Cay S. Horstmann

Java 9

Przewodnik doświadczonego programisty

Wydanie II

Helion

Tytuł oryginału: Core Java SE 9 for the Impatient (2nd Edition)

Tłumaczenie: Andrzej Stefański

ISBN: 978-83-283-4250-7

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jav9p2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jav9p2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

10

Programowanie współbieżne

W tym rozdziale

- 10.1. Zadania współbieżne
- 10.2. Obliczenia asynchroniczne
- 10.3. Bezpieczeństwo wątków
- 10.4. Algorytmy równoległe
- 10.5. Struktury danych bezpieczne dla wątków
- 10.6. Atomowe liczniki i akumulatory
- 10.7. Blokady i warunki
- 10.8. Wątki
- 10.9. Procesy
- Ćwiczenia

Język Java był pierwszym z popularnych języków programowania z wbudowanym wsparciem programowania współbieżnego. Pierwsi programiści korzystający z Javy entuzjastycznie przyjęli prostotę, z jaką można było wczytywać pliki w drugoplanowych wątkach lub implementować serwer internetowy obsługujący równolegle wiele żądań. W tamtym czasie skupiano się na obciążaniu procesora pracą w czasie, gdy niektóre zadania czekały na dane z sieci. Obecnie większość komputerów posiada wiele procesorów lub rdzeni i programistom zależy na tym, by wykorzystać wszystkie procesory.

W tym rozdziale nauczysz się, jak dzielić obliczenia na współbieżne zadania i w jaki sposób bezpiecznie je wykonywać. Skupię się na potrzebach programistów aplikacji, a nie systemowych piszących serwery internetowe lub oprogramowanie serwerowe.

Z tego powodu informacje w tym rozdziale rozplanowałem w taki sposób, by tam, gdzie to możliwe, najpierw pokazać narzędzia, których powinieneś używać w swojej pracy. Konstrukcje niskopoziomowe opiszę w dalszej części tego rozdziału. Znajomość tych niskopoziomowych szczegółów przydaje się do tego, by mieć wyczucie, z jakim kosztem wiąże się wykonywanie

poszczególnych operacji. Najlepiej jednak pozostawić niskopoziomowe programowanie wątków ekspertom. Jeśli zechcesz stać się jednym z nich, polecam wspaniałą książkę *Java Concurrency in Practice* (Brian Goetz i inni, Addison-Wesley, 2006).

Najważniejsze punkty tego rozdziału:

1. `Runnable` opisuje zadanie, które może być wykonane asynchronicznie, ale nie zwraca wyniku.
2. `ExecutorService` planuje wykonanie instancji zadań.
3. `Callable` opisuje zadanie, które może być wykonane asynchronicznie i zwraca wynik.
4. Możesz wysłać jedną lub więcej instancji `Callable` do `ExecutorService` i połączyć wyniki, gdy będą dostępne.
5. Gdy wiele wątków pracuje na wspólnych danych bez synchronizacji, wyniki są nieprzewidywalne.
6. Lepiej używać algorytmów równoległych i bezpiecznych struktur danych, niż korzystać z blokad.
7. Równoległe operacje na strumieniach i tablicach automatycznie i bezpiecznie zrównoleglają wykonywanie operacji.
8. `ConcurrentHashMap` to bezpieczna dla wątków tablica skrótów pozwalająca aktualizować elementy za pomocą operacji atomowych.
9. Możesz użyć klasy `AtomicLong` jako współdzielonego licznika bez konieczności tworzenia blokad lub wykorzystać `LongAdder` w przypadku dużej rywalizacji.
10. Blokada zapewnia, że tylko jeden wątek w danej chwili wykonuje krytyczny fragment kodu.
11. Zadanie, które można przerwać, powinno kończyć działanie, gdy ustawiana jest flaga `interrupted` lub pojawia się wyjątek `InterruptedException`.
12. Długie zadanie nie powinno blokować interfejsu użytkownika w programie, ale postęp i końcowa aktualizacja muszą być wykonywane w wątku obsługującym interfejs użytkownika.
13. Klasa `Process` pozwala wykonywać polecenia w oddzielnych procesach oraz wchodzić w interakcję ze strumieniem wejściowym, wyjściowym i błędów.

10.1. Zadania współbieżne

Projektując program współbieżny, musisz myśleć na temat zadań, które mogą być uruchamiane jednocześnie. W kolejnych punktach zobaczysz, jak wykonywać zadania równolegle.

10.1.1. Uruchamianie zadań

W języku Java interfejs `Runnable` opisuje zadanie, które chcesz uruchomić, być może równolegle z innymi.

```
public interface Runnable {
    void run();
}
```

Tak jak w przypadku wszystkich metod, kod metody `run` jest wykonywany w **wątku**. Wątek jest mechanizmem pozwalającym na wykonanie ciągu instrukcji, zazwyczaj dostarczanym przez system operacyjny. Wiele wątków działa równolegle, korzystając z różnych procesorów lub różnych odcinków czasu na tym samym procesorze.

Jeśli chcesz wykonać `Runnable` w oddzielnym wątku, mógłbyś tworzyć wątek dla każdego `Runnable` — zobaczysz, jak to zrobić w punkcie 10.8.1, „Uruchamianie wątku". W praktyce jednak zazwyczaj nie ma sensu utrzymywanie relacji „jeden do jednego" pomiędzy zadaniami i wątkami. Gdy zadania są krótkie, lepiej uruchomić wiele zadań w tym samym wątku, by nie marnować czasu na tworzenie wątku. Gdy Twoje zadania wymagają wielu obliczeń, aby zminimalizować narzut powstający przy przełączaniu między wątkami, lepiej jest utworzyć po jednym wątku na procesor, zamiast tworzyć wątek dla każdego zadania. Nie warto zawracać sobie tym głowy przy tworzeniu zadań, dlatego najlepiej oddzielić tworzenie zadań i planowanie ich wykonania.

W bibliotece Java wspierającej współbieżność `ExecutorService` planuje wykonanie i wykonuje zadanie, wybierając wątki, w których ma być ono uruchomione.

```
Runnable task = () -> { ... };
ExecutorService executor = ...;
executor.execute(task);
```

Klasa `Executors` ma metody wytwórcze dla różnych usług wykonawców z różnymi regułami planowania. Wywołanie

```
exec = Executors.newCachedThreadPool();
```

zwraca usługę z wykonawcą zoptymalizowanym dla programów z wieloma krótkimi zadaniami lub spędzającymi większość czasu na oczekiwaniu. Każde zadanie jest wykonywane w bezczynnym wątku, jeśli to możliwe, ale gdy wszystkie wątki są zajęte, tworzony jest nowy wątek. Liczba równoległych wątków nie jest ograniczona. Wątki bezczynne przez dłuższy czas są likwidowane.

Wywołanie

```
exec = Executors.newFixedThreadPool(liczbawątków);
```

zwraca pulę z określoną liczbą wątków. Gdy wysyłasz zadanie, czeka ono w kolejce do czasu, gdy pojawi się dostępny wątek. Jest to dobre rozwiązanie w przypadku zadań wymagających wielu obliczeń lub dla ograniczenia zasobów wykorzystywanych przez usługę. Możesz ustalić liczbę wątków na podstawie liczby dostępnych procesorów, którą otrzymujesz, pisząc

```
int processors = Runtime.getRuntime().availableProcessors();
```

Teraz pora uruchomić program demonstrujący współbieżność z kodów dołączonych do książki. Wykonuje on równolegle dwa zadania.

```
public static void main(String[] args) {
    Runnable powitania = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Witaj " + i);
    };
    Runnable pożegnania = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Żegnaj " + i);
    };
    ExecutorService executor = Executors.newCachedThreadPool();
    executor.execute(powitania);
    executor.execute(pożegnania);
}
```

Uruchom program kilka razy, by zobaczyć, w jaki sposób przeplatają się wyświetlane liczby.

```
Żegnaj 1
...
Żegnaj 871
Żegnaj 872
Witaj 806
Żegnaj 873
Żegnaj 874
Żegnaj 875
Żegnaj 876
Żegnaj 877
Żegnaj 878
Żegnaj 879
Żegnaj 880
Żegnaj 881
Witaj 807
Żegnaj 882
...
Witaj 1000
```

Możesz zauważyć, że program czeka chwilę po wyświetleniu ostatniej linii. Kończy działanie dopiero, gdy wątki z puli są przez jakiś czas bezczynne i `Executor` kończy ich działanie.

Jeśli równolegle wykonywane zadania próbują odczytywać lub aktualizować współdzieloną wartość, wynik może być nieprzewidywalny. Szczegółowo omówimy ten problem w podrozdziale 10.3., „Bezpieczeństwo wątków”. Na razie przyjmijmy, że zadania nie będą współdzieliły modyfikowalnych danych.

10.1.2. Obiekty Future

Interfejs `Runnable` wykonuje zadanie, ale nie zwraca wartości. Jeśli masz zadanie obliczające wynik, zamiast `Runnable` użyj interfejsu `Callable<V>`. Jego metoda `call`, w przeciwieństwie do metody `run` interfejsu `Runnable`, zwraca wartość:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Dodatkowo metoda `call` może wyrzucić dowolne wyjątki, które mogą zostać przekazane do kodu otrzymującego wyniki.

Aby wykonać `Callable`, prześlij go do usługi wykonawcy:

```
ExecutorService executor = Executors.newFixedThreadPool();
Callable<V> zadanie = ...;
Future<V> wynik = executor.submit(zadanie);
```

Wysyłając zadanie, otrzymujesz obiekt `Future` reprezentujący obliczenia, których wynik będzie dostępny w przyszłości. Interfejs `Future` ma następujące metody:

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;
boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
boolean isDone()
```

Metoda `get` jest zablokowana, dopóki nie pojawi się wynik lub nie zostanie przekroczony czas wykonania. Oznacza to, że wątek zawierający jej wywołanie nie posuwa się do przodu, dopóki metoda nie zakończy się lub nie wyrzuci wyjątku. Jeśli metoda `call` wyrzuciła wyjątek, metoda `get` wyrzuca wyjątek `ExecutionException` opakowujący wyrzucony wyjątek. Jeśli zostanie przekroczony czas wykonania, metoda `get` wyrzuca wyjątek `TimeoutException`.

Metoda `cancel` próbuje anulować zadanie. Jeśli zadanie jeszcze nie zostało uruchomione, nie zostanie wykonane. W przeciwnym wypadku, jeśli `mayInterruptIfRunning` ma wartość `true`, wątek wykonujący zadanie jest przerywany.

Jeśli chcesz, by można było przerwać zadanie, musi ono jakiś czas sprawdzać żądania przerwania. Jest to wymagane w przypadku wszystkich zadań, które będziesz chciał anulować po uzyskaniu wyniku w innym podzadaniu. Więcej informacji na temat przerywania zadań znajdziesz w punkcie 10.8.2., „Przerywanie wątków”.

Może się zdarzyć, że zadanie będzie musiało poczekać na wynik wielu podzadań. Zamiast wysyłać każde podzadanie oddzielnie, możesz użyć metody `invokeAll`, do której przekazujesz kolekcję instancji `Callable`.

Załóżmy, że chcesz policzyć, jak często słowo pojawia się w zbiorze plików. Dla każdego pliku utwórz `Callable<Integer>`, który zwraca licznik dla tego pliku. Następnie prześlij je wszystkie do wykonawcy. Gdy wszystkie zadania zostaną wykonane, otrzymasz listę obiektów implementujących interfejs `Future` (z których wszystkie są już wykonane) i możesz zsumować odpowiedzi.

```
String słowo = ...;
Set<Path> ścieżki = ...;
List<Callable<Long>> zadania = new ArrayList<>();
for (Path p : ścieżki) zadania.add(
    () -> { return liczba wskazanych słów w p });
```

```
List<Future<Long>> wyniki = executor.invokeAll(zadania);
// To wywołanie pozostanie zablokowane do czasu zakończenia wszystkich zadań
long suma = 0;
for (Future<Long> wynik : wyniki) suma += wynik.get();
```

Istnieje też odmiana `invokeAll` z ograniczeniem czasu, anulującym wszystkie zadania, które nie zakończyły się w określonym czasie.

Jeśli martwi Cię to, że wywołujące zadanie blokuje działanie, dopóki nie zostaną wykonane wszystkie podzadania, możesz wykorzystać `ExecutorCompletionService`. Zwraca ona wartości `Future` w takiej kolejności, w jakiej kończą działanie.

```
ExecutorCompletionService usługa
    = new ExecutorCompletionService(executor);
for (Callable<T> zadanie : zadania) service.submit(zadanie);
for (int i = 0; i < zadania.size(); i++) {
    Wykonaj usługa.take().get()
    Inne operacje
}
```

Metoda `invokeAny` działa jak `invokeAll`, ale zwraca wartość, gdy tylko dowolne z przekazanych zadań zakończy działanie bez wyrzucania wyjątku. Wtedy zwraca ona wartość tego obiektu `Future`. Pozostałe zadania są anulowane. Jest to przydatne w przypadku wyszukiwania, które można zakończyć po znalezieniu pierwszego dopasowania. Poniższy fragment kodu ustala położenie pliku zawierającego wskazane słowo:

```
String słowo = ...;
Set<Path> pliki = ...;
List<Callable<Path>> zadania = new ArrayList<>();
for (Path p : pliki) zadania.add(
    () -> { if (słowo występuje w p) return p; else throw ... });
Path znalezione = executor.invokeAny(zadania);
```

Jak możesz tutaj zobaczyć, klasa `ExecutorService` wykonuje dla Ciebie sporo pracy. Nie tylko przyporządkowuje zadania do wątków, ale też obsługuje wyniki działania zadań, wyjątki i anulowanie.

Java EE dostarcza podklasę `ManagedExecutorService`, która może być użyta w zadaniach równoległych w środowisku Java EE.

10.2. Obliczenia asynchroniczne

W poprzednim podrozdziale nasze podejście do obliczeń równoległych ograniczało się do podzielenia zadania i oczekiwania na zakończenie obróbki wszystkich jego części. Oczekiwanie nie zawsze jest dobrym pomysłem. W kolejnych punktach zobaczysz, w jaki sposób implementować obliczenia niewymagające oczekiwania, czyli obliczenia **asynchroniczne**.

10.2.1. Klasa CompletableFuture

Jeśli masz obiekt typu `Future`, by uzyskać jego wartość, musisz wywołać metodę `get`, co zablokuje wykonywanie kodu do czasu otrzymania wyniku jej działania. Klasa `CompletableFuture` implementuje interfejs `Future` i dostarcza drugi mechanizm pozwalający uzyskać wynik. Rejestrujesz tutaj funkcję do wywołania zwrotnego, która będzie wywołana (w jakimś wątku) z wynikiem, gdy tylko on się pojawi.

```
CompletableFuture<String> f = ...;
   f.thenAccept((String s) -> Przetwarzanie wyniku s);
```

W ten sposób możesz bez blokowania przetwarzać wynik, gdy tylko będzie on dostępny.

Istnieje kilka metod API, które zwracają obiekty `CompletableFuture`. Na przykład klasa `HttpClient` może pobrać asynchronicznie stronę internetową:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(new URI(urlString)).GET().build();
CompletableFuture<HttpResponse<String>> f = client.sendAsync(request, BodyHandler.
➥asString());
```

Aby uruchomić zadanie asynchronicznie i otrzymać `CompletableFuture`, nie przesyłasz go bezpośrednio do usługi wykonawcy. Zamiast tego wywołujesz metodę statyczną `Completable➥Future.supplyAsync`:

```
CompletableFuture<String> f = CompletableFuture.supplyAsync(
   () -> { String result; Przetwarzanie wyniku ; return result; }, executor);
```

Jeśli pominiesz wykonawcę, zadanie jest wykonywane przez wykonawcę domyślnego (dokładniej wykonawcę zwracanego przez `ForkJoinPool.commonPool()`).

Zauważ, że pierwszy argument tej metody to `Supplier<T>`, a nie `Callable<T>`. Oba interfejsy opisują funkcje bez argumentów i zwracają wartość typu `T`, ale funkcja `Supplier` nie może wyrzucić wyjątku kontrolowanego.

Obiekt typu `CompletableFuture` może ukończyć działanie na dwa sposoby: z wynikiem albo wyrzucając nieprzechwycony wyjątek. Aby obsłużyć oba przypadki, użyj metody `whenComplete`. Przekazana funkcja jest wywoływana z wynikiem (lub z wartością `null`, jeśli wyniku nie ma) oraz wyjątkiem (lub wartością `null`, jeśli wyjątku nie ma).

```
f.whenComplete((s, t) -> {
    if (t == null) { Przetwarzanie wyniku s; }
    else { Przetwarzanie Throwable t; }
});
```

Klasa `CompletableFuture` jest tak nazywana, ponieważ można ręcznie ustawić w niej wartość zakończenia. (W innych bibliotekach obsługujących współbieżność tego typu obiekt nazywany jest obietnicą [ang. *promise*]). Oczywiście gdy tworzysz `CompletableFuture` z `supplyAsync`, wartość zakończenia jest wewnętrznie ustawiana po zakończeniu zadania. Jednak jawne ustawianie wyniku daje dodatkową elastyczność. Na przykład dwa zadania mogą równocześnie pracować nad ustalaniem odpowiedzi:

```
CompletableFuture<Integer> f = new CompletableFuture<>();
executor.execute(() -> {
   int n = workHard(arg);
   f.complete(n);
});
executor.execute(() -> {
   int n = workSmart(arg);
   f.complete(n);
});
```

Aby zamiast tego zakończyć działanie obiektu wyjątkiem, wywołaj

```
Throwable t = ...;
f.completeExceptionally(t);
```

Bezpiecznie jest wywołać `complete` lub `completeExceptionally` na tym samym obiekcie typu `Future` w wielu wątkach. Jeśli działanie obiektu zostało wcześniej zakończone, te wywołania nie spowodują żadnego działania.

Metoda `isDone` mówi o tym, czy obiekt typu `Future` zakończył się (normalnie lub wyrzucając wyjątek). W poprzednim przykładzie metody `workHard` i `workSmart` mogą wykorzystać tę informację, aby zatrzymać pracę, gdy wynik zostanie ustalony przez inną metodę.

W odróżnieniu od zwykłego obiektu typu `Future` przetwarzanie obiektu `Completable↪Future` nie jest przerywane, gdy wywołasz jego metodę `cancel`. Anulowanie po prostu informuje obiekt typu `Future`, że ma zakończyć działanie, wyrzucając wyjątek `CancellationException`. Ogólnie ma to sens, ponieważ `CompletableFuture` może nie mieć jednego wątku odpowiedzialnego za jego zakończenie. To ograniczenie jednak dotyczy również instancji `CompletableFuture` zwracanych przez metody takie jak `supplyAsync`, które zasadniczo mogą być przerywane. Obejście tego znajdziesz w ćwiczeniu 27.

10.2.2. Tworzenie obiektów typu CompletableFuture

Nieblokujące wywołania są zaimplementowane za pomocą wywołań zwrotnych. Programista rejestruje do wywołania zwrotnego funkcję, która ma zostać wykonana po zakończeniu zadania. Oczywiście jeśli kolejne działanie jest również asynchroniczne, następne działanie po nim jest obsługiwane przez inne wywołanie zwrotne. Nawet jeśli programista ma na myśli algorytm „najpierw wykonaj krok 1., następnie krok 2., potem krok 3.”, logika programu jest rozproszona w różnych wywołaniach zwrotnych. Sytuacja pogarsza się, gdy trzeba dodać obsługę błędów. Załóżmy, że krokiem 2. jest „logowanie użytkownika”. Może pojawić się konieczność powtórzenia tego kroku, ponieważ użytkownik może pomylić się przy wpisywaniu danych. Próba zaimplementowania takiego przepływu sterowania w szeregu wywołań zwrotnych lub zrozumienie tego po zaimplementowaniu mogą stanowić wyzwanie.

Klasa `CompletableFuture` rozwiązuje ten problem, dostarczając mechanizm pozwalający na łączenie asynchronicznych zadań w ciąg przetwarzania.

Dla przykładu załóżmy, że chcemy wyodrębnić wszystkie odnośniki ze strony internetowej, by stworzyć robota internetowego. Przypuśćmy, że mamy metodę

```
public void CompletableFuture<String> wczytajStronę(URI url)
```

która zwraca tekst ze strony internetowej po jego otrzymaniu. Jeśli metoda

```
public static List<URI> pobierzOdnośniki(String strona)
```

zwraca odnośniki ze strony HTML, możesz zaplanować jej wywołanie, gdy pojawi się strona:

```
CompletableFuture<String> treść = wczytajStronę(url);
CompletableFuture<List<URI>> odnośniki = treść.thenApply(Parser::pobierzOdnośniki);
```

Metoda `thenApply` również nie blokuje — zwraca inną wartość `Future`. Gdy zakończy się ustalanie pierwszej wartości `Future`, jej wynik jest dostarczany do metody `getLinks`, a wartość zwrócona przez tę metodę staje się ostatecznym wynikiem.

W przypadku `CompletableFuture` określasz po prostu, co ma zostać wykonane i w jakiej kolejności. Nie stanie się to oczywiście natychmiast, ale ważne jest to, że cały kod znajduje się w jednym miejscu.

Koncepcyjnie `CompletableFuture` to proste API, ale istnieje wiele metod do tworzenia instancji tej klasy. Spójrzmy najpierw na te, które operują na pojedynczej instancji (patrz tabela 10.1). (Dla każdej z opisanych metod istnieją dwa odpowiedniki `Async`, o których nie piszę. Jeden z nich korzysta ze współdzielonej `ForkJoinPool`, a drugi ma parametr `Executor`). W tabeli korzystam ze skróconego zapisu niezgrabnych interfejsów funkcjonalnych, pisząc `T -> U` zamiast `Function<? superT, U>`. Nie są to oczywiście rzeczywiste typy języka Java.

Widziałeś już metodę `thenApply`. Przyjmijmy, że `f` jest funkcją, która pobiera wartości typu `T` i zwraca wartości typu `U`. Wywołania

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

zwracają obiekt `Future`, który wykonuje funkcję `f` na wyniku z tego obiektu, gdy stanie się ona dostępna. Drugie wywołanie uruchamia `f` w jeszcze innym wątku.

Zamiast pobierać funkcję `T -> U`, metoda `thenCompose` pobiera funkcję `T -> CompletableFuture<U>`. Brzmi to raczej abstrakcyjnie, ale może być całkiem naturalne. Rozważ odczytywanie strony internetowej z podanego adresu URL. Zamiast dostarczać metodę

```
public String pobierzStronęBlokując(URL url)
```

bardziej eleganckim rozwiązaniem jest sprawić, by metoda ta zwracała obiekt `Future`:

```
public CompletableFuture<String> pobierzStronę(URL url)
```

Załóżmy teraz, że mamy inną metodę pobierającą URL z danych wprowadzanych przez użytkownika, na przykład z okna dialogowego, które nie ujawnia danych, zanim użytkownik nie kliknie przycisku *OK*. To również jest przyszłe zdarzenie:

```
public CompletableFuture<URL> pobierzURL(String pytanie)
```

Mamy tutaj dwie funkcje: `T -> CompletableFuture<U>` oraz `U -> CompletableFuture<V>`. Oczywiście tworzą one razem funkcję `T -> CompletableFuture<V>`, jeśli druga funkcja jest wywoływana po zakończeniu pierwszej. Właśnie tak działa `thenCompose`.

W poprzednim punkcie widziałeś użycie metody `whenComplete` do obsługi wyjątków. Istnieje też metoda `handle`, która wymaga podania funkcji przetwarzającej wynik lub wyjątek i obliczającej nowy wynik. W wielu przypadkach prościej jest wywołać zamiast tego metodę `exceptionally`:

```
CompletableFuture<String> contents = readPage(url).exceptionally(t -> { Log t; return
emptyPage; });
```

Funkcja przekazana jako handler jest wywoływana tylko, jeśli wystąpi wyjątek — tworzy ona wynik, który można wykorzystać do dalszego przetwarzania w potoku. Jeśli nie pojawi się wyjątek, zostanie wykorzystany pierwotny wynik.

Metody z tabeli 10.1 z wynikiem typu `void` są zazwyczaj wykorzystywane na końcu ciągu przetwarzania.

Tabela 10.1. *Dodawanie działania do obiektu CompletableFuture<T>*

Metoda	Parametr	Opis
`thenApply`	`T -> U`	Wykonaj funkcję na wyniku
`thenAccept`	`T -> void`	Jak `thenApply`, ale zwraca `void`
`thenCompose`	`T -> CompletableFuture<U>`	Wywołaj funkcję na wyniku i wykonaj zwróconą wartość `Future`
`handle`	`(T, Throwable) -> U`	Przetwórz wynik lub błąd
`whenComplete`	`(T, Throwable) -> void`	Jak `handle`, ale zwraca `void`
`exceptionally`	`Throwable -> T`	Zamienia błąd na domyślny wynik
`thenRun`	`Runnable`	Wykonuje obiekt `Runnable` zwracający `void`

Przejdźmy teraz do metod, które łączą wiele obiektów `Future` (patrz tabela 10.2).

Tabela 10.2. *Łączenie wielu obiektów łączących*

Metoda	Parametry	Opis
`thenCombine`	`CompletableFuture<U>, (T, U) -> V`	Wykonuje oba i łączy wyniki za pomocą przekazanej funkcji
`thenAcceptBoth`	`CompletableFuture<U>, (T, U) -> void`	Jak `thenCombine`, ale zwraca `void`
`runAfterBoth`	`CompletableFuture<?>, Runnable`	Wykonuje `Runnable` po zakończeniu obu
`applyToEither`	`CompletableFuture<T>, T -> V`	Gdy dostępny jest wynik z jednego lub drugiego, przekaż go do wskazanej funkcji
`acceptEither`	`CompletableFuture<T>, T -> void`	Jak `applyToEither`, ale zwraca `void`
`runAfterEither`	`CompletableFuture<?>, Runnable`	Wykonaj `Runnable` po zakończeniu jednego lub drugiego
`static allOf`	`CompletableFuture<?>...`	Zakończ działanie z wynikiem `void` po zakończeniu wszystkich przekazanych `Futures`
`static anyOf`	`CompletableFuture<?>...`	Zakończ działanie z wynikiem `void` po zakończeniu jednego z przekazanych `Futures`

Pierwsze trzy metody uruchamiają działania `CompletableFuture<T>` oraz `CompletableFuture<U>` równocześnie i łączą wyniki.

Kolejne trzy metody uruchamiają równocześnie dwa działania `CompletableFuture`. Gdy tylko jedno z nich zakończy działanie, jego wynik jest przekazywany i drugi wynik jest ignorowany.

I wreszcie statyczne metody `allOf` i `anyOf` przyjmują dowolną liczbę obiektów `CompletableFuture` i zwracają `CompletableFuture<Void>`, który jest wykonany, gdy wszystkie, lub dowolna z nich, zakończą działanie. Metoda `allOf` nie zwraca wyniku. Metoda `anyOf` *nie* kończy pozostałych zadań. Ćwiczenia 28. i 29. pokazują użyteczne usprawnienia tych dwóch metod.

Jeśli spojrzeć od strony technicznej, metody w tym punkcie przyjmują parametry typu `CompletionStage`, a nie `CompletableFuture`. Interfejs CompletionStage opisuje, w jaki sposób przygotować obliczenia asynchroniczne, podczas gdy interfejs `Future` skupia się na wyniku obliczeń. Interfejs `CompletableFuture` obejmuje zarówno `CompletionStage`, jak i `Future`.

10.2.3. Długie zadania obsługujące interfejs użytkownika

Jednym z powodów wykorzystywania wątków jest konieczność dbania o responsywność programów. Jest to szczególnie istotne w przypadku aplikacji z interfejsem użytkownika. Gdy Twój program musi zrobić coś wymagającego czasu, nie możesz tego wykonać w wątku obsługującym interfejs użytkownika, ponieważ w tym czasie interfejs użytkownika byłby zamrożony. Zamiast tego uruchamia się inny wątek do tego zadania.

Na przykład: jeśli chcesz wczytać stronę internetową, na której użytkownik kliknął przycisk, nie rób tego w ten sposób:

```
Button czytaj = new Button("Czytaj");
czytaj.setOnAction(event -> { // Źle — zajmująca dużo czasu akcja jest wykonywana w wątku UI
    Scanner in = new Scanner(url.openStream())
    while (in.hasNextLine()) {
        String line = in.nextLine();
        ...
    }
});
```

Zamiast tego wykonaj zadanie w oddzielnym wątku.

```
read.setOnAction(event -> { // Dobrze — długotrwałe zadanie wykonywane w oddzielnym wątku
    Runnable task = () -> {
        Scanner in = new Scanner(url.openStream());
        while (in.hasNextLine()) {
            String line = in.nextLine();
            ...
        }
    }
    executor.execute(task);
});
```

Nie możesz jednak bezpośrednio aktualizować interfejsu użytkownika z wątku wykonującego długie zadanie. Interfejsy użytkownika takie jak JavaFX, Swing czy Android nie są bezpieczne dla wątków. Nie możesz operować na elementach interfejsu użytkownika z wielu wątków, ponieważ możesz je uszkodzić. W rzeczywistości JavaFX oraz Android sprawdzają to i wyrzucają wyjątek, jeśli próbujesz uzyskać dostęp do interfejsu użytkownika z wątku innego niż obsługujący interfejs użytkownika.

Dlatego musisz skierować wszystkie czynności związane z aktualizacją interfejsu użytkownika do wykonania w obsługującym go wątku. Każda biblioteka interfejsu użytkownika dostarcza mechanizm umożliwiający przekazywanie `Runnable` do wykonania w wątku interfejsu użytkownika. Na przykład w JavaFX możesz użyć

```
Platform.runLater(() ->
    message.appendText(line + "\n"));
```

Nudne jest implementowanie długotrwałych operacji w taki sposób, by przekazywały użytkownikowi informacje zwrotne dotyczące postępu, dlatego biblioteki interfejsu użytkownika zazwyczaj udostępniają pewnego rodzaju klasę pomocniczą zarządzającą szczegółami, taką jak `SwingWorker` w bibliotece Swing czy `AsyncTask` w Androidzie. Określasz działania w długotrwałym zadaniu (które jest wykonywane w oddzielnym wątku), a także aktualizacje informacji o postępie oraz zakończeniu (które są wykonywane w wątku obsługującym interfejs użytkownika).

Klasa `Task` w JavaFX przyjmuje odrobinę inne podejście do aktualizowania informacji o postępie. Udostępnia metody pozwalające na aktualizację właściwości zadania (komunikat, stopień zaawansowania oraz końcową wartość) w długotrwałym wątku. Wiążesz właściwości z elementami interfejsu użytkownika, a te są następnie aktualizowane w wątku interfejsu użytkownika.

10.3. Bezpieczeństwo wątków

Wielu programistów początkowo myśli, że współbieżne programowanie jest dość proste. Po prostu dzielisz swoją pracę na zadania i to wszystko. Co może się stać?

W kolejnych punktach pokażę Ci, co może nie wyjść, i dokonam przeglądu wysokopoziomowych sposobów radzenia sobie z tym.

10.3.1. Widoczność

Nawet operacje tak proste jak zapis i odczyt zmiennej mogą być niewiarygodnie skomplikowane w nowoczesnych procesorach. Popatrzmy na przykład:

```
private static boolean done = false;

public static void main(String[] args) {
    Runnable powitania = () -> {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Witaj " + i);
        done = true;
```

```
        };
        Runnable pożegnanie = () -> {
            int i = 1;
            while (!done) i++;
            System.out.println("Żegnaj " + i);
        };
        Executor executor = Executors.newCachedThreadPool();
        executor.execute(powitania);
        executor.execute(pożegnanie);
    }
```

Pierwsze zadanie wyświetla `"Witaj"` tysiąc razy, a następnie ustawia wartość `done` na `true`. Drugie zadanie czeka do chwili, gdy zmienna `done` przyjmie wartość `true`, i wtedy wyświetla raz `"Żegnaj"`, zwiększając licznik w oczekiwaniu na to szczęśliwe zakończenie.

Możesz oczekiwać, że wynikiem działania będzie coś w stylu

```
Witaj 1
...
Witaj 1000
Żegnaj 501249
```

Gdy uruchomiłem ten program na swoim laptopie, program wyświetlił wiersze do `"Witaj 1000"` i nie zakończył działania. Efekt działania

```
done = true;
```

może nie być *widoczny* dla wątku, w którym uruchomione jest drugie zadanie.

Dlaczego miałoby to nie być widoczne? Współczesne kompilatory, maszyny wirtualne i procesory wykonują wiele optymalizacji. Te optymalizacje zakładają, że kod jest sekwencyjny, jeśli jawnie nie zostanie zadeklarowane coś innego. Jednym ze sposobów optymalizacji jest przechowywanie fragmentów pamięci w pamięci podręcznej.

Myślimy o miejscach w pamięci takich jak zmienna `done` niczym o bitach gdzieś w tranzystorach układu RAM. Ale pamięci RAM są powolne — kilka razy wolniejsze niż nowoczesne procesory. Dlatego procesor próbuje przechowywać potrzebne dane w rejestrach lub pamięci podręcznej na płycie głównej i w ostateczności przepisuje zmiany do pamięci. Ta pamięć podręczna jest wręcz niezastąpiona, jeśli chodzi o wydajność procesora. Istnieją operacje do synchronizowania przechowywanych w pamięci podręcznej kopii, ale znacząco wpływają one na wydajność i są wykonywane tylko na żądanie.

Innym sposobem optymalizacji jest zamiana kolejności instrukcji. Kompilator, maszyna wirtualna i procesor mogą zmieniać kolejność instrukcji, aby przyspieszyć wykonywanie operacji, pod warunkiem że nie zmieni to semantyki sekwencyjnego programu.

Na przykład przeanalizuj obliczenia:

```
x = Coś, co nie korzysta z y;
y = Coś, co nie korzysta z x;
z = x + y;
```

Pierwsze dwa kroki muszą być wykonane przed trzecim, ale mogą wystąpić w dowolnej kolejności. Procesor może (i często będzie) wykonywał dwa pierwsze kroki równocześnie lub zamieni ich kolejność, jeśli dane wejściowe do drugiego kroku będą dostępne wcześniej.

W naszym przypadku pętla

```
while (!done) i++;
```

może być przekształcona do postaci

```
if (!done) while (true) i++;
```

ponieważ ciało pętli nie zmienia wartości `done`.

Domyślnie przy optymalizacji przyjmowane jest założenie, że nie ma równoległych prób dostępu do pamięci. Jeśli takie wywołania istnieją, maszyna wirtualna musi o tym wiedzieć, aby mogła utworzyć instrukcje procesora zapobiegające niewłaściwej zmianie kolejności.

Istnieje kilka sposobów zapewniania widoczności aktualizacji zmiennej. Oto lista:

1. Wartość zmiennej `final` jest widoczna po inicjalizacji.
2. Początkowa wartość zmiennej statycznej jest widoczna po inicjalizacji statycznej.
3. Zmiany zmiennej z modyfikatorem `volatile` są widoczne.
4. Zmiany wprowadzane przed zwolnieniem blokady są widoczne dla każdego pobierającego tę samą blokadę (patrz punkt 10.7.1, „Blokady").

W naszym przypadku problem znika, jeśli zadeklarujesz współdzieloną zmienną `done` z modyfikatorem `volatile`:

```
private static volatile boolean done;
```

Dzięki temu kompilator generuje instrukcje, które sprawiają, że maszyna wirtualna wydaje procesorowi polecenia synchronizacji pamięci podręcznej. W wyniku tego wszystkie zmiany zmiennej `done` w jednym zadaniu stają się widoczne w innym zadaniach.

Modyfikator `volatile` wystarczy, by rozwiązać ten konkretny problem. Jak zobaczysz jednak w kolejnym punkcie, deklarowanie współdzielonych zmiennych jako `volatile` nie jest ogólnym rozwiązaniem.

Wspaniałym pomysłem jest deklarowanie każdego pola, które nie zmienia się po inicjalizacji jako `final`. Dzięki temu nie będzie problemów z jej widocznością.

10.3.2. Wyścigi

Załóżmy, że mamy wiele współbieżnych zadań, które aktualizują współdzielony licznik przechowujący zmienną całkowitą.

```
private static volatile int licznik = 0;
...
licznik++; // Zadanie 1.
...
licznik++; // Zadanie 2.
...
```

Zmienna została zadeklarowana z modyfikatorem `volatile`, dzięki czemu aktualizacje są widoczne. To jednak nie wystarcza.

Aktualizacja `licznik++` w rzeczywistości oznacza

```
rejestr = licznik + 1;
licznik = rejestr;
```

Jeśli między te operacje zostaną wstawione inne operacje, w zmiennej `count` może zostać zapisana niewłaściwa wartość. Korzystając ze słownictwa stosowanego przy obliczeniach równoległych, mówimy, że operacja inkrementacji nie jest *atomowa*. Rozważ taki scenariusz:

```
int licznik = 0;          // Początkowa wartość
rejestr1 = licznik + 1;  // Wątek 1. oblicza licznik + 1
... // Wątek 1. jest wywłaszczany
rejestr2 = licznik + 1;  // Wątek 2. oblicza licznik + 1
licznik = rejestr2;       // Wątek 2. zapisuje 1 w licznik
... // Wątek 1. zaczyna ponownie działać
licznik = rejestr1;       // Wątek 1. zapisuje 1 w licznik
```

W tej sytuacji `licznik` ma wartość `1`, nie `2`. Tego rodzaju błąd jest nazywany **wyścigiem** (ang. *race condition*), ponieważ zależy on od tego, który wątek wygra „wyścig" i jako pierwszy zaktualizuje współdzieloną zmienną.

Czy ten problem rzeczywiście się pojawia? Oczywiście. Uruchom program demonstracyjny z kodów dołączonych do książki. Ma on 100 wątków, każdy zwiększający licznik 1000 razy i wyświetlający wynik.

```
for (int i = 1; i <= 100; i++) {
    int idZadania = i;
    Runnable zadanie = () -> {
        for (int k = 1; k <= 1000; k++)
            licznik++;
        System.out.println(idZadania + ": " + licznik);
    };
    executor.execute(zadanie);
}
```

Wyniki zaczynają się od dobrze wyglądających wartości w stylu

```
1: 1000
3: 2000
2: 3000
6: 4000
```

Po chwili zaczyna wyglądać to trochę niepokojąco:

```
72: 58196
68: 59196
73: 61196
71: 60196
69: 62196
```

Przyczyną tego może być na przykład to, że niektóre wątki zostały zatrzymane w nieodpowiednich momentach. Ważne jest, co dzieje się z zadaniem, które zakończyło działanie jako ostatnie. Czy licznik doszedł do wartości `100000`?

Uruchomiłem program wiele razy na moim wielordzeniowym laptopie i nigdy mu się to nie udało. Lata temu, gdy komputery osobiste miały pojedynczy procesor, wyścigi było dużo trudniej zaobserwować i programiści nie zauważali tak dużych problemów często. Nie ma jednak znaczenia, czy zła wartość pojawi się w ciągu sekund, czy godzin.

Ten przykład pokazuje prosty przypadek współdzielonego licznika w przykładowym programie. Ćwiczenie 17. pokazuje ten sam problem w realistycznym przypadku. Nie chodzi tutaj jednak wyłącznie o liczniki. Wyścigi są problemem zawsze, gdy modyfikowane są współdzielone zmienne. Na przykład przy dodawaniu wartości do początku kolejki kod odpowiedzialny za jej wstawienie może wyglądać tak:

```
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n;
tail.value = newValue;
```

Wiele rzeczy może pójść źle, jeśli taki ciąg instrukcji będzie zatrzymany w nieszczęśliwie wybranym momencie i inne zadanie przejmie kontrolę, uzyskując dostęp do kolejki w chwili, gdy jest ona modyfikowana.

Wykonaj ćwiczenie 21., by poczuć, w jaki sposób struktura danych może zostać uszkodzona przez równoległe wprowadzanie zmian.

Trzeba się upewnić, że cała sekwencja operacji zostanie wykonana razem. Taka sekwencja instrukcji nazywana jest sekcją krytyczną. Możesz wykorzystać blokady, by chronić sekcje krytyczne, a krytyczne ciągi operacji uczynić atomowymi. Nauczysz się programowania z blokadami w punkcie 10.7.1, „Blokady”.

Choć korzystanie z blokad do ochrony sekcji krytycznych jest proste, blokady nie są ogólnym rozwiązaniem wszystkich problemów związanych ze współbieżnością. Poprawne ich użycie jest trudne i łatwo popełnić błędy, które znacząco obniżą wydajność, lub nawet spowodować zakleszczenie (ang. *deadlock*).

10.3.3. Strategie bezpiecznego korzystania ze współbieżności

W językach takich jak C i C++ programiści muszą manualnie alokować i zwalniać pamięć. Brzmi to niebezpiecznie i takie jest. Wielu programistów w przykry sposób spędziło wiele godzin na poszukiwaniu błędów związanych z alokacją pamięci. W języku Java istnieje mechanizm *garbage collector* i niewielu programistów Javy musi martwić się zarządzaniem pamięcią.

Niestety, nie ma podobnego mechanizmu obsługującego dostęp do współdzielonych danych w programach wielowątkowych. Najlepsze, co możesz zrobić, to przestrzegać zaleceń pozwalających na zarządzanie nieuchronnymi niebezpieczeństwami.

Bardzo efektywną strategią jest **ograniczanie**. Po prostu unikaj współdzielenia danych pomiędzy zadaniami. Na przykład: gdy Twoje zadania muszą coś zliczać, utwórz w każdym z nich oddzielny licznik, zamiast aktualizować wspólny licznik. Gdy zadania zakończą działanie, niech przekażą swoje wyniki do innego zadania, które je połączy.

Inną dobrą strategią jest korzystanie z obiektów **niemodyfikowalnych**. Współdzielenie niemodyfikowalnych obiektów jest bezpieczne. Na przykład zamiast dodawać wyniki do współdzielonej kolekcji, zadanie może generować niemodyfikowalne kolekcje wyników. Inne zadanie łączy wyniki w kolejnej niemodyfikowalnej strukturze danych. Idea jest prosta, ale na kilka rzeczy trzeba uważać — zobacz punkt 10.3.4, „Klasy niemodyfikowalne".

Trzecią strategią jest stosowanie **blokad**. Dając tylko jednemu zadaniu dostęp do danych w danej chwili, można uchronić je przed uszkodzeniem. W podrozdziale 10.5, „Struktury danych bezpieczne dla wątków", zobaczysz dostarczane przez bibliotekę języka Java, wspierającą współbieżność, struktury danych, których można bezpiecznie używać w programach wielowątkowych. Punkt 10.7.1, „Blokady", pokazuje, w jaki sposób działa blokowanie i jak eksperci tworzą takie struktury danych.

Blokowanie jest podatne na błędy i może być drogie, ponieważ ogranicza możliwości równoczesnego wykonywania. Jeśli na przykład masz wiele zadań dostarczających wyniki do wspólnej tablicy skrótów i jest ona blokowana przy każdej aktualizacji, jest to prawdziwe wąskie gardło. Jeśli większość zadań musi czekać na swoją kolej, nie wykonują one użytecznej pracy. Czasem możliwe jest **partycjonowanie** danych w taki sposób, by do różnych fragmentów można było odwoływać się w tym samym czasie. Kilka struktur danych z biblioteki wspierającej współbieżność w języku Java korzysta z partycjonowania, tak jak równoległe algorytmy z biblioteki obsługującej strumienie. Nie próbuj tego w domu! Naprawdę trudno wykonać to poprawnie. Zamiast tego korzystaj ze struktur danych i z algorytmów z biblioteki języka Java.

10.3.4. Klasy niemodyfikowalne

Klasa jest niemodyfikowalna, gdy jej instancje po utworzeniu nie mogą się zmieniać. Pierwsze wrażenie jest takie, że nie są one zbyt przydatne, ale to nieprawda. Popularna klasa `String` jest niemodyfikowalna, tak samo jak klasy z bibliotek obsługujących datę i czas (więcej w rozdziale 12.). Żadna instancja obiektu opisującego datę nie jest modyfikowalna, ale możesz za jej pomocą tworzyć kolejne instancje, na przykład instancję opisującą następny dzień.

Popatrzmy na zestaw przechowujący wyniki. Możesz wykorzystać modyfikowalny obiekt klasy `HashSet` i aktualizować go poleceniem

```
results.addAll(newResults);
```

Jest to jednak ewidentnie niebezpieczne.

Niemodyfikowalny zbiór zawsze tworzy nowe zbiory. Wyniki aktualizujesz wtedy w taki sposób:

```
results = results.union(newResults);
```

To ciągle się zmienia, ale jest dużo prościej kontrolować, co stanie się z jedną zmienną niż z całym zbiorem mającym wiele metod.

Implementowanie niemodyfikowalnych klas nie jest trudne, ale powinieneś zwrócić uwagę na takie problemy:

1. Nie zmieniaj stanu obiektu po jego utworzeniu. Upewnij się, że deklarujesz zmienne instancji z modyfikatorem `final`. Nie ma powodu, by tego nie robić, a zyskujesz ważną korzyść. Maszyna wirtualna zapewnia, że zmienna instancji typu `final` jest widoczna po utworzeniu (punkt 10.3.1, „Widoczność").
2. Oczywiście żadna z metod nie może modyfikować danych. Powinieneś oznaczyć je jako `final` lub, lepiej, zadeklarować klasę jako `final`, by metody modyfikujące zmienne nie mogły być dodawane w klasach potomnych.
3. Nie pozwól przeciekać stanom, które mogą być modyfikowane na zewnątrz. Żadna z Twoich (innych niż prywatne) metod nie może zwracać referencji do elementów wewnętrznych, które mogą być wykorzystane do wprowadzenia modyfikacji, jak wewnętrzna tablica czy kolekcja. Gdy jedna z Twoich metod wywołuje metodę innej klasy, nie może przekazywać żadnych tego typu referencji, ponieważ wywoływana metoda mogłaby w takim przypadku wykorzystać ją do wprowadzenia modyfikacji. Zamiast tego przekaż kopię.
4. Tak samo nie przechowuj żadnej referencji do modyfikowalnego obiektu przekazanej do konstruktora. Zamiast tego wykonaj kopię.
5. Nie pozwól, by referencja `this` wyszła poza konstruktor. Gdy wywołujesz inną metodę, wiesz, że nie należy przekazywać żadnych wewnętrznych referencji, ale co z `this`? Jest ona bezpieczna po utworzeniu klasy, ale jeśli ujawnisz `this` w konstruktorze, ktoś mógłby śledzić niekompletny obiekt. Pamiętaj też o konstruktorach przekazujących referencje do wewnętrznych klas zawierających ukrytą referencję `this`. Oczywiście takie sytuacje są dość rzadkie.

10.4. Algorytmy równoległe

Przed rozpoczęciem zrównoleglania przetwarzania danych powinieneś sprawdzić, czy w bibliotece języka Java nie zostało to już wykonane. Biblioteka strumieni lub klasa `Arrays` może już zawierać implementację tego, co jest Ci potrzebne.

10.4.1. Strumienie równoległe

Biblioteka strumieni automatycznie zrównolegla operacje na wielkich strumieniach równoległych. Na przykład jeśli `coll` jest dużą kolekcją ciągów znaków i chcesz ustalić, jak wiele z nich zaczyna się od litery `A`, wywołaj

```
long result = coll.parallelStream().filter(s -> s.startsWith("A")).count();
```

Metoda `parallelStream` zwraca strumień równoległy. Strumień jest podzielony na segmenty. Filtrowanie i zliczanie jest wykonywane dla każdego segmentu, a wyniki są łączone. Nie musisz martwić się o szczegóły.

Gdy korzystasz ze strumieni równoległych z funkcjami lambda (na przykład jako argumentów do filtrowania i mapowania we wcześniejszych przykładach), trzymaj się z dala od niebezpiecznych modyfikacji współdzielonych obiektów.

Aby strumienie równoległe dobrze działały, należy spełnić szereg warunków:

- Musi istnieć odpowiednia ilość danych. Strumienie równoległe dodają znaczący narzut, który zwraca się jedynie w przypadku dużych zbiorów danych.
- Dane muszą być przechowywane w pamięci. Oczekiwanie na załadowanie danych byłoby nieefektywne.
- Strumień powinien być łatwo podzielny na podzbiory. Strumień tworzony z tablicy lub zbalansowanego drzewa binarnego sprawdza się dobrze, ale lista powiązana lub wynik `Stream.iterate` — już nie.
- Operacje strumienia powinny wykonywać znaczącą część pracy. Jeśli całkowita praca nie jest duża, nie ma sensu narażać się na dodatkowy koszt przygotowywania operacji równoległych.
- Operacje strumienia nie powinny mieć blokad.

Innymi słowy, nie zamieniaj wszystkich swoich strumieni w strumienie równoległe. Korzystaj ze strumieni równoległych tylko, jeśli wykonujesz znaczącą część przetwarzania na danych, które znajdują się już w pamięci.

10.4.2. Równoległe operacje na tablicach

Klasa `Arrays` ma wiele zrównoleglonych operacji. Tak jak w przypadku równoległych operacji na strumieniach z poprzedniego punktu, operacje dzielą tablicę na fragmenty, przetwarzają je równocześnie i łączą wyniki.

Statyczna metoda `Arrays.parallelSetAll` wypełnia tablicę wartościami obliczonymi przez funkcję. Funkcja odbiera indeks elementu i oblicza wartość w tym miejscu.

```
Arrays.parallelSetAll(wartości, i -> i % 10);
    // Wypełnia wartości cyframi 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Widać, że ta operacja korzysta ze zrównoleglenia. Istnieją jej odmiany dla wszystkich tablic zawierających typy proste i dla tablic obiektów.

Metoda `parallelSort` może sortować tablicę zmiennych typów prostych lub obiektów. Na przykład

```
Arrays.parallelSort(słowa, Comparator.comparing(String::length));
```

Za pomocą wszystkich metod możesz dostarczyć ograniczenia zakresu, takie jak

```
Arrays.parallelSort(wartości, wartości.length / 2, wartości.length); // Sortuje górną połowę
```

Na pierwszy rzut oka wygląda trochę dziwnie, że metody te mają `parallel` w nazwach — użytkownik nie powinien zajmować się tym, w jaki sposób wykonywane jest ustawianie lub sortowanie. Projektanci API chcieli jednak wyjaśnić to, że operacje są zrównoleglane. W ten sposób użytkownicy dostają ostrzeżenie, by unikać funkcji generujących lub porównujących z efektami ubocznymi.

W końcu istnieje też metoda `parallelPrefix`, która jest bardziej wyspecjalizowana — prosty przykład daje ćwiczenie 4.

Aby wykonywać inne zrównoleglone operacje na tablicach, przekształć je w równoległe strumienie. Na przykład: aby obliczyć sumę długiej tablicy liczb całkowitych, wywołaj

```
long suma = IntStream.of(wartości).parallel().sum();
```

10.5. Struktury danych bezpieczne dla wątków

Jeśli wiele wątków równocześnie modyfikuje strukturę danych taką jak kolejka lub tablica skrótów, łatwo uszkodzić wewnętrzny stan struktury danych. Na przykład jeden wątek może rozpocząć dodawanie nowego elementu. Załóżmy, że zostanie on wywłaszczony podczas zmiany powiązań i inny wątek zacznie modyfikować to samo miejsce. Drugi wątek może wykorzystać niewłaściwe odnośniki i zrobić spustoszenie, prawdopodobnie wyrzucając wyjątki lub może nawet wpadając w nieskończoną pętlę.

Jak zobaczysz w punkcie 10.7.1, „Blokady”, możesz korzystać z blokad, by upewnić się, że tylko jeden wątek może mieć dostęp do struktury danych, w danej chwili blokując wszystkie inne. Możesz jednak zrobić to lepiej. Kolekcje z pakietu `java.util.concurrent` zostały sprytnie zaimplementowane, tak że wiele wątków może z nich korzystać bez wzajemnego blokowania się, pod warunkiem że będą uzyskiwały dostęp do różnych części struktury danych.

Te kolekcje zwracają iteratory o **małej spójności**. Oznacza to, że iteratory prezentują elementy, które istniały na początku iteracji i mogą (ale nie muszą) odzwierciedlać niektóre lub wszystkie modyfikacje wykonane po ich utworzeniu. Taki iterator nie wyrzuci jednak wyjątku `ConcurrentModificationException`.

W odróżnieniu od tego iterator kolekcji z pakietu `java.util` wyrzuca wyjątek `ConcurrentModificationException`, gdy kolekcja zostanie zmodyfikowana po utworzeniu iteratora.

10.5.1. Klasa ConcurrentHashMap

Klasa `ConcurrentHashMap` jest przede wszystkim mapą skrótów, na której operacje są bezpieczne dla wątków. Niezależnie od tego, ile wątków będzie pracować na mapie w tej samej chwili, jej wnętrze nie zostanie uszkodzone. Oczywiście niektóre wątki mogą być tymczasowo blokowane, ale mapa może wydajnie wspierać dużą liczbę równoległych odczytów i pewną liczbę równoległych zapisów.

To jednak nie wystarczy. Przypuśćmy, że chcemy wykorzystać mapę do zliczania, jak często pewne mechanizmy są obserwowane. Przykładowo załóżmy, że wiele wątków zlicza słowa i chcemy obliczyć częstotliwość ich występowania. Oczywiście poniższy kod aktualizujący licznik nie jest bezpieczny dla wątków:

```
ConcurrentHashMap<String, Long> mapa = new ConcurrentHashMap<>();
...
Long staraWartość = mapa.get(słowo);
Long nowaWartość = staraWartość == null ? 1 : staraWartość + 1;
mapa.put(word, nowaWartość); // Błąd — może nie zastąpić staraWartość
```

Inny wątek może aktualizować dokładnie ten sam licznik w tej samej chwili.

Aby bezpiecznie zaktualizować wartość, wykorzystaj metodę `compute`. Jest ona wywoływana z kluczem i funkcją obliczającą nową wartość. Funkcja ta pobiera klucz i związaną z nim wartość lub wartość `null`, jeśli takiego klucza nie ma, oraz oblicza nową wartość. Na przykład w taki sposób możemy zaktualizować licznik:

```
mapa.compute(słowo, (k, v) -> v == null ? 1 : v + 1);
```

Metoda `compute` jest **atomowa** — żaden inny wątek nie może zmodyfikować elementu mapy podczas wykonywania operacji.

Istnieją też odmiany `computeIfPresent` oraz `computeIfAbsent`, które obliczają nową wartość, jeśli istnieje stara wartość lub jeśli takiej wartości jeszcze nie ma.

Inną atomową operacją jest `putIfAbsent`. Licznik może być inicjalizowany jako

```
map.putIfAbsent(słowo, 0L);
```

Często potrzebujesz wykonać coś specjalnego po dodaniu klucza po raz pierwszy. Metoda `merge` sprawia, że jest to szczególnie wygodne. Zawiera ona parametr dla początkowej wartości, który jest wykorzystywany, jeśli klucz nie jest jeszcze obecny. W innym przypadku wywoływana jest funkcja, którą dostarczyłeś, łącząca istniejącą wartość i wartość początkową. (W przeciwieństwie do `compute` funkcja nie przetwarza klucza).

```
mapa.merge(słowo, 1L, (istniejącaWartość, nowaWartość) -> istniejącaWartość +
nowaWartość);
```

lub po prostu

```
mapa.merge(słowo, 1L, Long::sum);
```

Oczywiście funkcje przekazane do `compute` i `merge` powinny działać szybko i nie powinny próbować modyfikować mapy.

Istnieją metody, które w sposób atomowy usuwają lub zastępują element, jeśli jest on w danej chwili taki sam jak istniejący. Przed udostępnieniem metody `compute` ludzie pisali kod zwiększający licznik w taki sposób:

```
do {
    staraWartość = map.get(słowo);
    nowaWartość = staraWartość + 1;
} while (!mapa.replace(word, staraWartość, nowaWartość));
```

Istnieje kilka **operacji masowych** do przeszukiwania, przekształcania lub przeglądania `ConcurrentHashMap`. Działają one na zrzucie danych i mogą być bezpiecznie wykonywane nawet w sytuacji, gdy inne wątki pracują na mapie. W dokumentacji API szukaj operacji, których nazwy zaczynają się od `search`, `reduce` i `forEach`. Istnieją odmiany, które działają na kluczach, wartościach i elementach. Metody `reduce` mają też wersje dla funkcji redukujących wartości typu `int`, `long` i `double`.

10.5.2. Kolejki blokujące

Jednym z częściej wykorzystywanych narzędzi do synchronizowania pracy zadań jest **kolejka blokująca**. Zadanie dostarczające dane umieszcza elementy w kolejce, a zadanie odbierające dane pobiera je z kolejki. Kolejka pozwala bezpiecznie przekazywać dane z jednego zadania do innego.

Jeśli próbujesz dodać element w sytuacji, gdy kolejka jest pełna, lub próbujesz usunąć element z pustej kolejki, operacja blokuje działanie. W ten sposób kolejka rozkłada obciążenie. Jeśli zadanie dostarczające dane działa wolniej niż zadanie odbierające, to drugie jest blokowane i czeka na nowe wyniki. Jeśli zadania dostarczające dane działają szybciej, kolejka zatyka się do chwili, gdy zadanie odbierające dane nadrobi zaległości.

Tabela 10.3 pokazuje metody kolejek blokujących. Metody te dzielą się na trzy kategorie różniące się działaniem wykonywanym w sytuacji, gdy kolejka jest pełna lub pusta. Dodatkowo poza metodami blokującymi istnieją też metody wyrzucające wyjątek w przypadku niepowodzenia oraz metody zwracające informację o niepowodzeniu zamiast wyrzucania wyjątku, jeśli nie mogą wykonać swoich zadań.

Tabela 10.3. *Operacje kolejek blokujących*

Metoda	Normalne działanie	Działanie w przypadku problemów
`put`	Dodaje element na koniec	Blokuje, jeśli kolejka jest pełna
`take`	Usuwa i zwraca pierwszy element	Blokuje, jeśli kolejka jest pusta
`add`	Dodaje element na koniec	Wyrzuca `IllegalStateException`, jeśli kolejka jest pełna
`remove`	Usuwa i zwraca pierwszy element	Wyrzuca `NoSuchElementException`, jeśli kolejka jest pusta
`element`	Zwraca pierwszy element	Wyrzuca `NoSuchElementException`, jeśli kolejka jest pusta
`offer`	Dodaje element i zwraca `true`	Zwraca `false`, jeśli kolejka jest pełna
`poll`	Usuwa i zwraca pierwszy element	Zwraca `null`, jeśli kolejka jest pusta
`peek`	Zwraca pierwszy element	Zwraca `null`, jeśli kolejka jest pusta

Metody `poll` i `peek` zwracają `null`, by zasygnalizować niepowodzenie. Dlatego wstawianie wartości `null` do takich kolejek nie jest poprawne.

Istnieją też odmiany metod `offer` i `poll` z ograniczeniem czasowym. Na przykład wywołanie

```
boolean sukces = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

próbuje przez 100 milisekund wstawić element na koniec kolejki. Jeśli się to uda, zwraca `true`; w innym przypadku zwraca `false` po upływie wskazanego czasu. Podobnie wywołanie

```
Object głowa = q.poll(100, TimeUnit.MILLISECONDS)
```

próbuje przez 100 milisekund pobrać pierwszy element z kolejki. Jeśli się to uda, zwraca ten element; w innym przypadku zwraca `null` po upływie wskazanego czasu.

Pakiet `java.util.concurrent` dostarcza kilka odmian kolejek blokujących. Kolejka `LinkedBlockingQueue` jest oparta na liście powiązanej, a `ArrayBlockingQueue` korzysta z tablicy rotacyjnej.

Ćwiczenie 11. pokazuje, w jaki sposób korzystać z kolejek blokujących przy analizowaniu plików z katalogu. Jeden wątek przechodzi przez drzewo katalogów i wstawia pliki do kolejki. Kilka wątków usuwa pliki i je przeszukuje. W takiej aplikacji prawdopodobnie wątek tworzący elementy szybko wypełni kolejkę plikami i zostanie zablokowany do czasu, gdy wątki odbierające elementy wykonają swoją pracę.

Typowym wyzwaniem w takim przypadku jest zatrzymanie wątków odbierających elementy (konsumentów). Taki wątek nie może po prostu kończyć działania, gdy kolejka zostanie opróżniona. W końcu wątek uzupełniający elementy (producent) mógł jeszcze nie wystartować lub może działać wolniej. Jeśli istnieje jeden taki wątek, może on wstawiać do kolejki wskaźnik oznaczający „ostatni element”, podobnie jak czasem umieszcza się imitację walizki z napisem „ostatni bagaż” na pasie do odbierania bagażu.

10.5.3. Inne struktury danych bezpieczne dla wątków

Tak jak możesz wybierać pomiędzy mapami skrótów i drzewami w pakiecie `java.util`, tak istnieje przystosowana do pracy współbieżnej mapa o nazwie `ConcurrentSkipListMap`, której działanie opiera się na porównywaniu kluczy. Możesz ją wykorzystać, jeśli musisz przejść przez klucze w kolejności sortowania lub jeśli potrzebujesz jednej z dodatkowych metod z interfejsu `NavigableMap` (patrz rozdział 7.). Istnieje też podobny `ConcurrentSkipListSet`.

Kolekcje `CopyOnWriteArrayList` i `CopyOnWriteAraySet` są bezpieczne dla wątków dzięki temu, że wszystkie metody modyfikujące wykonują kopię wykorzystywanej tablicy. Takie działanie jest korzystne, jeśli liczba wątków przechodzących przez kolekcję jest znacząco większa niż liczba wątków, które ją modyfikują. Gdy tworzysz iterator, zawiera on referencję do bieżącej tablicy. Jeśli tablica zostanie później zmodyfikowana, iterator nadal ma starą tablicę, ale tablica kolekcji jest zamieniana. W konsekwencji starszy iterator ma spójny (choć potencjalnie nieaktualny) obraz, do którego może uzyskać dostęp bez dodatkowych kosztów związanych z synchronizacją.

Przypuśćmy, że potrzebujesz dużego, bezpiecznego dla wątków zestawu zamiast mapy. Nie ma klasy `ConcurrentHashSet` i dobrze wiesz, że nie warto tworzyć własnej wersji. Oczywiście możesz wykorzystać `ConcurrentHashMap` z przypadkowymi wartościami, ale daje Ci to mapę, a nie zestaw, i nie możesz korzystać z operacji dostarczanych przez interfejs `Set`.

Statyczna metoda newKeySet zwraca Set<K>, który w rzeczywistości opakowuje ConcurrentHashMap<K, Boolean>. (Wszystkie wartości mapy to Boolean.TRUE, ale nie ma to dla Ciebie znaczenia, ponieważ korzystasz z tego jak z zestawu).

```
Set<String> słowa = ConcurrentHashMap.newKeySet();
```

Jeśli masz istniejącą mapę, metoda keySet zwraca zestaw kluczy. Taki zestaw jest modyfikowalny. Jeśli usuniesz elementy zestawu, klucze (i ich wartości) zostaną usunięte z mapy. Nie ma jednak sensu dodawanie elementów do zestawu kluczy, ponieważ nie ma odpowiadających im wartości do dodania. Możesz użyć drugiej metody keySet, wykorzystując domyślną wartość przy dodawaniu elementów do zestawu:

```
Set<String> słowa = map.keySet(1L);
słowa.add("Java");
```

Jeśli nie było wcześniej w słowa słowa "Java", ma ono teraz przypisaną wartość jeden.

10.6. Atomowe liczniki i akumulatory

Jeśli wiele wątków aktualizuje wspólny licznik, musisz się upewnić, że będzie to wykonywane w sposób bezpieczny dla wątków. W pakiecie java.util.concurrent.atomic istnieje wiele klas, które korzystają z bezpiecznych i wydajnych instrukcji niskopoziomowych, aby zagwarantować atomowość działań na liczbach typu int, long i boolean, referencjach do obiektów oraz tablicach tych wartości. Poprawne wykorzystanie tych klas wymaga pewnego doświadczenia. Atomowe liczniki i akumulatory są mimo to wygodnym rozwiązaniem przy programowaniu aplikacji.

Na przykład możesz bezpiecznie tworzyć ciągi liczb, takie jak ten:

```
public static AtomicLong następnaLiczba = new AtomicLong();
// W jakimś wątku . . .
long id = następnaLiczba.incrementAndGet();
```

Metoda incrementAndGet atomowo zwiększa AtomicLong i zwraca wartość po inkrementacji. Oznacza to, że operacje pobrania wartości, dodania 1, zapamiętania jej i utworzenia nowej wartości nie mogą zostać przerwane. Gwarantowane jest, że poprawna wartość jest obliczana i zwracana, nawet jeśli wiele wątków korzysta równocześnie z tej samej instancji.

Istnieją metody służące do atomowego ustawiania, dodawania i odejmowania wartości, ale załóżmy, że chcesz dokonać bardziej skomplikowanej aktualizacji. Jednym ze sposobów jest użycie metody updateAndGet. Dla przykładu załóżmy, że chcesz śledzić największą wartość znalezioną przez różne wątki. Poniższy kod nie zadziała:

```
public static AtomicLong największa = new AtomicLong();
// W jakimś wątku . . .
największa.set(Math.max(największa.get(), observed)); // Błąd — wyścig!
```

Ta aktualizacja nie jest atomowa. Zamiast tego wywołaj updateAndGet z wyrażeniem lambda aktualizującym wartość. W naszym przykładzie możemy wywołać

```
największa.updateAndGet(x -> Math.max(x, observed));
```

lub

```
największa.accumulateAndGet(observed, Math::max);
```

Metoda `accumulateAndGet` pobiera binarny operator, który jest wykorzystywany do porównania atomowej wartości i dostarczonego argumentu.

Istnieją też metody `getAndUpdate` oraz `getAndAccumulate`, zwracające starą wartość.

Te metody są również dostępne w klasach: `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray` oraz `AtomicReferenceFieldUpdater`.

Gdy masz bardzo dużą liczbę wątków korzystających z tych samych wartości atomowych, obniża się wydajność, ponieważ aktualizacje są wykonywane **optymistycznie**. Oznacza to, że operacja oblicza nową wartość na podstawie podanej starej wartości, a następnie zamienia wartość, pod warunkiem że aktualna wartość jest równa starej wartości. Jeśli tak nie jest, proces się powtarza. Przy dużym obciążeniu aktualizacja wymaga zbyt wielu powtórek.

Klasy `LongAdder` oraz `LongAccumulator` rozwiązują ten problem dla pewnych typowych aktualizacji. Klasa `LongAdder` składa się z wielu zmiennych, których wspólna suma jest aktualną wartością. Wiele wątków może aktualizować różne składniki sumy, a nowe składniki są tworzone, gdy zwiększa się liczba wątków. Jest to wydajne w typowych zastosowaniach, w których wartość sumy nie jest potrzebna do czasu zakończenia pracy. Poprawa wydajności może być znacząca — patrz ćwiczenie 9.

Jeśli przewidujesz dużą rywalizację, powinieneś po prostu użyć `LongAdder` zamiast `AtomicLong`. Nazwy metod odrobinę się różnią. Wywołaj `increment`, by zwiększyć licznik, lub `add`, by dodać wartość, i `sum`, by pobrać wynik.

```
final LongAdder licznik = new LongAdder();
for (...)
    executor.execute(() -> {
        while (...) {
            ...
            if (...) count.increment();
        }
    });
...
long total = count.sum();
```

Oczywiście metoda `increment` *nie* zwraca starej wartości. Takie działanie zniwelowałoby zysk wydajności wynikający z podzielenia sumy na wiele składników.

Klasa `LongAccumulator` uogólnia tę ideę na dowolne działania związane z gromadzeniem. W konstruktorze dostarczasz operację wraz z elementem neutralnym. Aby dołączyć nowe wartości, wywołaj `accumulate`. Wywołaj `get`, by uzyskać bieżącą wartość.

```
LongAccumulator accumulator = new LongAccumulator(Long::sum, 0);
// W pewnych zadaniach . . .
accumulator.accumulate(wartość);
// Po zakończeniu zadań
long suma = accumulator.get();
```

Wewnętrznie akumulator ma zmienne: a_1, a_2, ..., a_n. Każda zmienna jest inicjalizowana elementem neutralnym (`0` w naszym przykładzie).

Po wywołaniu `accumulate` z wartością *v* jedna z nich jest atomowo aktualizowana działaniem $a_i = a_i \; op \; v$, gdzie *op* oznacza operację gromadzenia zapisaną w notacji infiksowej. W naszym przykładzie wywołanie `accumulate` oblicza $a_i = a_i + v$ dla pewnych *i*.

Wynikiem `get` jest $a_1 \; op \; a_2 \; op \; ... \; op \; a_n$. W naszym przykładzie jest to suma wartości zmiennych $a_1+a_2+...+a_n$.

Jeśli wybierzesz inną operację, możesz obliczyć wartość maksymalną lub minimalną (patrz ćwiczenie 10.). Ogólnie operacja musi być łączna i przemienna. Oznacza to, że ostateczny wynik musi być niezależny od kolejności, w jakiej pośrednie wartości były łączone.

Istnieją też klasy `DoubleAdder` i `DoubleAccumulator`, które działają w ten sam sposób, tyle że dla wartości typu `double`.

Jeśli korzystasz z tablicy skrótów zawierającej elementy typu `LongAdder`, możesz użyć poniższego zwrotu, by zwiększyć wartość dla wybranego klucza:

```
ConcurrentHashMap<String,LongAdder> liczniki = ...;
liczniki.computeIfAbsent(klucz, k -> new LongAdder()).increment();
```

Gdy licznik dla klucza jest zwiększany po raz pierwszy, tworzona jest dla niego nowa instancja `LongAdder`.

10.7. Blokady i warunki

Widziałeś już kilka narzędzi, których mogą bezpiecznie używać programiści tworzący aplikacje korzystające z operacji równoległych. Możesz być ciekaw, jak można utworzyć bezpieczny dla wątków licznik lub kolejkę blokującą. Kolejne punkty pokażą Ci, w jaki sposób jest to realizowane, abyś mógł zrozumieć koszty i stopień komplikacji.

10.7.1. Blokady

Aby uniknąć uszkodzenia współdzielonych zmiennych, należy upewnić się, że tylko jeden wątek w danej chwili może obliczać i ustawiać nową wartość. Kod, który musi być wykonany w całości bez przerwy, jest nazywany **sekcją krytyczną** (ang. *critical section*). Można wykorzystać *blokadę*, by zaimplementować sekcję krytyczną:

```
Lock blokadaLicznika = new ReentrantLock(); // Współdzielony przez wiele wątków
int licznik; // Współdzielony przez wiele wątków
...
blokadaLicznika.lock();
try {
    licznik++; // Sekcja krytyczna
} finally {
    blokadaLicznika.unlock(); // Upewnij się, że blokada jest zwolniona
}
```

W tym punkcie korzystam z klasy `ReentrantLock`, by wyjaśnić, w jaki sposób działa blokowanie. Jak zobaczysz w kolejnym punkcie, w wielu przypadkach nie jest konieczne korzystanie z jawnych blokad, ponieważ istnieją „wewnętrzne" blokady, z których można korzystać za pomocą słowa kluczowego `synchronized`. Prościej jest jednak zrozumieć, co dzieje się w środku, jeśli spojrzeć na działanie jawnie zapisanej blokady.

Pierwszy wątek chcący wykonać metodę `lock` blokuje obiekt `countLock` i przechodzi do wykonania krytycznej sekcji. Jeśli inny wątek próbuje wywołać `lock` na tym samym obiekcie, jest on blokowany do chwili, gdy pierwszy wątek wywoła `unlock`. W ten sposób mamy gwarancję, że tylko jeden wątek w danej chwili może wykonywać kod sekcji krytycznej.

Zauważ, że dzięki umieszczeniu wywołania metody `unlock` w klauzuli `finally` blokada jest zwalniana, jeśli w sekcji krytycznej pojawi się jakiś wyjątek. W innym przypadku blokada zostałaby ustawiona na stałe i żaden inny wątek nie mógłby przez nią przejść — co byłoby bardzo złą sytuacją. Oczywiście w takim przypadku sekcja krytyczna nie może wyrzucić wyjątku, ponieważ wykonuje ona jedynie zwiększenie wartości całkowitej. Warto jednak w takich przypadkach mimo wszystko korzystać z klauzuli `try/finally`, w razie gdyby później dodano więcej kodu.

Na pierwszy rzut oka wygląda na to, że wykorzystanie blokad do chronienia sekcji krytycznych jest wystarczająco proste. Jednak diabeł tkwi w szczegółach. Doświadczenie pokazało, że wielu programistów ma trudności z pisaniem poprawnego kodu korzystającego z blokad. Mogą używać złych blokad lub tworzyć sytuacje **zakleszczenia**, w których żaden z wątków nie może działać, ponieważ wszystkie czekają na zwolnienie blokady.

Z tego powodu programiści aplikacji powinni korzystać z blokad w ostateczności. W pierwszej kolejności należy unikać współdzielenia, korzystając z niemodyfikowalnych danych lub przekazując modyfikowalne dane z jednego wątku do drugiego. Jeśli musisz współdzielić, korzystaj z gotowych, bezpiecznych dla wątków struktur, takich jak `ConcurrentHashMap` lub `LongAdder`. Mimo to warto znać blokady, by rozumieć, w jaki sposób takie struktury danych mogą być implementowane.

10.7.2. Słowo kluczowe synchronized

W poprzednim punkcie pokazałem Ci, w jaki sposób korzystać z `ReentrantLock` do implementowania sekcji krytycznej. Nie musisz korzystać z jawnej blokady, ponieważ w języku Java *każdy obiekt* ma **wewnętrzną blokadę**. Zrozumienie działania wewnętrznych blokad ułatwia jednak wcześniejsze przyjrzenie się jawnym blokadom.

Słowo kluczowe `synchronized` jest wykorzystywane do aktywowania wewnętrznej blokady. Może to być wykonane na dwa sposoby. Możesz zablokować blok:

```
synchronized (obj) {
    Sekcja krytyczna
}
```

Co jest równoważne

```
obj.wewnętrznaBlokada.lock();
try {
    Sekcja krytyczna
} finally {
    obj. wewnętrznaBlokada.unlock();
}
```

Obiekt nie ma w rzeczywistości pola, które jest wewnętrzną blokadą. Kod ten tylko pokazuje, co się dzieje, gdy korzystasz ze słowa kluczowego `synchronized`.

Możesz też zadeklarować metodę ze słowem kluczowym `synchronized`. Wtedy jej ciało jest blokowane na parametrze `this`. Czyli

```
public synchronized void method() {
    Ciało metody
}
```

jest równoważne z

```
public void method() {
    this. wewnętrznaBlokada.lock();
    try {
        Ciało metody
    } finally {
        this. wewnętrznaBlokada.unlock();
    }
}
```

Przykładowo licznik może być po prostu zadeklarowany jako

```
public class Licznik {
    private int wartość;
    public synchronized int zwiększ() {
        wartość++;
        return wartość;
    }
}
```

Dzięki wykorzystaniu wewnętrznych blokad instancji `Licznik` nie ma potrzeby tworzenia jawnych blokad.

Jak widzisz, wykorzystanie słowa kluczowego pozwala tworzyć dość przejrzysty kod. Oczywiście aby zrozumieć ten kod, musisz wiedzieć, że każdy obiekt ma wewnętrzną blokadę.

Wymuszanie atomowości to niejedyne zastosowanie blokad. Dbają one również o widoczność. Na przykład rozważ zmienną `done`, sprawiającą tak wiele problemów w punkcie 10.3.1, „Widoczność”. Jeśli korzystasz z blokad zarówno do zapisywania, jak i odczytywania zmiennej, masz pewność, że wywołujący `get` widzi wszystkie aktualizacje zmiennej wykonywane wywołaniami `set`.

```
public class Flag {
    private boolean done;
    public synchronized void set() { done = true; }
    public synchronized boolean get() { return done; }
}
```

Inspiracją do utworzenia metod synchronizowanych była koncepcja **monitora**, którą zaproponowali Per Brinch Hansen i Tony Hoare w latach 70. ubiegłego wieku. Monitor jest w istocie klasą, w której wszystkie zmienne instancji są prywatne i wszystkie metody są zabezpieczone prywatną blokadą.

W języku Java możliwe jest posiadanie publicznych zmiennych instancji i łączenie metod synchronizowanych z niesynchronizowanymi. Bardziej problematyczne jest to, że wewnętrzna blokada jest dostępna dla wszystkich.

Wielu programistów uznało to za mylące. Na przykład Java 1.0 ma klasę `Hashtable` z synchronizowanymi metodami modyfikującymi tablicę. Aby bezpiecznie przechodzić przez taką tablicę, możesz pobrać blokadę w taki sposób:

```
synchronized (tablica) {
    for (K key : tablica.keySet()) ...
}
```

Tutaj `tablica` oznacza zarówno tablicę skrótów, jak i blokadę, z której korzystają jej metody. Jest to typowe źródło nieporozumień — patrz ćwiczenie 22.

10.7.3. Oczekiwanie warunkowe

Rozważ prostą klasę `Kolejka` z metodami do dodawania i usuwania obiektów. Synchronizowanie metod zapewnia atomowość tych operacji.

```
public class Kolejka {
    class Węzeł { Object wartość; Węzeł następny; };
    private Węzeł głowa;
    private Węzeł ogon;

    public synchronized void dodaj(Object nowaWartość) {
        Węzeł n = new Węzeł();
        if (głowa == null) głowa = n;
        else ogon.następny = n;
        ogon = n;
        ogon.wartość = nowaWartość;
    }

    public synchronized Object usuń() {
        if (głowa == null) return null;
        Node n = głowa;
        głowa = n.następny;
        return n.wartość;
    }
}
```

Załóżmy teraz, że chcemy zmodyfikować metodę `usuń` w taki sposób, by blokowała, jeśli kolejka jest pusta.

Sprawdzenie, czy kolejka nie jest pusta, musi pojawić się wewnątrz metody synchronizowanej, w przeciwnym razie sprawdzenie takie byłoby niewiarygodne — inny wątek mógłby opróżnić kolejkę w międzyczasie.

```
public synchronized Object pobierz() {
    if (głowa == null) ... // I co?
    Node n = głowa;
    głowa = n.następny;
    return n.wartość;
}
```

Co powinno się wydarzyć, jeśli kolejka jest pusta? Żaden inny wątek nie może dodać elementów, dopóki bieżący wątek trzyma blokadę. Tutaj właśnie pojawia się metoda `wait`.

Jeśli metoda `pobierz` ustali, że nie może dalej działać, wywołuje metodę `wait`:

```
public synchronized Object pobierz() throws InterruptedException {
while (głowa == null) wait();
...
}
```

Bieżący wątek jest teraz dezaktywowany i zwalnia blokadę. To dopuszcza inny wątek, który może, miejmy nadzieję, dodać elementy do kolejki. Jest to nazywane **oczekiwaniem warunkowym** (ang. *waiting on a condition*).

Zauważ, że metoda `wait` jest metodą klasy `Object`. Korzysta ona z blokady powiązanej z obiektem.

Istnieje podstawowa różnica między wątkiem oczekującym na blokadę a wątkiem, który wywołał `wait`. Gdy wątek wywołuje metodę `wait`, wprowadza do obiektu **zestaw `wait`**. Wątek nie jest uruchamiany, gdy blokada staje się dostępna. Zamiast tego pozostaje nieaktywny, dopóki inny wątek nie wywoła metody `notifyAll` na tym samym obiekcie.

Gdy inny wątek doda element, powinien wywołać tę metodę:

```
public synchronized void add(Object newValue) {
    ...
    notifyAll();
}
```

Wywołanie `notifyAll` reaktywuje wszystkie wątki zapisane w zestawie `wait`. Gdy wątki zostają usunięte z zestawu `wait`, są ponownie gotowe do uruchomienia i mechanizm zarządzający powinien ponownie je aktywować. Wtedy powinny one ponownie spróbować pobrać blokadę. Gdy jednemu z nich się powiedzie, kontynuuje działanie, powracając z wywołania `wait`.

Wtedy wątek powinien ponownie sprawdzić warunek. Nie ma gwarancji, że teraz warunek jest spełniony — metoda notifyAll jedynie sygnalizuje oczekującemu wątkowi, że pojawiła się taka *możliwość* i że warto ponownie sprawdzić warunek. Z tego powodu test jest umieszczony w pętli

```
while (głowa == null) wait();
```

Wątek może wywołać `wait`, `notifyAll` lub `notify` na obiekcie jedynie pod warunkiem, że ma blokadę na tym obiekcie.

Inna metoda, `notify`, odblokowuje jedynie pojedynczy wątek z zestawu `wait`. Jest to bardziej wydajne niż odblokowywanie wszystkich wątków, ale istnieje też niebezpieczeństwo. Jeśli wybrany wątek ustali, że nie może kontynuować działania, jest ponownie blokowany. Jeśli żaden inny wątek nie wywoła na nim ponownie `notify`, w programie pojawia się zakleszczenie.

Przy implementowaniu struktur danych z metodami blokującymi można skorzystać z metod `wait` i `notifyAll`. Nie jest jednak łatwo odpowiednio je wykorzystać. Programiści aplikacji nigdy nie powinni mieć potrzeby sięgania po te metody. Zamiast tego mogą użyć przygotowanych struktur danych, takich jak `LinkedBlockingQueue` lub `ConcurrentHashMap`.

10.8. Wątki

Zbliżając się do końca tego rozdziału, dotarliśmy wreszcie do momentu, kiedy powinniśmy porozmawiać na temat wątków, podstawowych struktur, które w rzeczywistości wykonują zadania. Normalnie lepiej jest korzystać z wykonawców, którzy zarządzają wątkami za Ciebie, ale kolejne punkty dadzą Ci trochę podstawowych informacji na temat pracy bezpośrednio z wątkami.

10.8.1. Uruchamianie wątku

Wątek w języku Java uruchamia się tak:

```
Runnable zadanie = () -> { ... };
Thread wątek = new Thread(zadanie);
wątek.start();
```

Statyczna metoda `sleep` sprawia, że bieżący wątek zasypia na określony czas, więc inne wątki mają możliwość działać.

```
Runnable task = () -> {
    ...
    Thread.sleep(millis);
    ...
}
```

Jeśli chcesz zaczekać, aż wątek zakończy działanie, wywołaj metodę `join`:

```
thread.join(millis);
```

Te dwie metody wyrzucają kontrolowany wyjątek `InterruptedException` omówiony w kolejnym punkcie.

Wątek kończy działanie, gdy metoda `run` zwraca wartość — albo w standardowy sposób, albo z powodu wyrzucenia wyjątku. W tym drugim przypadku wywoływany jest mechanizm odpowiedzialny za obsługę **nieprzechwyconych wyjątków** wątku. Gdy wątek jest tworzony, obsługa takich wyjątków jest przekierowywana do mechanizmu obsługi dla grupy wątków, który z kolei jest przekierowaniem do globalnego mechanizmu obsługi (patrz rozdział 5.). Możesz zmienić mechanizm obsługi dla wątku, wywołując metodę `setUncaughtExceptionHandler`.

Początkowe wersje języka Java definiowały metodę `stop`, która natychmiast zatrzymywała działanie wątku, oraz metodę `suspend`, która blokowała wątek do czasu, aż inny wątek wywołał `resume`. Obie metody są już przestarzałe.

Metoda `stop` jest niebezpieczna z zasady. Przypuśćmy, że wątek zostanie zatrzymany podczas wykonywania sekcji krytycznej — na przykład wstawiania elementu do kolejki. Wtedy kolejka pozostaje częściowo zaktualizowana. Jednak blokada chroniąca sekcję krytyczną jest zwolniona i inne wątki mogą korzystać z uszkodzonej struktury danych. Powinieneś zgłosić przerwanie wątku, który chcesz zatrzymać. Wątek, który otrzymał takie żądanie, może wtedy zakończyć działanie w chwili, gdy będzie to bezpieczne.

Metoda `suspend` nie jest tak ryzykowna, ale nadal jest problematyczna. Jeśli wątek zostanie zawieszony podczas trzymania blokady, każdy inny wątek próbujący uzyskać tę blokadę zostanie zablokowany. Jeśli wątek przywracający działanie takiego wątku należy do tej grupy, w programie pojawia się zakleszczenie.

10.8.2. Przerywanie wątków

Przypuśćmy, że dla danego zapytania zawsze satysfakcjonujący jest pierwszy uzyskany wynik. Jeśli poszukiwanie odpowiedzi jest podzielone na wiele zadań, będziesz chciał anulować wszystkie inne, gdy tylko pojawi się pierwszy wynik. W języku Java anulowanie zadań jest przeprowadzane we **współpracy**.

Każdy wątek ma **status przerwany**, który sygnalizuje, że ktoś chce przerwać działanie wątku. Nie ma dokładnej definicji, co oznacza „przerwanie”, ale większość programistów wykorzystuje to do sygnalizowania żądania anulowania.

Obiekt `Runnable` może sprawdzić ten status, co zazwyczaj jest wykonywane w pętli:

```
Runnable task = () -> {
    while (praca do wykonania) {
        if (Thread.currentThread().isInterrupted()) return;
        Inne operacje
    }
};
```

Gdy wątek jest przerywany, metoda `run` po prostu kończy działanie.

Istnieje też statyczna metoda `Thread.interrupted`, która pobiera status informujący o żądaniu przerwania dla bieżącego wątku, usuwa jego zawartość i zwraca starą wartość.

Czasem wątek staje się tymczasowo nieaktywny. Może się tak zdarzyć, jeśli wątek czeka na wartość, która ma być obliczona przez inny wątek, lub na możliwość wymiany danych albo jest usypiany, by dać innym wątkom możliwość działania.

Jeśli wątek jest przerywany w czasie oczekiwania lub uśpienia, jest natychmiast reaktywowany — ale w takim przypadku status nie zostaje ustawiony. Zamiast tego wyrzucany jest wyjątek `InterruptedException`. Jest to wyjątek kontrolowany i musisz przechwycić go wewnątrz metody `run` obiektu `Runnable`. Typowym działaniem w przypadku takiego wyjątku jest zakończenie działania metody `run`:

```
Runnable task = () -> {
    try {
        while (praca do wykonania) {
            Polecenia do wykonania
            Thread.sleep(millis);
        }
    }
    catch (InterruptedException ex) {
        // Nic
    }
};
```

Jeśli przechwytujesz `InterruptedException` w ten sposób, nie ma konieczności sprawdzania statusu `interrupted`. Jeśli wątek został przerwany poza wywołaniem `Thread.sleep`, status jest ustawiany i metoda `Thread.sleep` wyrzuca wyjątek `InterruptedException`, gdy tylko zostanie wywołana.

Wyjątek `InterruptedException` może wydawać się nieznośny, ale nie powinieneś go jedynie przechwytywać i ukrywać przy wywołaniu metody takiej jak `sleep`. Jeśli nie możesz zrobić nic innego, przynajmniej ustaw odpowiedni status:

```
try {
Thread.sleep(millis);
} catch (InterruptedException ex) {
Thread.currentThread().interrupt();
}
```

Lub lepiej po prostu przekaż wyjątek do odpowiedniego mechanizmu obsługi:

```
public void mySubTask() throws InterruptedException {
...
Thread.sleep(millis);
...
}
```

10.8.3. Zmienne lokalne w wątku

Czasem możesz uniknąć współdzielenia, wykorzystując klasę pomocniczą `ThreadLocal` i dając każdemu wątkowi oddzielną instancję. Na przykład klasa `NumberFormat` nie jest bezpieczna dla wątków. Przypuśćmy, że mamy zmienną statyczną

```
public static final NumberFormat formatWaluty = NumberFormat.getCurrencyInstance();
```

Jeśli dwa wątki wykonują operacje takie jak

```
String należnaKwota = currencyFormat.format(suma);
```

to wynik może być nieprawidłowy, ponieważ wewnętrzne struktury danych wykorzystywane przez instancję `NumberFormat` mogą być zniszczone przez równoczesny dostęp. Mógłbyś wykorzystać blokadę lub udostępnić synchronizowaną metodę, aby zapewnić atomowy dostęp do współdzielonej zmiennej `NumberFormat`. Alternatywnie mógłbyś utworzyć lokalny obiekt `NumberFormat`, jeśli byłby potrzebny, ale to również marnotrawstwo.

Aby utworzyć instancję dla każdego wątku, użyj poniższego kodu:

```
public static final ThreadLocal<NumberFormat> formatWaluty =
    ThreadLocal.withInitial(() -> NumberFormat.getCurrencyInstance());
```

Aby uzyskać dostęp do właściwego formatowania, wywołaj

```
String należnaKwota = currencyFormat.get().format(suma);
```

Gdy po raz pierwszy wywołujesz `get` w danym wątku, wyrażenie lambda z konstruktora jest wywoływane, by utworzyć dla tego wątku instancję. Od tej chwili metoda `get` zwraca instancję należącą do bieżącego wątku.

10.8.4. Dodatkowe właściwości wątku

Klasa `Thread` udostępnia wiele właściwości dla wątków, ale większość z nich przydaje się bardziej studentom przy zdawaniu egzaminów certyfikacyjnych niż programistom aplikacji. W tym punkcie przejrzymy je pobieżnie.

Wątki mogą być łączone w grupy i istnieją metody API służące do zarządzania grupami wątków, takie jak przerwanie działania wszystkich wątków w grupie. Obecnie preferowanym mechanizmem do zarządzania grupami zadań są wykonawcy.

Możesz ustawić **priorytety** dla wątków i wtedy wątki o wysokim priorytecie są kierowane do uruchomienia przed tymi z niskim priorytetem. Zakładamy tu, że priorytety są uwzględniane przez wirtualną maszynę i platformę hosta, ale szczegóły są w dużym stopniu zależne od platformy. Dlatego korzystanie z priorytetów to niepewna sprawa i nie jest zalecane.

Wątki mają **stany** i możesz określić, czy wątek jest nowy, uruchomiony, zablokowany przez wejście lub wyjście danych, oczekujący czy zakończony. Korzystając z wątków jako programista aplikacji, rzadko masz powód, by ustalać ich stany.

Wątki mają nazwy — możesz zmienić nazwę, by ułatwić debugowanie. Na przykład:

```
Thread.currentThread().setName("Bitcoin-miner-1");
```

Gdy wątek kończy działanie z powodu nieprzechwyconego wyjątku, wyjątek jest przekazywany do ustalonego dla wątku **mechanizmu obsługującego nieprzechwycone wyjątki** (ang. *uncaught exception handler*). Domyślnie ślad jego stosu jest zrzucany do `System.err`, ale możesz też zainstalować własny mechanizm obsługi (patrz rozdział 5.).

Demon (ang. *daemon*) to wątek, który nie ma innego zadania niż obsługa innych wątków. Jest to przydatne w przypadku wątków wysyłających sygnał zegarowy lub czyszczących stare zapisy pamięci podręcznej. Gdy pozostają jedynie wątki demonów, wirtualna maszyna kończy działanie.

Aby utworzyć wątek demona, przed uruchomieniem wątku wywołaj `thread.setDaemon(true)`.

10.9. Procesy

Jak dotąd widziałeś, w jaki sposób wykonywać kod Java w oddzielnych wątkach tego samego programu. Czasem musisz uruchomić inny program. Aby to wykonać, użyj klasy `ProcessBuilder` i `Process`. Klasa `Process` wykonuje polecenie w oddzielnym procesie systemu operacyjnego i pozwala na interakcję ze strumieniami standardowego wejścia, wyjścia i błędów. Klasa `ProcessBuilder` pozwala konfigurować obiekt `Process`.

Klasa `ProcessBuilder` jest bardziej elastycznym zamiennikiem wywołań `Runtime.exec`.

10.9.1. Tworzenie procesu

Rozpocznij tworzenie procesu od określenia polecenia, które zechcesz wykonać. Możesz dostarczyć `List<String>` lub po prostu ciągi znaków tworzące polecenie.

```
ProcessBuilder builder = new ProcessBuilder("gcc", "myapp.c");
```

Pierwszy ciąg znaków musi być poleceniem do wykonania, a nie poleceniem wbudowanym powłoki. Na przykład: by uruchomić polecenie `dir` w Windows, musisz utworzyć proces, korzystając z ciągów znaków: `"cmd.exe"`, `"/C"` i `"dir"`.

Każdy proces ma **katalog roboczy**, który jest wykorzystywany do ustalania względnych nazw katalogów. Domyślnie proces otrzymuje ten sam katalog roboczy, który ma wirtualna maszyna, i jest to zazwyczaj ten sam katalog, z którego uruchomiłeś program Java. Możesz to zmienić, korzystając z metody `directory`:

```
builder = builder.directory(ścieżka.toFile());
```

Każda z metod służących do konfiguracji obiektu `ProcessBuilder` zwraca ten obiekt, dzięki czemu możesz polecenia połączyć w łańcuch. Ostatecznie możesz wywołać

```
Process p = new ProcessBuilder(command).directory(file)....start();
```

Następnie będziesz chciał określić, co stanie się ze standardowymi strumieniami wejścia, wyjścia i błędu procesu. Domyślnie każdy z nich jest potokiem, do którego możesz się odwołać za pomocą

```
OutputStream processIn = p.getOutputStream();
InputStream processOut = p.getInputStream();
InputStream processErr = p.getErrorStream();
```

Zauważ, że strumień wejściowy procesu jest strumieniem wyjściowym maszyny wirtualnej! Zapisujesz do tego strumienia i to, co tam zapiszesz, pojawia się na wejściu procesu. Tak samo w drugą stronę: odczytujesz to, co proces zapisuje do strumieni wyjściowego i błędów. Dla Ciebie są to strumienie wejściowe.

Możesz określić, że strumienie wejściowy, wyjściowy i błędu nowego procesu będą takie same jak maszyny wirtualnej. Jeśli użytkownik uruchomi maszynę wirtualną w konsoli, wszystkie dane wprowadzane przez użytkownika są przekazywane do procesu i dane generowane przez proces pojawiają się na konsoli. Wywołaj

```
builder.inheritIO()
```

by ustawić w ten sposób wszystkie trzy strumienie. Jeśli chcesz wykorzystać jedynie niektóre strumienie, przekaż wartość

```
ProcessBuilder.Redirect.INHERIT
```

do metody `redirectInput`, `redirectOutput` lub `redirectError`. Na przykład

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

Możesz przekierować strumienie procesu do plików, przekazując obiekty `File`:

```
builder.redirectInput(inputFile)
    .redirectOutput(outputFile)
    .redirectError(errorFile)
```

Pliki dla strumienia wyjściowego i błędów są tworzone lub czyszczone przy uruchomieniu procesu. Aby dopisać do istniejących plików, użyj

```
builder.redirectOutput(ProcessBuilder.Redirect.appendTo(plikWyjściowy));
```

Często wygodnie jest połączyć strumienie wyjściowy i błędów tak, by widzieć wyniki działania i błędy w takiej kolejności, w jakiej proces je generuje. Wywołaj

```
builder.redirectErrorStream(true)
```

aby uruchomić łączenie. Jeśli to zrobisz, nie będziesz mógł wywołać metody `redirectError` na obiekcie `ProcessBuilder` ani `getErrorStream` na obiekcie `Process`.

W końcu możesz chcieć modyfikować zmienne środowiska procesu. Tego niestety nie można włączyć do łańcucha wywołań. Musisz pobrać środowisko z obiektu `ProcessBuilder` (inicjalizowane przez zmienne środowiska procesu uruchamiającego maszynę wirtualną), a następnie dodać lub usunąć elementy.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

10.9.2. Uruchamianie procesu

Po skonfigurowaniu obiektu `ProcessBuilder` wywołaj metodę `start`, by uruchomić proces. Jeśli skonfigurowałeś strumienie wejściowy, wyjściowy i błędów jako potoki, możesz w tym momencie zapisywać do strumienia wejściowego i odczytywać strumienie wyjściowy i błędów. Na przykład

```
Process process = new ProcessBuilder("/bin/ls", "-l")
    .directory(Paths.get("/tmp").toFile())
    .start();
```

```
try (Scanner in = new Scanner(process.getInputStream())) {
    while (in.hasNextLine())
        System.out.println(in.nextLine());
}
```

Przestrzeń bufora strumieni procesu jest ograniczona. Nie powinieneś przepełniać strumienia wejściowego i musisz odczytywać dane wyjściowe często. Jeśli wymiana danych jest intensywna, możesz potrzebować oddzielnych wątków do wysyłania i pobierania danych.

Aby czekać na zakończenie procesu, wywołaj

```
int wynik = process.waitFor();
```

lub, jeśli nie chcesz czekać w nieskończoność:

```
long opóźnienie = ...;
if (process.waitfor(opóźnienie, TimeUnit.SECONDS)) {
    int result = process.exitValue();
    ...
} else {
    process.destroyForcibly();
}
```

Pierwsze wywołanie `waitFor` zwraca wartość zwróconą przez proces (standardowo `0` przy poprawnym zakończeniu lub różny od zera kod błędu). Drugie wywołanie zwraca `true`, jeśli proces nie przekroczył czasu wykonania. Wtedy musisz pobierać zwróconą wartość, wywołując metodę `exitValue`.

Zamiast czekać na zakończenie procesu, możesz po prostu go uruchomić i czasem wywoływać `isAlive`, by sprawdzić, czy nadal działa. Aby zakończyć proces, wywołaj `destroy` lub `destroyForcibly`. Różnica między tymi wywołaniami zależy od platformy. W systemie Unix pierwsze z nich kończy proces sygnałem `SIGTERM`, drugie — `SIGKILL`. (Metoda `supportsNormalTermination` zwraca true, jeśli metoda destroy może normalnie zakończyć działanie procesu).

I wreszcie — możesz otrzymać asynchroniczne powiadomienie po zakończeniu procesu. Wywołanie `process.onExit()` zwraca obiekt typu `CompletableFuture<Process>`, którego możesz użyć do zaplanowania dowolnej operacji.

```
process.onExit().thenAccept(
   p -> System.out.println("Exit value: " + p.exitValue()));
```

10.9.3. Uchwyty procesów

Aby uzyskać więcej informacji na temat procesu, który uruchomił Twój program, lub dowolnego innego procesu działającego na Twojej maszynie, użyj interfejsu `ProcessHandle`. Możesz uzyskać `ProcessHandle` na cztery sposoby:

1. Jeśli masz obiekt `p` typu `Process`, `p.toHandle()` zwraca jego `ProcessHandle`.
2. Jeśli masz ID procesu w systemie operacyjnym w zmiennej typu `long`, `ProcessHandle.of(id)` zwraca uchwyt do tego procesu.

3. `ProcessHandle.current()` zwraca uchwyt do procesu, w którym uruchomiona jest ta maszyna wirtualna Java.
4. `ProcessHandle.allProcesses()` zwraca `Stream<ProcessHandle>` ze wszystkimi procesami systemu operacyjnego, które są widoczne dla bieżącego procesu.

Mając uchwyt procesu, możesz uzyskać jego ID, proces nadrzędny, procesy podrzędne oraz wszystkie procesy potomne:

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
```

Zwracane przez metody `allProcesses`, `children` i `descendants` instancje `Stream <Process ↪Handle>` prezentują stan w danej chwili. Każdy proces w takim strumieniu może zostać zakończony do czasu, gdy zechcesz się nim zająć; mogą też zostać uruchomione inne procesy, które nie znajdują się w strumieniu.

Metoda `info` zwraca obiekt `ProcessHandle.Info` z metodami pozwalającymi uzyskać informacje na temat procesu.

```
Optional<String[]> arguments()
Optional<String> command()
Optional<String> commandLine()
Optional<String> startInstant()
Optional<String> totalCpuDuration()
Optional<String> user()
```

Wszystkie te metody zwracają wartości `Optional`, ponieważ istnieje możliwość, że dany system operacyjny może nie być w stanie dostarczyć takiej informacji.

Do monitorowania lub wymuszania zakończenia procesu interfejs `ProcessHandle` ma te same metody: `isAlive`, `supportsNormalTermination`, `destroy`, `destroyForcibly` i `onExit`, co klasa `Process`. Nie ma jednak odpowiednika metody `waitFor`.

Ćwiczenia

1. Korzystając z równoległych strumieni, znajdź wszystkie pliki z katalogu zawierające podane słowo. Jak wyszukać tylko pierwsze słowo? Czy pliki są faktycznie przeszukiwane równolegle?
2. Jak duża musi być tablica, by `Arrays.parallelSort` działało szybciej niż `Arrays.sort` na Twoim komputerze?
3. Zaimplementuj metodę zwracającą zadanie, które wczytuje wszystkie słowa z pliku, próbując odnaleźć wskazane słowo. Zadanie powinno zakończyć działanie natychmiast (z komunikatem do debugowania) po jego przerwaniu. Przygotuj do wykonania oddzielne zadanie dla każdego pliku w katalogu. Przerwij wszystkie inne, jeśli jedno zakończy się sukcesem.

4. Jedna z równoległych operacji, nieomówionych w punkcie 10.4.2, „Równoległe operacje na tablicach”, to metoda `parallelPrefix` zamieniająca każdy element tablicy połączeniem wszystkich wcześniejszych elementów rozdzielanych przekazanym operatorem działania łącznego. Że co? Oto przykład. Weźmy tablicę `[1, 2, 3, 4, ...]` i operator *. Po wykonaniu `Arrays.parallelPrefix(values, (x, y) -> x * y)` tablica zawiera

   ```
   [1, 1 * 2, 1 * 2 * 3, 1 * 2 * 3 * 4, ...]
   ```

   Prawdopodobnie to zaskakujące, że ta operacja może być zrównoleglona. Najpierw połącz sąsiadujące elementy w taki sposób jak poniżej:

   ```
   [1, 1 * 2, 3, 3 * 4, 5, 5 * 6, 7, 7 * 8]
   ```

   Pogrubione zostały zmienione wartości. Jak widać, można wykonać to równocześnie w oddzielnych fragmentach tablicy. W kolejnym kroku zaktualizuj wyróżnione elementy, mnożąc je przez elementy, które znajdują się jedno lub dwa miejsca wcześniej:

   ```
   [1, 1 * 2, 1 * 2 * 3, 1 * 2 * 3 * 4, 5, 5 * 6, 5 * 6 * 7, 5 * 6 * 7 * 8]
   ```

   To również można wykonać równolegle. Po log(n) krokach proces jest zakończony. Ma to przewagę nad prostym linearnym przetwarzaniem, jeśli dostępna jest wystarczająca liczba procesorów.

   W tym ćwiczeniu powinieneś wykorzystać metodę `parallelPrefix` do zrównoleglenia obliczeń ciągu Fibonacciego. Wykorzystamy fakt, że n-ta liczba Fibonacciego jest górnym lewym współczynnikiem F^n, gdzie $F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.

   Utwórz tablicę wypełnioną macierzami 2×2. Zdefiniuj klasę `Macierz` z metodą do mnożenia, wykorzystaj `parallelSetAll`, by utworzyć tablicę macierzy, i wykorzystaj `parallelPrefix`, aby je pomnożyć.

5. Utwórz przykład pokazujący ucieczkę `this` przez konstruktor niemodyfikowalnej klasy (patrz 10.3.3., „Strategie bezpiecznego korzystania ze współbieżności”). Spróbuj pokazać coś przekonującego i niepokojącego. Jeśli skorzystasz z nasłuchiwania zdarzeń (jak wiele przykładów w sieci), powinien on nasłuchiwać czegoś interesującego, co nie jest proste w przypadku klasy niemodyfikowalnej.

6. Napisz aplikację, w której wiele wątków wczytuje wszystkie słowa z kolekcji plików. Użyj `ConcurrentHashMap<String, Set<File>>`, by śledzić, w których plikach każde ze słów się pojawia. Użyj metody `merge`, by zaktualizować mapę.

7. Powtórz poprzednie ćwiczenie, ale użyj `computeIfAbsent`. Jaka jest zaleta takiego podejścia?

8. W `ConcurrentHashMap<String, Long>` znajdź klucz z maksymalną wartością (dowolnie traktując równe wartości). Podpowiedź: `reduceEntries`.

9. Wygeneruj 1000 wątków, z których każdy zwiększa licznik 100 000 razy. Porównaj wydajność przy korzystaniu z `AtomicLong` i `LongAdder`.

10. Użyj `LongAccumulator`, by odnaleźć największy lub najmniejszy z zebranych elementów.

11. Użyj kolejki blokującej do przetwarzania plików w katalogu. Jeden wątek przechodzi przez drzewo plików i wstawia pliki do kolejki. Kilka wątków usuwa pliki i przeszukuje je pod kątem występowania wskazanego słowa kluczowego, wyświetlając dopasowane wartości. Gdy producent zakończy działanie, powinien umieścić w kolejce imitację pliku.

12. Powtórz poprzednie ćwiczenie, ale niech każdy wątek przeszukujący tworzy mapę słów i częstotliwości ich występowania, a następnie umieszcza ją w innej kolejce. Końcowy wątek łączy słowniki i wyświetla najczęściej pojawiające się słowa. Dlaczego nie musisz korzystać z `ConcurrentHashMap`?

13. Powtórz poprzednie ćwiczenie, tworząc `Callable<Map<String, Integer>>` dla każdego pliku i korzystając z odpowiedniej usługi wykonującej. Połącz wyniki, gdy wszystkie będą już dostępne. Dlaczego nie musisz korzystać z `ConcurrentHashMap`?

14. Użyj `ExecutorCompletionService` i połącz wyniki, gdy tylko będą dostępne.

15. Powtórz poprzednie ćwiczenie, korzystając z `ConcurrentHashMap` do zbierania częstotliwości występowania słów.

16. Powtórz poprzednie ćwiczenie, korzystając ze strumieni równoległych. Żadna z operacji na strumieniach nie powinna powodować efektów ubocznych.

17. Napisz program przechodzący przez drzewo katalogów i tworzący wątek dla każdego pliku. W wątkach policz liczbę słów w pliku i, bez korzystania z blokad, zaktualizuj współdzielony licznik zadeklarowany jako

```
public static long count = 0;
```

    Uruchom program wiele razy. Co się dzieje? Dlaczego?

18. Popraw program z poprzedniego ćwiczenia, korzystając z blokady.

19. Popraw program z poprzedniego ćwiczenia, korzystając z `LongAdder`.

20. Rozważ taką implementację stosu:

```
public class Stos {
    class Węzeł { Object wartość; Node następny; };
    private Węzeł wierzchołek;

    public void umieść(Object nowaWartość) {
        Węzeł n = new Węzeł();
        n.wartość = nowaWartość;
        n.następny = wierzchołek;
        wierzchołek = n;
    }

    public Object pobierz() {
        if (wierzchołek == null) return null;
        Node n = wierzchołek;
        wierzchołek = n.następny;
        return n.wartość;
    }
}
```

    Opisz dwa różne powody tego, że struktura danych może zawierać niepoprawne elementy.

21. Rozważ implementację kolejki:

```
public class Kolejka {
    class Węzeł { Object wartość; Węzeł następny; };
    private Węzeł głowa;
    private Węzeł ogon;

    public void dodaj(Object nowaWartość) {
        Węzeł n = new Węzeł();
        if (głowa == null) głowa = n;
        else ogon.następny = n;
        ogon = n;
        ogon.wartość = nowaWartość;
    }

    public Object usuń() {
        if (głowa == null) return null;
        Węzeł n = głowa;
        głowa = n.następny;
        return n.wartość;
    }
}
```

Opisz dwa różne problemy, które mogą spowodować, że struktura danych będzie zawierała niepoprawne elementy.

22. Co jest niepoprawne w poniższym fragmencie kodu?

```
public class Stos {
    private Object mojaBlokada = "LOCK";

    public void umieść(Object nowaWartość) {
        synchronized (mojaBlokada) {
            ...
        }
    }
    ...
}
```

23. Co jest niepoprawne w poniższym fragmencie kodu?

```
public class Stos {
    public void umieść(Object nowaWartość) {
        synchronized (new ReentrantLock()) {
            ...
        }
    }
    ...
}
```

24. Co jest niepoprawne w poniższym fragmencie kodu?

```
public class Stos {
    private Object[] wartość = new Object[10];
    private int rozmiar;

    public void umieść(Object nowaWartość) {
        synchronized (wartości) {
            if (rozmiar == wartości.length)
                wartości = Arrays.copyOf(wartości, 2 * rozmiar);
```

```
                wartości[rozmiar] = nowaWartość;
                rozmiar++;
            }
        }
        ...
    }
```

25. Napisz program pytający użytkownika o URL, wczytujący stronę internetową dostępną pod wskazanym adresem i wyświetlający wszystkie odnośniki. Wykorzystaj `CompletableFuture` w każdym kroku. Nie wywołuj `get`.

26. Napisz metodę

```
public static <T> CompletableFuture<T> powtarzaj(
    Supplier<T> działanie, Predicate<T> dopóki)
```

asynchronicznie powtarzającą działanie, dopóki nie wytworzy ono wartości, która zostanie zaakceptowana przez funkcję `dopóki`, również działającą asynchronicznie. Sprawdź funkcję wczytującą `java.net.PasswordAuthentication` z konsoli i funkcję, która symuluje sprawdzenie poprawności poprzez wstrzymanie działania na sekundę, a następnie sprawdzenie, czy hasło to `"secret"`. Podpowiedź: użyj rekurencji.

27. Zaimplementuj statyczną metodę `CompletableFuture<T> <T> supplyAsync ↪(Supplier<T> action, Executor exec)`, zwracającą instancję podklasy `CompletableFuture<T>`, której metoda `cancel` może przerwać wątek wykonujący metodę `action`, jeśli tylko zadanie jest wykonywane. W `Runnable` przechwyć bieżący wątek, a następnie wywołaj `action.get()` i zakończ `CompletableFuture`, zwracając wynik lub wyrzucając wyjątek.

28. Metoda

```
static CompletableFuture<Void> CompletableFuture.allOf
↪(CompletableFuture<?>... cfs)
```

nie zwraca wyniku argumentów, co sprawia, że jest kłopotliwa w użyciu. Zaimplementuj metodę, która łączy obiekty CompletableFuture tego samego typu:

```
static <T> CompletableFuture<List<T>> allOf(List<CompletableFuture<T>> cfs)
```

Zauważ, że metoda ta przyjmuje parametr List, ponieważ nie możesz mieć zmiennych argumentów typów prostych.

29. Metoda

```
static CompletableFuture<Object>
    CompletableFuture.anyOf(CompletableFuture<?>... cfs)
```

zwraca wartość, gdy tylko jeden z argumentów zakończy działanie, normalnie lub wyrzucając wyjątek. Jest to znaczna różnica w porównaniu z `ExecutorService. ↪invokeAny`, który kontynuuje działanie, dopóki jedno z zadań nie zakończy się sukcesem, i uniemożliwia wykorzystanie metody do równoczesnego wyszukiwania. Zaimplementuj metodę

```
static CompletableFuture<T> anyOf(List<Supplier<T>> actions, Executor exec)
```

która zwraca pierwszy prawdziwy wynik lub `NoSuchElementException`, jeśli wszystkie działania zakończą się wyjątkami.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Z

Ż

Dziś język Java — starannie zaprojektowany i wciąż rozwijany — stanowi standard w wielu potężnych korporacjach z branży IT. W język ten wbudowano funkcje ułatwiające implementację wielu złożonych zadań programistycznych. W jego nowej wersji znalazło się wiele usprawnień dotyczących najbardziej podstawowych technologii platformy Java. Nowe mechanizmy, na przykład modularyzacja czy nowatorskie podejście do programowania współbieżnego, podniosą efektywność pracy programisty. Jednak opanowanie tak potężnego narzędzia i używanie go na profesjonalnym poziomie stało się prawdziwym wyzwaniem.

Ta książka jest kompletnym i zwięzłym kompendium praktycznego wykorzystania Javy. Została pomyślana w taki sposób, aby nauka języka i bibliotek odbywała się możliwie szybko. Omówiono tu bardzo dużo materiału, ale jego uporządkowanie i sposób prezentacji ułatwiają szybki dostęp do danego zagadnienia i łatwe zrozumienie treści. Dzięki temu płynnie nauczysz się wszystkich nowości, od systemu modułów Project Jigsaw po wyrażenia lambda czy strumienie. Opanujesz tajniki programowania współbieżnego dzięki potężnym mechanizmom dostępnym w bibliotekach. Docenisz tę książkę, jeśli profesjonalnie piszesz aplikacje w Javie, zwłaszcza jeżeli chcesz tworzyć oprogramowanie działające po stronie serwera lub w systemie Android.

Najważniejsze zagadnienia:

- **Modularyzacja, w tym stosowanie modułów zewnętrznych**
- **Testowanie kodu za pomocą JShell REPL**
- **Wyrażenia lambda i praca z kolekcjami**
- **Korzystanie ze Streams API**
- **Operacje wejścia-wyjścia, wyrażenia regularne oraz procesy**
- **Współbieżność i zadania współpracujące ze sobą**

Cay S. Horstmann jest autorem wielu bestsellerowych książek na temat programowania w Javie, przeznaczonych zarówno dla profesjonalistów, jak i dla studentów informatyki. Jest profesorem informatyki na Uniwersytecie Stanowym San Jose. Zdobył tytuł Java Champion. Bardzo często wygłasza referaty na różnych konferencjach informatycznych.

Mistrz Javy — to tak dumnie brzmi!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-4250-7

Cena: 79,00 zł