby to wskazywać na złe zaprojektowanie systemu, ale tak naprawdę wynika to z faktu wykorzystywania przez maszynę wirtualną pełnej wydajności nowoczesnych systemów wieloprocesorowych. Przy braku synchronizacji model pamięci Javy dopuszcza, by kompilator zmienił kolejność operacji i buforował wartości w rejestrach. Dopuszcza też zmianę kolejności wykonania działań przez procesor i jego bufory. Więcej informacji na ten temat znajduje się w rozdziale 16.

Klasa NoVisibility to chyba najprostsza postać programu współbieżnego — dwa wątki i dwie współdzielone zmienne — a mimo to zbyt łatwo wysnuć złe wnioski co do jego działania i sposobu opuszczenia pętli. Wysnucie odpowiednich wniosków co do kolejności działań w niepoprawnie zsynchronizowanym programie wielowątkowym jest wręcz niemożliwe.

Wszystko to brzmi groźnie i rzeczywiście takie jest. Na szczęście istnieje prosty środek zaradczy — **każdorazowe użycie odpowiedniej synchronizacji, gdy tylko dane są współdzielone przez wiele wątków**.

3.1.1. Nieświeże dane

Klasa NoVisibility przedstawia jeden z powodów zwracania przez niepoprawnie zsynchronizowane programy zadziwiających wyników — **nieświeżość danych**. Gdy wątek odczytujący testuje wartość ready, może widzieć przedatowaną wartość. Jeśli synchronizacji nie stosuje się **przy każdym dostępie do zmiennej**, można uzyskać nieświeży odczyt. Co gorsza, taki odczyt nie odbywa się na zasadzie wszystko albo nic — wątek jedną zmienną odczyta aktualną, a drugą nieważną (nawet jeśli teoretycznie została zapisana jako pierwsza).

Czasem nieświeże jedzenie można spożyć — jest to tylko mniej przyjemne. Nieświeże dane bywają bardzo groźne. Choć nieaktualny licznik odwiedzin aplikacji internetowej raczej nikomu nie zaszkodzi[2], tak inna nieświeża wartość może doprowadzić do poważnych błędów. W klasie NoVisibility nieaktualne dane prowadzą czasem do wyświetlenia złego wyniku, a nawet do zablokowania jednego z wątków. Sprawa komplikuje się jeszcze bardziej, gdy nieświeżość dotyczy referencji do obiektów, na przykład łączy listy jednokierunkowej. **Nieświeże dane mogą powodować poważne i tajemnicze pomyłki, jak niespodziewane wyjątki, błędy struktury danych, niedokładne obliczenia oraz pętle nieskończone**.

Klasa MutableInteger z listingu 3.2 nie jest bezpieczna wątkowo, bo z pola value korzystają metody get() i set() bez synchronizacji. Poza innymi hazardami, klasa jest narażona na nieświeżość danych: jeśli jeden wątek wywołuje set(), drugi wątek wywołujący get() może nie widzieć aktualizacji.

Listing 3.2. *Niezabezpieczona przed wątkami klasa przechowująca liczbę całkowitą*

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() { return value; }
    public void set(int value) { this.value = value; }
}
```

[2] Odczyt danych bez synchronizacji przypomina użycie poziomu izolacji READ_UNCOMMITED (odczyt niezatwierdzony) w bazie danych, gdy staramy się zwiększyć wydajność kosztem dokładności. Odczyt niesynchronizowany to coś więcej niż tylko utrata dokładności, bo widoczna wartość współdzielonej zmiennej może być naprawdę poważnie przedatowana.

Bezpieczeństwo klasie `MutableInteger` zapewnimy, synchronizując metodę ustawiającą i pobierającą. Nową wersję przedstawia klasa `SynchronizedInteger` z listingu 3.3. Synchronizacja tylko metody ustawiającej nie wystarcza — w takiej sytuacji wątek pobierający wartość nadal byłby narażony na nieświeżość.

Listing 3.3. *Zabezpieczona przed wątkami klasa przechowująca liczbę całkowitą*

```
@ThreadSafe
public class SynchronizedInteger {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

3.1.2. Niepodzielne operacje 64-bitowe

Gdy wątek odczytuje zmienną bez synchronizacji, może widzieć nieświeżą wartość, ale przynajmniej jest to wartość umieszczona tam przez inny wątek, a nie jakaś losowa wartość. Mówi się w takiej sytuacji o **bezpieczeństwie poprawności**.

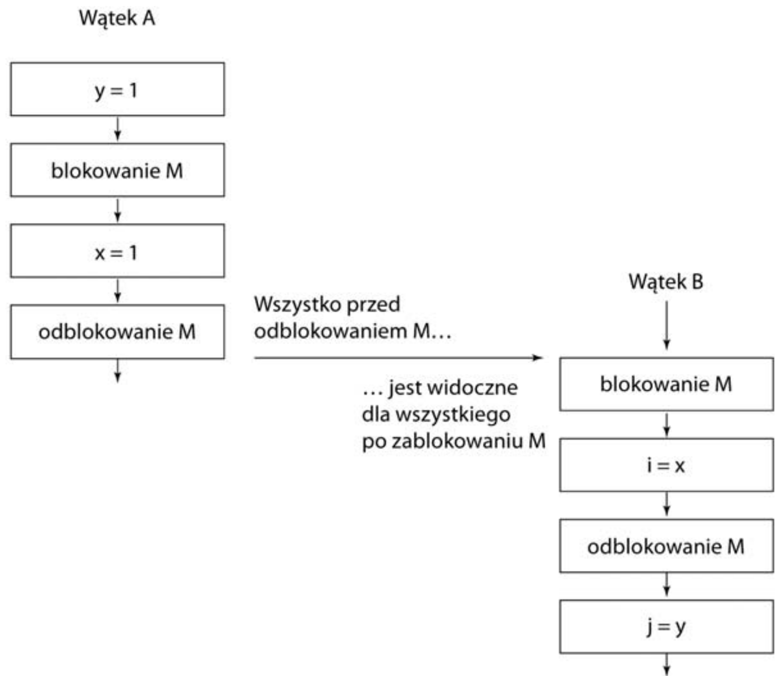
To bezpieczeństwo dotyczy wszystkich zmiennych poza jednym wyjątkiem — 64-bitowych zmiennych liczbowych (`double` i `long`) niezadeklarowanych jako `volatile` (patrz punkt 3.1.4). Model pamięci Javy wymaga, by operacje pobrania i zapamiętania były niepodzielne, ale dla nieulotnych zmiennych `long` i `double` maszyna wirtualna może potraktować odczyt lub zapis 64-bitowy jako dwie operacje 32-bitowe. Jeśli zapis i odczyt takiej nieulotnej zmiennej odbywa się w dwóch różnych wątkach, wątek odczytujący może przeczytać niższe 32 bity nowej wartości i wyższe 32 bity starej[3]. Nawet jeśli ktoś nie przejmuje się nieświeżymi danymi, powinien uważać na współdzielenie zmiennych typu `long` lub double w programach wielowątkowych, jeśli nie są one zadeklarowane jako `volatile` lub chronione blokadą.

3.1.3. Blokowanie i widoczność

Blokada wewnętrzna służy do zagwarantowania, że jeden wątek zauważy zmianę wykonaną przez inny wątek w sposób przewidywalny, co przedstawia rysunek 3.1. Gdy wątek A wykonuje blok `synchronized` i w tym czasie wątek B próbuje uzyskać dostęp do tego bloku chronionego tą samą blokadą, wartości zmiennych widoczne dla A przed zwolnieniem blokady będą również widoczne dla B, gdy uzyska wreszcie dostęp do bloku. Innymi słowy, wszystko, co A wykonało w synchronizowanym bloku, będzie widoczne dla B, gdy wykona blok chroniony tą samą blokadą. **Bez synchronizacji nie ma takiej gwarancji**.

[3] W czasie powstawania specyfikacji maszyny wirtualnej niewiele architektur procesorów mogło wydajnie obsłużyć niepodzielne 64-bitowe operacje arytmetyczne.

Rysunek 3.1. *Widoczność gwarantowana przez synchronizację*



Możemy dodać kolejną regułę wymagającą synchronizacji wszystkich wątków **tą samą** blokadą w momencie dostępu do współdzielonej zmiennej — gwarancję widoczności zmiany wykonanej przez jeden wątek dla wszystkich innych wątków. Jeśli wątek odczytuje wartość zmiennej bez synchronizacji, może otrzymać nieświeże dane.

> Blokowanie nie dotyczy wyłącznie wzajemnego wykluczenia wykonywania, ale również widoczności pamięci. Aby zapewnić, że wszystkie wątki widzą najnowsze wersje informacji we współdzielonych zmiennych, wątki odczytujące i zapisujące muszą synchronizować się za pomocą tej samej blokady.

3.1.4. Zmienne ulotne

Język Javy zawiera dodatkowo słabszą wersję synchronizacji w postaci zmiennych ulotnych, która zapewnia przewidywalny sposób propagacji aktualizacji do innych wątków. Pole z modyfikatorem `volatile` wskazuje kompilatorowi i systemowi wykonawczemu, że zmienna jest współdzielona i wykonywanych na niej w pamięci operacji nie należy układać w innej kolejności niż wskazana w kodzie źródłowym. Zmienne ulotne nie są przechowywane w rejestrach ani w buforach, gdy są ukryte przed innymi procesorami. W ten sposób odczyt w dowolnym momencie zawsze zwraca najnowszą wersję zmiennej zapisaną przez dowolny wątek.

Warto zmienne ulotne traktować w taki sposób, jakby zachowywały się jak klasa `SynchronizedInteger` z listingu 3.3, w której wywołania `get()` i `set()` zastąpiono odczytami

i zapisami[4]. Dostęp do zmiennej ulotnej nie zakłada żadnych blokad, więc nie blokuje żadnego wątku. W ten sposób zmienne ulotne nie generują takiego narzutu jak pełny mechanizm synchronizacji[5].

Efekt widoczności zmiennych ulotnych wykracza poza wartość samej zmiennej ulotnej. Gdy wątek A zapisze dane w zmiennej ulotnej, a wątek B odczyta tę samą zmienną, wartości **wszystkich** zmiennych, które były widoczne w A przed zapisaniem zmiennej ulotnej, będą widoczne w B po dokonaniu odczytu zmiennej ulotnej. Z punktu widzenia widoczności pamięci zapis zmiennej ulotnej przypomina wyjście z bloku synchronizującego. Nie radzimy zbyt mocno polegać na zmiennych ulotnych w kwestii widoczności. Kod, który korzysta z tego rozwiązania, trudniej zrozumieć i testować niż kod jawnie stosujący blokady.

> Używaj zmiennych ulotnych tylko wtedy, gdy upraszczają implementację i weryfikację strategii synchronizacji. Unikaj ich stosowania, gdy weryfikacja wymagałaby dokładnej analizy przypadku. Dobre użycia zmiennych ulotnych dotyczą zapewnienia widoczności ich własnego stanu, obiektu, do którego się odnoszą, lub wystąpienia istotnego zdarzenia (na przykład inicjalizacji lub wyłączania systemu).

Listing 3.4 ilustruje typowy przykład użycia zmiennych ulotnych — sprawdzenie znacznika stanu, by wykryć potrzebę opuszczenia pętli. W przykładzie wątek stara się zasnąć z użyciem metody zliczającej wirtualne owce. Aby przykład działał poprawnie, zmienna `asleep` musi być ulotna. W przeciwnym razie wątek mógłby nie zauważyć ustawienia zmiennej `asleep` przez inny wątek[6]. Nic nie stoi na przeszkodzie, by w tym miejscu użyć blokady zapewniającej widoczność, ale uczyniłaby ona kod mniej przejrzystym.

Listing 3.4. *Zliczanie owiec*

```
volatile boolean asleep;
...
    while (!asleep)
        countSomeSheep();
```

Zmienne ulotne są wygodne, ale mają swoje ograniczenia. Najczęściej stosuje się je jako znaczniki zakończenia, przerwania lub statusu (patrz listing 3.4). Można ich użyć również do innych rodzajów informacji o stanie, ale wtedy należy bardziej uważać.

4 Nie jest to dokładna analogia. Efekt widoczności pamięciowej klasy `SynchronizedInteger` jest mocniejszy od zmiennych ulotnych. Szczegóły w rozdziale 16..

5 Odczyty ulotne są tylko odrobinę wolniejsze od zwykłych odczytów w większości nowoczesnych architektur procesorów.

6 Uwaga dla testujących: dla aplikacji serwerowych zawsze włączaj opcję `-server` maszyny wirtualnej, nawet w trakcie implementacji i testowania. Maszyna wirtualna w wersji serwerowej przeprowadza więcej optymalizacji niż wersja kliencka, na przykład przez wyrzucanie zmiennych poza pętlę, jeśli nie są w niej modyfikowane. Kod mogący działać poprawnie w wersji klienckiej (w trakcie testów) przestanie działać na serwerze produkcyjnym (wersja serwerowa). Przypuśćmy, że zapomnieliśmy zadeklarować zmiennej `asleep` jako `volatile` z listingu 3.4. Wersja serwerowa usunie test z pętli (powstanie pętla nieskończona), ale tego kroku nie uczyni wersja kliencka. Pętla nieskończona występująca w trakcie testów jest mniej kosztowna od tej pojawiającej się tylko w wersji produkcyjnej.

Przykładowo semantyka modyfikatora `volatile` nie jest na tyle silna, by zagwarantować niepodzielność operacji inkrementacji (`count++`), chyba że można zagwarantować zapis zmiennej tylko przez jeden wątek. Zmienne niepodzielne zapewniają nierozłączną obsługę operacji odczyt, modyfikacja, zapis, więc często można je stosować jako „lepsze wersje zmiennych ulotnych"; więcej informacji na ten temat w rozdziale 15.).

> Blokada gwarantuje widoczność i niepodzielność. Zmienna ulotna może zagwarantować co najwyżej widoczność.

Zmienne ulotne stosuj tylko wtedy, gdy spełnione są wszystkie poniższe kryteria:

- zapis zmiennej nie zależy od jej aktualnej wartości lub gdy można zapewnić zapis aktualizacji tylko przez jeden wątek,
- zmienna nie jest składową niezmiennika obejmującego inne zmienne stanowe,
- blokowanie nie jest potrzebne z innych powodów w trakcie dostępu do zmiennej.

3.2. Publikacja i ucieczka

Publikacja obiektu oznacza jego udostępnienie kodowi spoza jego aktualnego zasięgu, na przykład przez zapamiętanie referencji do niego, by mógł z niego skorzystać inny kod, zwrócenie go z nieprywatnej metody lub przekazanie jako argument metody innej klasy. W większości sytuacji chcemy mieć pewność, że obiekty i ich wewnętrzne dane **nie** są publikowane. W innych przypadkach publikujemy obiekt w celu ogólnego użycia — wykonanie tego zadania w sposób bezpieczny wątkowo wymaga synchronizacji. Publikacja zmiennych z wewnętrznym stanem obiektu szkodzi hermetyzacji. Publikacja obiektu przed ich całkowitym utworzeniem zmniejsza bezpieczeństwo aplikacji. Publikację obiektu, gdy nie powinno to mieć miejsca, nazywa się **ucieczką** lub **wyciekiem**. Podrozdział 3.5 omawia idiomy bezpiecznej publikacji. Na razie przyjrzymy się sposobom ucieczki obiektu.

Najbardziej ewidentna forma publikacji polega na zapamiętaniu referencji w publicznym i statycznym polu, z którego może skorzystać dowolna klasa lub wątek (patrz listing 3.5). Metoda `initialize()` inicjalizuje nowy obiekt `HashSet` i publikuje go w zmiennej `knownSecrets`.

Listing 3.5. *Publikacja obiektu*

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

Publikacja jednego obiektu może pośrednio opublikować inne. Dodanie obiektu `Secret` do `knownSecrets` spowoduje publikację tego obiektu, bo każdy kod może przejść przez zbiór i uzyskać referencję do `Secret`. Zwrócenie referencji z nieprywatnej metody

również publikuje zwrócony obiekt. Klasa `UnsafeStates` z listingu 3.6 publikuje tablicę skrótów stanów, która być może powinna pozostać prywatna.

Listing 3.6. *Umożliwia ucieczkę wewnętrznego, zmiennego stanu. Nie rób tak*

```
class UnsafeStates {
    private String[] states = new String[]{
        "AK", "AL" ...
    };

    public String[] getStates() { return states; }
}
```

Publikacja `states` w ten sposób jest problematyczna, bo dowolny kod może zmienić jej zawartość. W przedstawionej sytuacji tablica `states` uciekła z jej domyślnego zakresu, bo to, co miało pozostać prywatne, tak naprawdę zostało upublicznione.

Publikacja obiektów publikuje wszystkie inne obiekty znajdujące się w jego nieprywatnych polach. Bardziej ogólnie: dowolny obiekt osiągalny z poziomu opublikowanego obiektu przez łańcuch nieprywatnych referencji i wywołań metod również został opublikowany.

Z perspektywy klasy C metoda **obca** to taka, której zachowanie nie zostało w pełni określone w C. Dotyczy to metod w innych klasach oraz metod przysłanianych (różnych od `private` i `final`) w samej klasie C. Przekazanie obiektu do metody obcej należy również traktować jako publikację tego obiektu. Nie wiemy tak naprawdę, jak zachowa się zewnętrzny kod — nie musi, ale może przekazać opublikowany obiekt innym obiektom, być może działającym w obrębie innych wątków.

Co tak naprawdę inny wątek zrobi z opublikowaną referencją do obiektu, nie ma znaczenia, bo cały czas istnieje ryzyko jego niepoprawnego użycia[7]. Ucieczka obiektu oznacza, że musimy założyć, iż inna klasa lub wątek specjalnie lub przypadkowo źle go użyje. To bardzo ważny powód przemawiający za hermetyzacją, bo ułatwia analizę programów pod kątem poprawności i utrudnia przypadkowe złamanie ograniczeń projektowych.

Ostatnim mechanizmem, dzięki któremu obiekt lub jego wewnętrzny stan mogą zostać opublikowane, jest publikacja egzemplarza klasy wewnętrznej. Przedstawia to klasa `ThisEscape` z listingu 3.7. Gdy `ThisEscape` publikuje obiekt `EventListener`, niejawnie publikuje również egzemplarz `ThisEscape`, bo egzemplarz klasy wewnętrznej zawiera ukrytą referencję do swego zewnętrznego kolegi.

Listing 3.7. *Niejawna ucieczka referencji this. Nie rób tak*

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
```

[7] Jeśli ktoś ukradnie Twoje hasło i umieści je na liście dyskusyjnej alt.free-passwords, informacja ta po prostu wyciekła. Nie ma znaczenia, czy ktoś rzeczywiście użył tego hasła w celu oszustwa. Publikacja referencji stwarza podobne ryzyko.

```
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }
}
```

3.2.1. Tworzenie bezpiecznych konstrukcji

Klasa ThisEscape ilustruje bardzo ważny szczególny przypadek ucieczki, bo w trakcie konstrukcji klasy ucieka referencja this. Opublikowanie wewnętrznego egzemplarza EventListener publikuje również egzemplarz ThisEscape. Obiekt znajdzie się w przewidywalnym, spójnym stanie dopiero po powrocie z konstruktora, więc ucieczka już we wnętrzu konstruktora może spowodować publikację niedokończonego obiektu. Sytuacja obejmuje również przypadek, w którym **publikacja to ostatnia instrukcja konstruktora**. Jeśli referencja this wycieknie w trakcie konstrukcji obiektu, obiekt traktuje się jako niepoprawnie skonstruowany[8].

Nie dopuszczaj do konstrukcji ucieczki referencji this w konstruktorze obiektu.

Typowym błędem umożliwiającym ucieczkę referencji this z konstruktora jest uruchamianie wątku we wnętrzu konstruktora. Gdy obiekt tworzy wątek w swym konstruktorze, niemal zawsze współdzieli z nim referencję this, czy to jawnie (przez przekazanie jej do konstruktora), czy niejawnie (ponieważ Thread i Runnable to klasy wewnętrzne obiektu). Nowy wątek może widzieć obiekt, którego część stanowi, przed pełną inicjalizacją. W zasadzie nie ma nic złego w **utworzeniu** wątku w konstruktorze — po prostu nie należy go od razu **uruchamiać**. Zamiast tego wystarczy udostępnić metodę start() lub initialize() uruchamiającą wewnętrzny wątek. Więcej informacji na temat cyklu życia usług znajduje się w rozdziale 7. Wywołanie przesłoniętej metody egzemplarza (czyli takiej, która nie jest prywatna ani ostateczna) z poziomu konstruktora również dopuszcza wyciek referencji this.

Gdy próbuje się zarejestrować nasłuchiwanie zdarzeń lub uruchomić wątek w konstruktorze, warto zapewnić poprawną konstrukcję, używając prywatnego konstruktora i publicznej metody fabrycznej, co przedstawia klasa SafeListener z listingu 3.8.

Listing 3.8. *Użycie metody fabrycznej w celu uniknięcia wycieku referencji this w trakcie konstrukcji obiektu*

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
```

[8] Uściślijmy: referencja this nie może wyciec z **wątku** przed powrotem z konstruktora. Referencję this można przechowywać w innym miejscu, jeśli tylko nie zostanie **wykorzystana** przez inny wątek przed zakończeniem konstrukcji. Listing 3.8 przedstawia poprawioną wersję.

```
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

3.3. Odosobnienie w wątku

Dostęp do współdzielonych zasobów wymaga synchronizacji. Jednym ze sposobów uniknięcia tego problemu jest brak współdzielenia. Jeśli dane są dostępne tylko z poziomu jednego wątku, nie potrzeba synchronizacji. Technika ta, zwana odosabnianiem wątku, jest jednym z najprostszych sposobów osiągnięcia bezpieczeństwa pod kątem wątków. Jeśli obiekt jest ograniczony do jednego wątku, jego użycie jest bezpieczne, nawet jeśli sam obiekt nie zawiera żadnych zabezpieczeń [CJP 2.3.2].

Swing intensywnie korzysta z odosabniania wątku. Komponenty wizualne i modele danych Swing nie są zabezpieczone pod kątem wątków. Ich bezpieczeństwo zapewnia specjalny wątek rozdzielania zadań. Aby poprawnie korzystać ze Swinga, kod uruchamiany z wątków innych niż wątek rozdzielający nie powinien mieć dostępu do tych obiektów. By ułatwić to zadanie, Swing dostarcza mechanizm `invokeLater`, który harmonogramuje obiekt `Runnable` w celu jego wykonania w wątku rozdzielającym. Wiele błędów współbieżności w aplikacjach Swing wynika właśnie z niepoprawnego użycia tych ukrytych obiektów z innego wątku.

Innym popularnym zastosowaniem odosabniania wątków jest użycie puli obiektów `Connection` z JDBC. Specyfikacja JDBC nie wymaga, by obiekty `Connection` były bezpieczne wątkowo[9]. W typowej aplikacji serwerowej wątek pobiera połączenie z puli, używa go do wykonania pojedynczego zapytania i zwraca obiekt do puli. Ponieważ większość żądań, wywołania serwletów i EJB, przetwarzana jest synchronicznie przez jeden wątek i pula nie przekaże tego samego połączenia innemu wątkowi, dopóki pierwszy go nie zwróci, wzorzec ten zapewnia niejawne odosobnienie obiektu `Connection` tylko do jednego wątku na czas wykonania żądania.

Podobnie jak język nie zawiera mechanizmu wymuszającego stosowanie blokady do zabezpieczania zmiennych, tak nie ma żadnego konkretnego sposobu ograniczenia stosowania obiektu tylko do jednego wątku. Odosobnienie w wątku to element projektu programu, który musi wymusić sama jego implementacja. Język i główne biblioteki

[9] Pule połączeń udostępniane przez serwery aplikacji są bezpieczne wątkowo; pule połączeń niemal zawsze są wykorzystywane przez wiele wątków, więc ich niezabezpieczanie nie miałoby sensu.

pomagają osiągnąć ten cel — zmienne lokalne i klasa `ThreadLocal` — ale nawet z nimi programista jest w pełni odpowiedzialny za to, by odosobnione w wątku obiekty nie uciekły do innych wątków.

3.3.1. Odosabnianie w wątku typu ad-hoc

Odosabnianie w wątku typu ad-hoc polega na zastosowaniu tylko i wyłączenie implementacyjnych mechanizmów zapewnienia odosobnienia. To podejście jest dosyć kruche, bo żadna z cech języka, na przykład widoczność modyfikatorów i zmienne lokalne, nie pomaga utrzymać obiektu tylko przy jednym wątku. W zasadzie referencje do obiektów odosobnionych w wątkach, na przykład komponentów wizualnych i modeli danych w aplikacjach graficznych, są często przechowywane w polach publicznych.

Decyzja dotycząca zastosowania odosobnienia wynika najczęściej z decyzji o implementacji konkretnego podsystemu, na przykład interfejsu graficznego, jako podsystemu jednowątkowego. Takie podsystemy niejednokrotnie oferują znaczące uproszczenie, które okazuje się znacznie bardziej kuszące od kruchości odosabniania typu ad-hoc[10].

Specjalny przypadek odosabniania wątku dotyczy zmiennych ulotnych. Na współdzielonej zmiennej ulotnej operacje odczyt, modyfikacja, zapis można przeprowadzić tylko wtedy, gdy ma się pewność, że zapis będzie odbywał się wyłącznie w jednym wątku. W tej sytuacji zapewniamy odosobnienie modyfikacji w jednym wątku, by uniknąć wyścigu. Wtedy ulotność gwarantuje, że pozostałe wątki zawsze będą widziały najbardziej aktualną wartość zmiennej.

Z powodu swej kruchości warto oszczędnie stosować odosabnianie typu ad-hoc i gdy tylko to możliwe, używać silniejszych form ograniczania (wykorzystując stos lub obiekty `ThreadLocal`).

3.3.2. Odosobnienie na stosie

Odosobnienie na stosie to szczególny przypadek odosobnienia w wątku, gdy obiekt jest dostępny tylko przy użyciu zmiennych lokalnych. Podobnie jak hermetyzacja ułatwia utrzymanie niezmienników, tak zmienne lokalne ułatwiają zatrzymanie obiektów w jednym wątku. Zmienne lokalne są same z siebie ograniczone do wątku wykonującego zadanie, bo istnieją na jego stosie, który nie jest dostępny w innych wątkach. Ten sposób odosabniania, nazywany również **wewnątrzwątkowym**, jest prostszy od utrzymania i mniej kruchy niż rozwiązanie ad-hoc.

Zmienne lokalne typów prostych, na przykład `numPairs` z metody `loadAndArk()` z listingu 3.9, uniemożliwiają złamanie odosobnienia na stosie, nawet gdyby chciałoby się je wykonać. Nie można uzyskać referencji do zmiennej typu prostego, więc sama semantyka języka ogranicza zmienne lokalne typów prostych do jednego wątku.

---

[10] Kolejnym powodem stosowania podsystemu jednowątkowego bywa chęć uniknięcia blokad wzajemnych; to jeden z głównych powodów, dla których większość systemów graficznych jest jednowątkowa. Systemy jednowątkowe opisuje rozdział 9.

Listing 3.9. *Odosobnienie zmiennych lokalnych typów prostych i referencyjnych*

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals ograniczony do metody, więc nie pozwól mu uciec!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

Odosobnienie na stosie referencji do obiektów wymaga już nieco uwagi ze strony programisty, by referencja nie uciekła. W metodzie `loadAndArk()` tworzymy obiekt `TreeSet` i zapamiętujemy w nim referencję do `animals`. W tym momencie istnieje tylko jedna referencja do zbioru przechowywana w zmiennej lokalnej, więc jest ograniczona jedynie do wątku wykonującego metodę. Jeśli jednak chcemy opublikować referencję do zbioru (lub któregokolwiek z jej elementów), odosobnienie przestanie istnieć i referencja `animals` może uciec.

Użycie obiektu niezabezpieczonego wątkowo wewnątrz kontekstu tylko jednego wątku jest w pełni bezpieczne. Uważaj: wymagania projektowe, by obiekt był ograniczony tylko do wątku wykonującego lub wiedza o tym, że obiekt ten nie jest bezpieczny wątkowo, istnieje najczęściej tylko w głowach programistów piszących kod. Jeśli założenie ograniczenia użycia obiektu tylko do jednego wątku nie zostało jawnie określone w dokumentacji, kolejni programiści mogą dopuścić do ucieczki referencji do obiektu.

3.3.3. Obiekt ThreadLocal

Bardziej formalnym sposobem uzyskania odosobnienia w wątku jest obiekt `ThreadLocal`, który umożliwia przechowywanie wartości powiązanej z konkretnym wątkiem. Obiekt zawiera metody dostępowe ułatwiające przechowywanie osobnej kopii wartości dla każdego wątku, który korzysta z kodu. Oznacza to, że metoda `get()` zawsze zwróci najnowszą wartość ustawioną metodą `set()` z poziomu aktualnie wykonywanego wątku.

Zmienne lokalnowątkowe często służą uniknięciu współdzielenia w projektach bazujących na zmiennych singletonach i zmiennych globalnych. Przykładowo aplikacja jednowątkowa może przechowywać globalny obiekt połączenia bazodanowego inicjalizowany w momencie uruchamiania programu, by uniknąć przekazywania obiektu `Connection` do każdej metody. Ponieważ połączenia JDBC mogą nie być bezpieczne

wątkowo, aplikacja wielowątkowa używająca globalnego połączenia bez żadnej dodatkowej koordynacji nie jest odpowiednio zabezpieczona. Używając obiektu `ThreadLocal` do przechowywania połączenia JDBC (patrz listing 3.10), każdy wątek uzyskuje własne połączenie bazodanowe.

Listing 3.10. *Użycie ThreadLocal do zapewnienia odosobnienia w wątku*

```
private ThreadLocal<Connection> connectionHolder
        = new ThreadLocal<Connection>() {
            public Connection initialValue() {
                return DriverManager.getConnection(DB_URL);
            };
        };

public Connection getConnection() {
    return connectionHolder.get();
}
```

Technikę tę można zastosować również wtedy, gdy często wykonywana operacja wymaga obiektu tymczasowego, na przykład bufora, i chce się uniknąć każdorazowej alokacji pamięci dla niego. Na przykład przed Javą 5.0 metoda `Integer.toString()` używała obiektu `ThreadLocal` do przechowywania 12-bajtowego bufora używanego do formatowania wyniku zamiast współdzielonego statycznego bufora (który wymagałby synchronizacji) i alokacji nowego bufora w każdym wywołaniu[11].

Gdy wątek wywołuje `ThreadLocal.get()` po raz pierwszy, `initialValue` zostaje wypełniona wartością domyślną dla tego wątku. W zasadzie można traktować `ThreadLocal<T>` jak odwzorowanie `Map<Thread,T>`, które przechowuje specyficzną wartość (w rzeczywistości implementacja klasy jest inna). Wartości specyficzne dla wątku znajdują się w samym obiekcie `Thread`, więc w momencie jego zakończenia są usuwane razem z wątkiem.

Przenosząc aplikację jednowątkową do środowiska wielowątkowego, można zachować bezpieczeństwo wątkowe, konwertując współdzielone zmienne globalne na zmienne lokalnowątkowe, jeśli tylko dopuszcza to sposób używania zmiennych globalnych. Bufor stosowany przez całą aplikację przestałby być szczególnie użyteczny, gdyby jego niezależne kopie stosował każdy wątek.

Obiekt `ThreadLocal` często pojawia się w szkieletach aplikacji. Przykładowo kontener J2EE dołącza kontekst transakcyjny do wykonywanego wątku na czas trwania wywołania EJB. Łatwo to zaimplementować, używając statycznego obiektu `ThreadLocal` przechowującego kontekst transakcyjny. Gdy kod szkieletu chce się dowiedzieć, jaka transakcja obecnie obowiązuje, sprawdza kontekst transakcyjny zawarty w `ThreadLocal`. Podejście jest wygodne, bo nie wymaga przekazywania kontekstu do każdej metody, ale łączy cały kod, który korzysta z tego mechanizmu, ze szkieletem.

---

[11] Technika ta rzadko okazuje się wydajniejsza, chyba że operację wykonuje się niezwykle często, a alokacja jest kosztowna. W Javie 5.0 zastosowano prostsze podejście z buforem alokowanym w każdym wywołaniu, wskazując na to, że dla tak niewielkiego bufora nie warto stosować bardziej zaawansowanego rozwiązania, bo nie przynosi żadnych korzyści w wydajności.

Łatwo można nadużywać `ThreadLocal`, traktując jego właściwości hermetyzujące jako licencję na stosowanie zmiennych globalnych lub sposób tworzenia „ukrytych" argumentów metod. Podobnie jak zmienne globalne, również zmienne lokalnowątkowe mogą utrudniać wielokrotne wykorzystanie i wprowadzać ukryte zależności między klasami. Z tego względu warto podchodzić do nich ostrożnie.

3.4. Niezmienność

Kolejny sposób obejścia potrzeby synchronizacji polega na użyciu obiektów **niezmiennych** [EJ Item 13]. Niemal wszystkie hazardy widoczności i niepodzielności opisywane do tej pory — widoczność nieświeżych danych, utrata aktualizacji, obserwacja obiektu w niespójnym stanie — dotyczą sytuacji, w których wiele wątków stara się uzyskać w tym samym czasie dostęp do zmiennego stanu. Jeżeli stan obiektu w ogóle się nie zmienia, wszystkie wskazane ryzyka i komplikacje znikają same.

Obiekt niezmienny to taki, którego stanu nie można zmienić po jego konstrukcji. Obiekty takie są bezpieczne pod kątem wątków; niezmienniki zostają ustawione w konstruktorze, a ponieważ stan obiektu się nie zmienia, niezmienniki zawsze są poprawne.

> Obiekty niezmienne zawsze są bezpieczne pod kątem wątków.

Obiekty niezmienne są **proste**. Mogą znajdować się tylko w jednym stanie wyliczonym w konstruktorze. Jednym z trudniejszych aspektów projektowania programu jest wskazywanie możliwych stanów złożonych obiektów. Określanie stanu obiektu niezmiennego nie sprawia najmniejszych trudności.

Obiekty niezmienne są też **bezpieczniejsze**. Przekazanie zmiennego obiektu do nieznanego kodu (lub jego publikacja w taki sposób, że może go odnaleźć nieznany kod) bywa niebezpieczne — nieznany kod może zmodyfikować stan obiektu lub, co gorsza, zapamiętać referencję do obiektu i zmienić go w innym czasie, używając innego wątku. Z drugiej strony, obiektu niezmiennego nie uszkodzi niebezpieczny ani błędny kod, więc jest bezpieczny i może być publikowany bez stosowania dodatkowych zabezpieczeń [EJ Item 24].

Ani specyfikacja języka Java, ani model pamięci Javy nie definiuje niezmienności w sposób formalny. **Nie jest** ona równoważna prostemu zadeklarowaniu wszystkich pól obiektu jako `final`. Obiekt ze wszystkimi polami typu `final` nadal może być zmienny, bo nic nie stoi na przeszkodzie, by zawierał referencje do zmiennych obiektów.

Obiekt niezmienny nadal może używać wewnętrznie obiektów zmiennych do zarządzania własnym stanem, co ilustruje klasa `ThreeStooges` z listingu 3.11. Choć obiekt `Set` przechowujący imiona jest zmienny, konstrukcja głównej klasy uniemożliwia modyfikację zbioru po jego utworzeniu. Referencja `stooges` jest typu `final`, więc cały stan obiektu udostępnia pola tego typu. Ostatni wymóg (poprawnego utworzenia) został spełniony, bo konstruktor w żaden sposób nie może spowodować wycieku referencji `this`.

Listing 3.11. *Klasa niezmienna powstała na podstawie zmiennego obiektu*

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

Obiekt jest **niezmienny**, jeśli:

- jego stanu nie można zmienić po zakończeniu działania konstruktora,
- wszystkie jego pola są typu `final`[12],
- jest poprawnie utworzony (referencja `this` nie ucieknie w trakcie konstrukcji).

Ponieważ stan programu zmienia się cały czas, wydawać by się mogło, że zastosowania obiektów niezmiennych są mocno ograniczone, ale w rzeczywistości tak nie jest. Istnieje bardzo poważna różnica między niezmiennym **obiektem**, a niezmienną **referencją** do obiektu. Stan programu przechowywany w niezmiennych obiektach można uaktualnić, „zastępując" stary obiekt nową wersją z nowym stanem. Kolejny podrozdział zawiera przykład tej techniki[13].

3.4.1. Pola typu final

Słowo kluczowe `final`, bardziej ograniczona wersja mechanizmu `const` z C++, obsługuje konstrukcję obiektów niezmiennych. Pól finalnych (ostatecznych) nie można modyfikować (choć obiekty, do których zawierają referencje, mogą się zmieniać). Co więcej, mają one specjalne znaczenie w modelu pamięci Javy. To użycie pól finalnych gwarantuje **bezpieczeństwo inicjalizacyjne** (patrz punkt 3.5.2), które umożliwia swobodny dostęp i współdzielenie obiektów niezmiennych.

[12] Technicznie możliwe jest uzyskanie obiektu niezmiennego bez wszystkich pól ustawionych na `final` — przykładem jest klasa `String` — ale wykorzystuje to bardzo delikatne uwarunkowania wykluczające wyścigi i wymaga dobrego zrozumienia modelu pamięci Javy. Dla ciekawskich: klasa `String` leniwie wylicza skrót tekstu przy pierwszym wywołaniu metody `hashCode()` i buforuje ją w polu niefinalnym. Wszystko działa poprawnie jedynie dlatego, że pole może uzyskać tylko jedną niedomyślną wartość, która zawsze jest taka sama, bo zostaje wyliczona na podstawie niezmiennego stanu (nie próbuj tego w domu).

[13] Wielu programistów obawia się, że to podejście powoduje problemy z wydajnością. W wielu sytuacjach są one bezpodstawne. Alokacja jest tańsza, niż może się wydawać, a obiekty niezmienne zwiększają wydajność, bo nie potrzebują blokad czy kopiowania defensywnego. Mają ograniczony wpływ na szybkość mechanizmu odzyskiwania pamięci.

Nawet jeśli obiekt jest zmienny, określenie kilku jego pól jako finalnych ułatwia analizę jego stanu, bo ograniczenie zmienności niektórych pól zmniejsza liczbę kombinacji stanu obiektu. Obiekt, który jest w „większości niezmienny”, ale ma jedno lub dwa zmienne pola, okazuje się łatwiejszy do analizy niż obiekt z wieloma zmiennymi polami. Dodatkowo deklaracja pola jako `final` informuje innych programistów, że wskazany element nie będzie się zmieniał.

> Podobnie jak zaleca się, by wszystkie pola określać jako prywatne, jeśli nie wymagają większej widoczności [EJ Item 12], zaleca się też oznaczanie wszystkich pól niezmieniających swego stanu modyfikatorem `final`.

3.4.2. Przykład — użycie volatile do publikacji obiektów niezmiennych

W klasie `UnsafeCachingFactorizer` ze strony 36. staraliśmy się użyć dwóch obiektów `AtomicReference` do zapamiętania ostatniej wartości i rozkładu, ale to podejście nie okazywało się bezpieczne wątkowo, bo niemożliwe było jednoczesne pobranie lub uaktualnienie obu wartości. Użycie zmiennych ulotnych również nie rozwiązałoby problemu. Z drugiej strony obiekty niezmienne potrafią czasem zapewnić słabą formę niepodzielności.

Serwlet wyliczający rozkład na czynniki wykonuje dwie operacje niepodzielne: aktualizację bufora i warunek sprawdzający, czy bufor zawiera wartość, którą chcemy wyliczyć. Gdy grupa powiązanych danych musi działać w sposób niepodzielny, warto rozważyć utworzenie dla nich klasy stanu niezmiennego, na przykład `OneValueCache`[14] z listingu 3.12.

Listing 3.12. *Niezmienny obiekt przechowujący buforowaną wartość i jej rozkład*

```
@Immutable
public class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
```

[14] Klasa `OneValueCopy` nie byłaby niezmienna, gdyby nie metody pobierające i wywołania `copyOf()`. Metoda `Arrays.copyOf()` pojawia się w Javie 6, ale we wcześniejszych wersjach można użyć metody `clone()`.

```
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

Wyścig dotyczący dostępu lub aktualizacji wielu powiązanych zmiennych udaje się wyeliminować, używając obiektu niezmiennego przechowującego wszystkie zmienne. Ze zmiennym obiektem trzeba stosować blokady, by zapewnić niepodzielność; wykorzystując niezmienny obiekt, wątek uzyskuje do niego dostęp, nie martwiąc się o inny wątek modyfikujący jego stan. Jeśli konieczne staje się uaktualnienie zmiennych, powstaje nowy obiekt stały, ale inne wątki (widzące starszą wersję) nadal mają dostęp do spójnego stanu.

Klasa `VolatileCachedFactorizer` z listingu 3.13 używa klasy `OneValueCache` do zapamiętania wartości i rozkładu na czynniki. Jeśli wątek ustawia ulotne pole `cache` na referencję do obiektu `OneValueCache`, nowe dane od razu widzą inne wątki.

Listing 3.13. *Buforowanie ostatniego wyniku w referencji ulotnej dotyczącej niezmiennego obiektu*

```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache = new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
```

Operacje dotyczące bufora nie interferują między sobą, bo klasa `OneValueCache` jest niezmienna, a pole `cache` za każdym razem jest udostępniane tylko raz w każdej z istotnych ścieżek. To połączenie kilku wartości powiązanych niezmiennikiem w jednym niezmiennym obiekcie oraz użycie referencji typu `volatile` zapewnia klasie `VolatileCachedFactorizer` bezpieczeństwo wątkowe, choć w ogóle nie stosujemy jawnych blokad.

3.5. Bezpieczna publikacja

Do tej pory skupialiśmy się na zapewnieniu braku publikacji obiektu, czyli zamknięciu w jednym wątku lub wewnątrz innego obiektu. Oczywiście są sytuacje, w których **chcemy** współdzielić obiekty między wątkami — wtedy musimy zatroszczyć się o bezpieczeństwo. Niestety, samo umieszczenie referencji do obiektu w polu publicznym, patrz listing 3.14, **nie wystarcza** do bezpiecznej publikacji obiektu.

Listing 3.14. *Publikacja obiektu bez odpowiedniej synchronizacji. Nie rób tak*

```
// Niezabezpieczona publikacja.
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```

Zadziwiające, jak ten niewinny przykładowy kod potrafi zaszkodzić aplikacji. Z powodu problemów z widocznością obiekt `Holder` może w innym wątku pojawić się w niespójnym stanie, nawet jeśli jego niezmienniki zostały poprawnie ustawione w konstruktorze! Niepoprawna publikacja umożliwia innemu wątkowi zaobserwowanie **częściowo skonstruowanego obiektu**.

3.5.1. Nieodpowiednia publikacja — gdy dobre obiekty idą w złą stronę

Nie warto polegać na integralności częściowo skonstruowanego obiektu. Wątek obserwujący spostrzeże obiekt w niespójnym stanie, a następnie zauważy, że jego stan nagle uległ zmianie, choć tak naprawdę nie był on modyfikowany od czasu publikacji. W rzeczywistości jeśli obiekt `Holder` z listingu 3.15 zostałby opublikowany w sposób niezabezpieczony (patrz listing 3.14), wątki inne niż publikujący po wywołaniu metody `assertSanity()` mogłyby otrzymać wyjątek `AssertionError`![15]

Listing 3.15. *Klasa ryzykująca błąd, jeśli nie zostanie poprawnie opublikowana*

```
public class Holder {
    private int n;

    public Holder(int n) { this.n = n; }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("Warunek jest prawdziwy.");
    }
}
```

Ponieważ nie użyliśmy synchronizacji do uwidocznienia obiektu `Holder` innym wątkom, mówimy o **niepoprawnej publikacji**. Niepoprawnie opublikowane obiekty narażamy na dwie przypadłości. Inne wątki mogą zauważyć nieświeżą wartość w polu `holder`, a tym samym zobaczyć referencję `null` lub inną wartość, choć w rzeczywistości została ona ustawiona. Co gorsza, mogą uzyskać poprawną referencję do obiektu `Holder`, ale

[15] Przedstawiany problem nie dotyczy klasy `Holder` jako takiej, ale sposobu upublicznienia jej obiektów. Klasę można zabezpieczyć przed niepoprawną publikacją, deklarując pole n jako `final`, bo wtedy obiekty klasy będą niezmienne, patrz punkt 3.5.2.

odczytać z niego nieświeży stan[16]. Aby jeszcze bardziej udziwnić sytuację, wątek może za pierwszym razem odczytać nieaktualną wartość z wątku, a za drugim razem przeczytać wartość aktualną. Właśnie z tego powodu test z metody `assertSanity()` może się okazać prawdziwy i spowodować zgłoszenie wyjątku `AssertionError`.

Choć ryzykujemy powtarzanie się, przypomnijmy, iż bardzo dziwne rzeczy mogą się dziać, jeśli data jest współdzielona przez wiele wątków bez należytej synchronizacji.

3.5.2. Obiekty niezmienne i bezpieczeństwo inicjalizacji

Ponieważ obiekty niezmienne są tak ważne, model pamięci Javy oferuje specjalną gwarancję bezpieczeństwa inicjalizacji współdzielonych obiektów niezmiennych. Przekonaliśmy się, że gdy referencja do obiektu staje się widoczna dla innych wątków, nie oznacza to jednocześnie, że to samo dzieje się ze stanem obiektu. Aby zapewnić spójny widok stanu obiektu, potrzebujemy synchronizacji.

Z drugiej strony, obiekty niezmienne można udostępniać bezpiecznie nawet wtedy, gdy **synchronizacja nie zabezpiecza publikacji referencji do obiektu**. Aby zagwarantować to bezpieczeństwo inicjalizacji, należy spełnić wszystkie wymogi niezmienności: niemodyfikowalny stan, wszystkie pola ustawione na `final` i odpowiednią konstrukcję (gdyby klasa `Holder` z listingu 3.15 była niezmienna, metoda `assertSanity()` nie mogłaby zgłosić wyjątku nawet w momencie nieodpowiedniej publikacji).

> Obiekty niezmienne mogą być bezpiecznie stosowane przez dowolne wątki bez dodatkowej synchronizacji, nawet jeśli synchronizacja nie służy do jej publikacji.

Ta gwarancja dotyczy wartości wszystkich pól finalnych poprawnie skonstruowanych obiektów. Pola finalne są dostępne w sposób bezpieczny bez dodatkowej synchronizacji. Jeśli jednak dotyczą one zmiennych obiektów, dostęp do stanu tych obiektów nadal wymaga synchronizacji.

3.5.3. Idiomy bezpiecznej synchronizacji

Obiekty, które nie są niezmienne, muszą zostać opublikowane w sposób bezpieczny, co najczęściej oznacza synchronizację zarówno przez wątek tworzący, jak i konsumujący. Na razie skupmy się na zapewnieniu, by wątek konsumujący zawsze widział stan obiektu po publikacji. Dopiero później zajmiemy się widocznością zmian po publikacji.

---

[16] Choć może się wydawać, że wartości podane w konstruktorze są pierwszymi zapisywanymi w polach, w rzeczywistości jest inaczej. Konstruktor `Object` przed uruchomieniem właściwego konstruktora ustawia wszystkie zmienne na wartości domyślne. Kod wątku może odczytać domyślną wartość, która tak naprawdę jest przestarzała.

Aby bezpiecznie opublikować obiekt, zarówno referencja do obiektu, jak i jego stan muszą być widoczne dla innych wątków dokładnie w tym samym momencie. Poprawnie skonstruowany obiekt bezpiecznie publikuj przez:

- inicjalizację referencji do obiektu z poziomu elementu `static`,
- przechowywanie referencji w polu `volatile` lub obiekcie `AtomicReference`,
- przechowywanie referencji w polu `final` poprawnie utworzonego obiektu,
- przechowywanie referencji w polu poprawnie chronionym blokadą.

Wewnętrzna synchronizacja w kolekcjach zabezpieczonych wątkowo oznacza, że umieszczanie obiektu w takiej kolekcji (na przykład klasie `Vector` lub `synchronizedList`) spełnia ostatni z wymienionych wymogów. Jeśli wątek A umieszcza obiekt X w kolekcji zabezpieczonej wątkowo i wątek B stara się ją odczytać, gwarantuje się, że B otrzyma stan X w takiej wersji, w jakiej zostawił go A, mimo że kod obsługujący X nie stosuje żadnej **jawnej** synchronizacji. Biblioteka kolekcji zabezpieczonych wątkowo oferuje następujące gwarancje bezpieczeństwa publikacji (choć dokumentacja nie zawsze wskazuje je dostatecznie mocno):

- Umieszczenie klucza lub wartości w `Hashtable`, `synchronizedMap` lub `ConcurrentMap` bezpiecznie publikuje ten element dowolnemu wątkowi, który pobiera go z `Map` (bezpośrednio lub za pomocą iteratora).
- Umieszczenie elementu w `Vector`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `synchronizedList` lub `synchronizedSet` bezpiecznie publikuje go dowolnemu wątkowi, który pobiera go z kolekcji.
- Umieszczenie elementu w `BlockingQueue` lub `ConcurrentLinkedQueue` bezpiecznie publikuje go dowolnemu wątkowi, który pobiera go z kolejki.

Inne mechanizmy biblioteki klas (na przykład `Future` i `Exchanger`) również wprowadzają bezpieczeństwo publikacji. Zajmiemy się ich zaletami w momencie ich wprowadzania.

Inicjalizacja statyczna to często najprostszy i najbezpieczniejszy sposób publikacji obiektu, który może być wykonany statycznie.

```
public static Holder holder = new Holder(42);
```

Inicjalizacja statyczna zachodzi w maszynie wirtualnej w momencie wczytywania klasy. Z powodu wewnętrznej synchronizacji w maszynie wirtualnej mechanizm ten gwarantuje bezpieczną publikację obiektu [JLS 12.4.2].

3.5.4. Obiekty technicznie niezmienne

Bezpieczna publikacja wystarcza innym wątkom do bezpiecznego dostępu bez synchronizacji do obiektów, które nie będą modyfikowane po publikacji. Mechanizm bezpiecznej publikacji gwarantuje, że opublikowany stan obiektu będzie widziany przez wszystkie referencje w momencie udostępnienia im referencji do obiektu. Jeśli tylko stan ten nie będzie ulegał zmianom, wystarcza to, by dostęp do obiektu był bezpieczny.

Obiekty, które w sposób formalny nie są niezmienne, ale których stanu nie można zmienić po opublikowaniu, nazywamy **niezmiennymi technicznie**. Nie muszą spełniać ścisłej definicji niezmienności z podrozdziału 3.4 — wystarczy że są przez program traktowane tak, jakby rzeczywiście były niezmienne po publikacji. Wykorzystanie obiektów niezmiennych technicznie upraszcza implementację i poprawia wydajność przez redukcję potrzeby synchronizacji.

> Bezpiecznie opublikowane obiekty **niezmienne technicznie** mogą być bezpiecznie używane przez dowolny wątek bez dodatkowej synchronizacji.

Przykładowo obiekt `Date` jest zmienny[17], ale gdy będziemy go stosować tak, jakby był niezmienny, nie musimy wprowadzać blokad, które w przeciwnym razie byłyby potrzebne do odpowiedniego współdzielenia obiektu i synchronizacji. Przypuśćmy, że chcemy przechowywać obiekt `Map` zawierający daty ostatniego logowania użytkowników.

```
public Map<String, Date> lastLogin =
    Collections.synchronizedMap(new HashMap<String, Date>());
```

Jeśli wartości `Date` nie są modyfikowane po ich umieszczeniu w `Map`, wtedy synchronizacja zapewniana przez implementację `synchronizedMap` wystarcza do poprawnej publikacji obiektu `Date` i nie jest potrzebna żadna dodatkowa synchronizacja w momencie dostępu do niej.

3.5.5. Obiekty zmienne

Jeśli obiekt może zmieniać się po utworzeniu, bezpieczna publikacja zapewnia jedynie jego widoczność w stanie z momentu publikacji. Synchronizację trzeba wtedy stosować nie tylko w momencie publikacji obiektu, ale również przy każdym dostępie do niego, by w ten sposób zapewnić widoczność kolejnych modyfikacji. Bezpieczne współdzielenie obiektu zmiennego wymaga bezpiecznej publikacji **oraz** albo bezpieczeństwa wątkowego, albo ochrony blokadą.

> Wymagania dotyczące publikacji obiektu zależą od jego zmienności.
>
> - Obiekty niezmienne mogą być publikowane w dowolny sposób.
> - Obiekty niezmienne technicznie muszą być publikowane bezpiecznie.
> - Obiekty zmienne muszą być publikowane bezpiecznie i być bezpieczne wątkowo lub chronione blokadą.

3.5.6. Bezpieczne współdzielenie obiektów

Gdy uzyskujemy referencję do obiektu, powinniśmy dokładnie wiedzieć, co możemy z nim zrobić. Czy musimy uzyskać blokadę przed jego użyciem? Czy możemy zmienić jego stan czy tylko go odczytać? Wiele błędów współbieżności pojawia się, bo

[17] Prawdopodobnie jest to błąd projektowy biblioteki klas dotyczących dat — *przyp. aut.*

programista źle zrozumiał „reguły gry” współdzielonymi obiektami. Publikując obiekt, zawsze zaznaczaj, w jaki sposób powinien być obsługiwany.

Oto kilka najważniejszych strategii korzystania ze współdzielonych obiektów w aplikacjach współbieżnych.

Odosobnienie w wątku. Obiekt odosobniony w wątku należy wyłącznie do jednego wątku i jest przez niego modyfikowany.

Współdzielenie tylko z prawem do odczytu. Obiekt współdzielony tylko do odczytu może być współbieżnie odczytywany przez wiele wątków bez dodatkowej synchronizacji, ale żaden wątek nie powinien go modyfikować. Obiektami tego typu są obiekty niezmienne i niezmienne technicznie.

Współdzielenie z bezpieczeństwem wątkowym. Obiekt bezpieczny wątkowo wewnętrznie synchronizuje dostęp, więc wiele wątków może bezpiecznie korzystać z jego publicznego interfejsu bez potrzeby dodatkowej synchronizacji.

Ochrona. Obiekt chroniony dostępny jest dopiero po uzyskaniu odpowiedniego klucza. Obiekty chronione to takie, które znajdują się w obiektach bezpiecznych wątkowo lub zostały opublikowane i są chronione przez konkretną blokadę.