Wprowadzenie do programowania współbieżnego

autor wykładu: Bartłomiej Starosta

Spis treści

1 Podstawowe Pojęcia **1**
1.1 Wstęp 1
1.2 Zastosowania wątków 2

2 Problemy związane ze współbieżnością **3**
2.1 Współdzielenie zasobów 4
2.2 Wzajemne wykluczanie 6
2.3 Sekcja krytyczna 7
2.4 Przykłady 7
2.4.1 Pięciu Filozofów 7
2.4.2 Współdzielenie zmiennych 9

3 Wątki w Javie **11**
3.1 Implementacja 13
3.2 API - wstęp 14
3.3 Tworzenie wątków 15
3.3.1 Dziedziczenie klasy `Thread` 15
3.3.2 Implementowanie interfejsu `Runnable` 16
3.3.3 Klasa anonimowa 17

3.4 Zatrzymywanie wątków . . . 18
3.5 Synchronizowanie . . . 19
3.5.1 Synchronizowanie bloków . . . 20
3.5.2 Synchronizowanie metod . . . 21
3.5.3 Przykład braku synchronizacji . . . 22
3.5.4 Przykład złej synchronizacji . . . 23
3.6 Stany wątków . . . 25
3.7 Sterowanie wykonaniem . . . 27
3.7.1 Zawieszanie i wznawianie . . . 28
3.7.2 Usypianie . . . 34
3.7.3 Przekazywanie sterowania . . . 34
3.7.4 Blokujące wejście-wyjście . . . 34
3.8 Priorytety . . . 35
3.9 Rodzaje wątków . . . 36
3.9.1 Zwykłe (nie-demony) . . . 36
3.9.2 Demony . . . 37
3.10 Komunikacja . . . 38
3.11 Grupy wątków . . . 41

4 API - szczegóły **41**
4.1 Interfejs `Runnable` . . . 42

4.2 Klasa `Object` . . . 42
4.3 Klasa `Thread` . . . 42
4.3.1 Najważniejsze konstruktory . . . 42
4.3.2 Metody sterujące . . . 43
4.3.3 Metody pomocnicze . . . 43
4.3.4 Metody zaniechane . . . 44
4.4 Wyjątki . . . 44

5 AWT i Swing **45**
5.1 Kolejkowanie zadań . . . 45
5.1.1 Kolejkowanie asynchroniczne . . . 46
5.1.2 Kolejkowanie synchroniczne . . . 46
5.2 Generowanie zdarzeń . . . 47

6 Podsumowanie **48**
6.1 Podstawowa zasada programowania współbieżnego . . . 49
6.2 Dobre rady . . . 50
6.2.1 Nakazy . . . 50
6.2.2 Zasady dobrego stylu . . . 50

1 Podstawowe Pojęcia

1.1 Wstęp

Współczesne systemy operacyjne domowego użytku przyzwyczaiły nas do posługiwania się kilkoma programami jednocześnie. Każdy z uruchomionych programów wydaje się mieć procesor i inne zasoby sprzętowe (pamięć, dysk) do swojej własnej dyspozycji. W rzeczywistości każdy z nich jest uruchamiany na krótki odcinek czasu (*kwant*), jednak na tyle często, że użytkownik nie dostrzega opóźnień z tym związanych nawet w programach interaktywnych. Podobnie rzecz się ma w maszynach wieloprocesorowych, chyba że liczba wykonywanych programów jest mniejsza niż liczba dostępnych procesorów. Opisaną sytuację, kiedy w systemie wykonywanych jest jednocześnie wiele programów nazywa się *wielozadaniowością* albo *wieloprogramowością*. Programy wykonywane *współbieżnie* (czyli równocześnie) w takim systemie nazywane są *procesami*.

Czasami mamy do czynienia z podobną, ale inną sytuacją: w obrębie jednego procesu-programu wykonywanych jest kilka niezależnych (choć powiązanych ze sobą) podprogramów. Na przykład używając przeglądarki www niejednokrotnie napotykamy problem załadownia pliku (być może dużego) z serwera na maszynę lokalną. Gdyby przeglądarka musiała czekać na całkowite ściągnięcie pliku przed rozpoczęciem dalszych działań byłoby to bardzo niewygodne. Dlatego ładowanie pliku odbywa się równocześnie (współbieżnie) z innymi działaniami

przeglądarki, w obrębie jednego procesu jaki ona stanowi. Taką sytuację nazywamy *wielowątkowością*, a wykonywane podprogramy - *wątkami*.

Java pozwala na pisanie programów wielowątkowych poprzez specjalne konstrukcje językowe, klasy oraz zarządcę wątków w maszynie wirtualnej. Tak więc w obrębie jednego programu w Javie może wykonywać się rownocześnie i niezależnie wiele podprogramów realizujących odrębne, acz powiązane ze sobą podzadania.

Podsumujmy:

Współbieżność jest abstrakcyjnym pojęciem oznaczającym jednoczesne wykonywanie wielu zadań: procesów, wątków, na jednej bądź wielu maszynach, jedno- bądź wieloprocesorowych.

Proces to program działający w systemie. To pojęcie obejmuje również zasoby, które on wykorzystuje: pamięć, pliki, konsola i inne.

Wątek to podprogram wykonujący się w procesie. Program wielowątkowy składa się z co najmniej jednego wątku i ich liczba może się zmieniać w trakcie wykonania tego programu.

Wywłaszczenie oznacza odebranie procesora aktualnie wykonującemu się zadaniu (procesowi, wątkowi) i przydzielenie go innemu.

1.2 Zastosowania wątków

Wątków używa się w programach, które muszą wykonywać wiele czynności równocześnie. Szczególnie dotyczy to zadań wymagających oczekiwania i mogących przez to zatrzymać wykonujący się program. Taką operację można zlecić niezależnemu wątkowi, nie przerywając działania całego programu (np. złożonych obliczeń). Przykładem może być obsługa wejścia-wyjścia (pliki, połączenia sieciowe) lub interakcja z GUI (obsługa zdarzeń).

Różnica między wielowątkowością a wieloprocesowością od strony programisty ujawnia się poprzez ułatwione korzystanie ze wspólnych zasobów (w przypadku wątków). Procesy działają w odrębnych przestrzeniach adresowych i wymiana danych pomiędzy nimi jest dość złożona. Wątki działają w jednej przestrzeni adresowej, więc mogą korzystać ze wspólnych danych (składowych obiektów) procesu - programu, w którym się wykonują.

Wykonywanie wielu zadań (wątków, procesów) w jednym systemie równocześnie ujemnie wpływa na wydajność, ponieważ część czasu pracy procesora poświęcona jest na ich szeregowanie. Wygoda w projektowaniu i użytkowaniu oprogramowania i sprzętu jaką otrzymuje się w zamian wynagradza jednak tę niedogodność. Trzeba pamiętać również, że czas potrzebny na przełączanie wątków jest dużo mniejszy niż czas potrzebny na zmianę wykonywanego procesu, ponieważ wątki działają na wspólnej przestrzeni adresowej. W systemach wieloprocesorowych można uzyskać pewne przyspieszenie, jeśli zadania będą wykonywały się na różnych procesorach.

2 Problemy związane ze współbieżnością

Niewłaściwe posługiwanie się wątkami lub procesami może doprowadzić do nieoczekiwanych sytuacji. Ogólnie rzecz biorąc polegają one na niemożności dalszego wykonywania jednego lub większej liczby zadań; być może całego współbieżnego programu, a nawet systemu operacyjnego. Aby ich uniknąć należy bardzo ostrożnie postępować z udostępnianiem zadaniom wspólnych zasobów. Przez zasób rozumie się zmienną, składową w klasie, obiekt (z punktu widzenia języka programowania), a także strumień, plik, fragment pamięci, dysk lub inne urządzenia w komputerze (z punktu widzenia systemu lub maszyny). Istnieje złożona teoria poświęcona koordynowaniu zadań, komunikacji pomiędzy nimi i współdzieleniu zasobów.

2.1 Współdzielenie zasobów

W samochodzie jest jedna kierownica i jedno miejsce dla kierowcy. Jeśli jedzie dwóch kierowców, to mogą oni się co jakiś czas zamieniać miejscami, aby prowadzić pojazd. Tylko jeden z nich może kierować w danym momencie. W samochodzie mamy też jedną przednią szybę. Nawet jeśli jest dwóch lub więcej kierowców, mogą oni jej używać (obserwować drogę) równocześnie bez ryzyka wystąpienia konfliktu. Ten przykład ilustruje (w bardzo uproszczony sposób) dwa problemy związane z jednoczesnym dostępem do zasobów przez wiele zadań:

Dostęp swobodny - odczyt

Przypadek szyby - możliwy jednoczesny, bezkonfliktowy dostęp przez wiele zadań. Szyba reprezentuje zasób, z którego mogą korzystać jednocześnie różne zadania, ponieważ nie jest on modyfikowany. Żadne z nich nie może również zawłaszczyć takiego zasobu, uniemożliwiając dostęp do niego innym.

Dostęp ograniczony - modyfikacja

Przypadek kierownicy - wymaga wyłączności w dostępie do zasobu, przynajmniej na czas modyfikacji. Kierownica reprezentuje zasób, który może być udostępniony w danej chwili tylko jednemu zadaniu - kierowcy. Ponadto, jeśli koordynacja zadań (komunikacja między kierowcami) jest niewłaściwa, to może dojść do sytuacji, w której jedno zadanie (kierowca) zawłaszczy zasób (kierownicę) i nie będzie chciało go oddać innym. System będzie musiał usunąć (wywłaszczyć) je siłą.

Uwaga

W praktyce programistycznej nie zawsze pobranie wartości zasobu (odczyt) jest operacją o dostępie swobodnym, nie wymagającą wyłączności - nawet jeśli wartość ta nie ulega zmianie w czasie. Operacje czytania z plików lub połączeń sieciowych mogą trwać dowolnie długo

prowadząc w efekcie do zawłaszczenia zasobu. W tego typu przypadkach trzeba koordynować współpracę zadań. Pomaga w tym zarządca wątków (ze strony maszyny wirtualnej Javy) lub procesów (ze strony systemu operacyjnego).

2.2 Wzajemne wykluczanie

Aby uniemożliwić wielu zadaniom jednoczesny dostęp do zasobu należy go w jakiś sposób zablokować, gdy jest używany przez jedno z nich. Kiedy zasób zostanie zwolniony przez aktualnego użytkownika, będzie można go przydzielić innym zadaniom. Używa się w tym celu rozmaitych mechanizmów. Mogą one stanowić część języka lub być dołączane jako zewnętrzne biblioteki. Można również w pomysłowy sposób wykorzystać blokujące operacje wejścia-wyjścia, o czym dalej. Niezależnie od sposobu blokowania i udostępniania zasobów należy unikać następujących anomalii:

Blokada (*impas, deadlock, zakleszczenie*) polega na zawłaszczeniu przez zadania zasobów w taki sposób, że żadne z nich nie może się dalej wykonywać. Przeważnie ma to miejsce, gdy każde z zadań wymaga do dalszego działania zasobu będącego aktualnie w wyłącznym posiadaniu przez inne. W takim przypadku program (zestaw zadań) przestaje się wykonywać i jedynym wyjściem jest jego zewnętrzne zakończenie.

Zagłodzenie występuje wtedy, gdy wśród wielu zadań oczekujących (w zawieszeniu) na dostęp do zasobu jedno lub więcej jest zawsze przy przydzielaniu zasobu ignorowany. Może to być spowodowane źle zaprojektowanym algorytmem lub niesprawiedliwym przydzielaniem procesora przez zarządce zadań w systemie.

2.3 Sekcja krytyczna

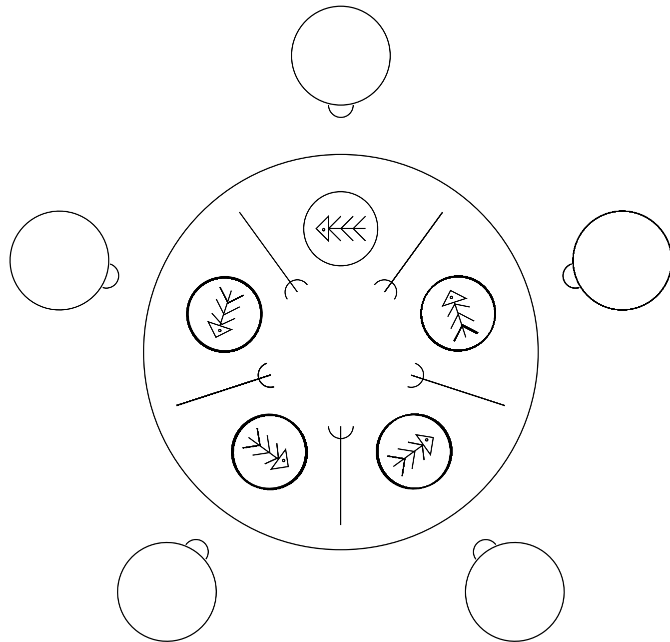
Zestaw instrukcji w programie, które są związane z wyłącznym dostępem do zasobu nazywa się *sekcją krytyczną*. W przykładzie z samochodem sekcją krytyczną było miejsce dla kierowcy. Może tam przebywać tylko jeden z nich. Ten, który tam się znajduje ma dostęp do zasobu - kierownicy.

W Javie wzajemne wykluczanie wątków w dostępie do zasobów osiąga się dzięki *synchronizatorom* (inaczej: *blokada, monitor, rygiel*). Są to zwykłe obiekty klasy `Object`, które w połączeniu ze słowem kluczowym `synchronized` działają jak drzwi do pomieszczenia będącego sekcją krytyczną. Wątki zawieszone na owym obiekcie można sobie wyobrażać jako oczekujących przed drzwiami do pomieszczenia. Po wykonaniu swojego zadania wątek opuszcza pomieszczenie, udostępniając je innym. O tym, który zostanie wpuszczony decyduje zarządca, jednak stara się on być sprawiedliwy w przydzielaniu dostępu do pomieszczenia.

2.4 Przykłady

2.4.1 Pięciu Filozofów

Problem pięciu filozofów jest klasycznym przykładem ilustrującym podstawowe zagadnienia współbieżności: blokadę i zagłodzenie.



Pięciu filozofów siedzi przy stole, na którym znajduje się pięć talerzy z rybami i pięć widelców. Filozofowie wykonują na przemian dwie czynności: jedzenie i myślenie. Do jedzenia potrzebne są dwa widelce, a więc nie mogą wszyscy jeść równocześnie. Trzeba przyjąć taką strategię przydzielania widelców, aby każdy z filozofów mógł się najeść.
Ponumerujmy filozofów i widelce liczbami [0..4]. Po prawej stronie $i-tego$ filozofa leży $i-ty$ widelec, po lewej - o numerze $(i+1)mod5$.

Rozwiązanie z możliwością zagłodzenia: Niech każdy z filozofów oczekuje momentu, w którym obydwa widelce jemu potrzebne będą wolne (dla $i-tego$ filozofa będą to widelce i i $(i{+}1)mod5$). Kiedy to nastąpi, podnosi je i zaczyna jeść. Niestety, taka strategia przydzielania widelców może doprowadzić do zagłodzenia jednego z filozofów w następujący sposób:

1. podnoszą widelce filozofowie 1 i 3.
2. kończą jeść i odkładają widelce
3. w tym momencie filozofowie 2 i 4 pobierają widelce.
4. teraz oni kończą jeść i odkładają widelce
5. ponownie punkt 1

W ten sposób filozof o numerze 0 zostanie zagłodzony, jeśli oczekiwał na widelce (konkurował o nie na przemian z $1-szym$ i $4-tym$, ale nigdy nie dostawał obydwu potrzebnych).

Rozwiązanie z możliwością blokady: Każdy z filozofów podnosi widelec, będący po jego prawej stronie, a następnie oczekuje na lewy. Jeśli wszyscy oni pobrali prawe widelce jednocześnie (zanim którykolwiek zdążył sięgnąć po lewy), to następuje blokada - żaden nie jest w stanie wykonać następnego posunięcia, ponieważ każdemu do działania potrzebny jest widelec, będący w posiadaniu przez innego filozofa.

Rozwiązanie prawidłowe wymaga zatrudnienia lokaja, który dopuszczałby do jedzenia co najwyżej czterech filozofów jednocześnie. Po jedzeniu, filozof musiałby ponownie prosić lokaja o pozwolenie na wejście do jadalni.

2.4.2 Współdzielenie zmiennych

Rozważmy poniższy fragment kodu i załóżmy, że dwa wątki: `T1` i `T2` wywołują metodę `swap()` (zamieniającą wartości zmiennych `from` i `to`) z tego samego obiektu klasy `Swapper`.

```
public class Swapper {                                          ---------->
                                        T1 start                            T2 start
    int from = 0;                  T2 czeka przed swap()           T1 wstrzymany po 1
    int to   = 1;
    public void swap() {                   T1                                 T2
            // 1:
        int temp = from;             temp == from == 0                   temp == 0
            // 2:                       ------> T2
        from = to;                      T1 <-----                        from == 1
            // 3:                      from == 0
        to = temp;                       to == 0                           to == 0
    }                                                                   T1 <-------
}                                  T1 kończy swap():                T2 kończy swap():
                                   from == to == 0                 from == 1, to == 0
```

Załóżmy, że wątek `T1` rozpoczął wykonywanie metody `swap()` i został wstrzymany po wykonaniu instrukcji 1. W tym momencie wartości zmiennych są następujące: `temp == 0` (w wątku `T1`) oraz `from == 0` i `to == 1` (w obydwu wątkach). Teraz wątek `T2` rozpoczyna

wykonywanie metody `swap()` na rzecz tego samego obiektu. Początkowe wartości zmiennych nie zmieniły się (`from == 0`, `to == 1`), więc po wyjściu z tej metody będą zamienione: `from == 1`, `to == 0`. Teraz niech wątek `T1` zostanie wznowiony i kontynuuje wykonywanie metody od instrukcji 2. Po wyjściu z metody wszystkie zmienne mają takie same wartości, wbrew oczekiwaniom co do metody `swap()`.

Widzimy więc, że niekorzystny przeplot wątków może spowodować zupełnie nieoczekiwane zachowanie programów. Przyczyną jest zezwolenie wątkom na swobodny, niekontrolowany dostęp do wspólnych zasobów (zmiennych). Lekarstwem na takie sytuacje jest synchronizowanie funkcji (lub bloków kodu - sekcji krytycznych), o czym dalej. Ważne jest również odpowiednie zaprojektowanie algorytmu.

3 Wątki w Javie

Początkowo maszyna wirtualna powołuje do życia wątek główny, odpowiedzialny za wywołanie funkcji `public static void main(String[] args)` z klasy startowej. Podczas jego wykonywania można tworzyć wątki składające się na program. Oczywiście nie muszą one być tworzone wyłącznie w głównym wątku systemowym, ale również we wszystkich pozostałych wątkach. Oprócz głównego wątku systemowego maszyna wirtualna czasem uruchamia jeszcze dodatkowe wątki wspomagające (np. do obsługi zdarzeń).

Ponieważ Java dostarcza własnych mechanizmów zarządzania współbieżnością (ukrytych w VM), wydawałoby się, że programy będą wykonywać się podobnie niezależnie od systemu operacyjnego. Tak jednak nie jest, ponieważ niektóre systemy dodatkowo wspomagają maszynę wirtualną w zarządzaniu wątkami (Win95/98/NT/2000, Solaris, OS/2), a inne nie (DOS). W systemie Linux wątki Javy (i nie tylko) są realizowane przy pomocy zewnętrznych bibliotek ze względu na brak wsparcia dla wielowątkowości ze strony jądra. Takie wątki są faktycznie procesami Linuxa, mają jednak wspólne dane (za wyjątkiem numeru procesu, który powinien być identyczny we wszystkich wątkach, a nie jest).

Ponadto relacje pomiędzy wątkami Javy a systemowymi zależą od użytej VM i modelu wielowątkowości dostarczanego przez system. Generalnie - z punktu widzenia maszyny wirtualnej Javy - są dwa takie modele:

- oparty na *współpracy* - wątek sam decyduje kiedy chce oddać procesor innym (przeważnie po wywołaniu określonych funkcji wspomagających)
- oparty na *wywłaszczaniu* - wątek dostaje *kwant* czasu na dostęp do procesora i po jego upłynięciu jest wywłaszczany przez zarządcę, który przydziela procesor innemu wątkowi.

Aby uniknąć problemów z przenośnością programów trzeba je bardzo ostrożnie projektować. Najlepiej jest przyjąć, że system będzie wywłaszczał nasz wątek, a oprócz tego co jakiś czas oddawać dobrowolnie procesor innym wątkom przy użyciu odpowiednich funkcji (o czym dalej), aby nie dopuścić do ewentualnego zawłaszczenia procesora przez jeden wątek.

W systemach wspierających wątki (bądź je emulujących - jak Linux) można zauważyć, że na proces maszyny wirtualnej składa się więcej wątków, niż uruchomił użytkownik poprzez program w Javie. Jest tak dlatego, że VM prowadzi pewne działania (np. odśmiecanie) jako niezależne wątki systemowe, jednak nie są to zwykłe wątki Javy.

3.1 Implementacja

Wbrew definicji wątku, w Javie nie operują one na wspólnej pamięci procesu, lecz tworzą prywatne kopie zmiennych. Zdecydowano się na takie rozwiązanie ze względów efektywnościowych. Kopie zmiennych w każdym wątku są uzgadniane z oryginałem w momentach synchronizacji (przy wejściu do i wyjściu z bloku synchronizowanego - patrz dalej). Jeśli takie momenty nie występują albo są rzadkie, to może dojść do sytuacji kiedy kopia różni się od oryginału. Aby temu zapobiec należy zmienne współdzielone przez wątki (których wartość może być zmieniana w trakcie wykonania przez różne wątki) poprzedzać specyfikatorem `volatile`. Taka deklaracja powoduje, że wątki będą uzgadniały swoją kopię zmiennej z oryginałem przy każdym odwołaniu do tej zmiennej.

Operacje na zmiennych typu `long` i `double` mogą nie być atomowe. Zależy to od implementacji VM (w rzeczywistości od architektury sprzętowej). Zmienne tego typu mogą być traktowane jak dwa niezależne słowa 32-bitowe. Operacje na tych zmiennych są wtedy rozbijane na dwie części: osobno dla każdej z 32-bitowych połówek. Jeśli chcemy uzyskać atomowość

(niepodzielność) operacji na zmiennych tego typu należy poprzedzać ich deklarację słowem kluczowym `volatile` albo synchronizować kod. Oczywiście ma to sens głównie w sytuacji gdy więcej niż jeden wątek ma do takiej zmiennej dostęp, w tym przynajmniej jeden z możliwością zapisu. Jeśli się tego nie zrobi, to może się zdarzyć, że dwa wątki usiłujące równocześnie zapisać wartość zmiennej np. typu `long`, mogą pozostawić ją w stanie nieokreślonym - oto scenariusz:

1. wątek **A** zapisuje pierwszą połowę zmiennej
2. wątek **A** zostaje wywłaszczony
3. wątek **B** zapisuje całą zmienną (obie połowy)
4. wątek **A** dostaje procesor i zapisuje drugą połowę zmiennej
5. teraz zmienna składa się z pierwszej części zapisanej przez **B**, i drugiej zapisanej przez **A**.

Jak widać przykład ten jest bardzo podobny do funkcji `swap()` z klasy `Swapper`.

W przyszłości ta sytuacja ulegnie zmianie. Jedynym powodem specyficznego traktowania tych zmiennych jest fakt, że pewne popularne procesory nie potrafią efektywnie wykonywać atomowych operacji na 64 bitowych typach danych. Zatem jest to cecha zależna od implementacji i na 64-bitowych architekturach można spodziewać się atomowego traktowania `long` i `double`.

3.2 API - wstęp

Wątki są obiektami klasy `Thread`. Utworzenie obiektu tej klasy nie oznacza jeszcze ropoczęcia wykonywania nowego ciągu instrukcji. Dopiero wydanie mu polecenia `start()` powoduje zainicjowanie wątku i rozpoczęcie wykonywania jego (tj. wątku) metody `run()`.

Metoda `run()` jest dla wątku tym, czym `main()` dla aplikacji. Prawie zawsze zawiera pętlę, która jest wykonywana przez cały czas życia wątku. Zakończenie wykonywania tej funkcji oznacza zakończenie pracy wątku.

Metoda `run()` jest zadeklarowana w interfejsie `Runnable`. Można umieścić jej definicję w dowolnej klasie (implementując w ten sposób `Runnable`) i w oparciu o nią stworzyć wątek.

Nie trzeba przechowywać odniesienia do obiektu wątku (z obawy przed odśmieceniem). Dopóki wątek działa, nie zostanie usunięty przez zarządcę nieużytków. Maszyna wirtualna zapamiętuje odnośnik do niego podczas inicjalizacji i przechowuje do momentu jego zakończenia.

3.3 Tworzenie wątków

Tworzenie wątku polega na dostarczeniu metody `run()` do fabrykowanego obiektu klasy `Thread` (lub jej podklasy). Sa dwa podstawowe schematy postępowania: dziedziczenie klasy `Thread` lub implementowanie interfejsu `Runnable`.

3.3.1 Dziedziczenie klasy `Thread`

```
class MyThread extends Thread {

    public void run(){
        // treść wątku
    }

    MyThread(){
    }

        // możliwe użycie w konstruktorze
    MyThread(String name){
        this.start();
    }

        // przykładowe użycie w programie
    public static
    void main(String[] args]){

        new MyThread().start();
            // albo tylko tyle:
        new MyThread("start() w konstruktorze");
    }
}
```

Ponieważ klasa `Thread` implementuje interfejs `Runnable`, więc zawiera metodę `run()`. Ma ona jednak puste ciało, a więc nic nie robi. W podklasie klasy `Thread` możemy ją przesłonić, definiując w ten sposób zadanie jakie ma wykonywać wątek.

Po sfabrykowaniu obiektu podklasy wywołujemy na jego rzecz metodę `start()` (odziedziczoną z klasy `Thread`), która inicjuje wątek, a następnie wywołuje (naszą) metodę `run()`. Wywołanie metody `start()` można w tym przypadku również umieścić w konstruktorze tej klasy.

Tego sposobu używa się, gdy chcemy mieć bezpośredni dostęp do specyficznych funkcji klasy `Thread` (np. z wnętrza funkcji `run()`).

3.3.2 Implementowanie interfejsu `Runnable`

```
class MyApp implements Runnable {

    public void run(){
        // treść wątku
    }

        // użycie w metodzie tej klasy
    void init(){
        new Thread(this).start();
    }

        // użycie w programie
        // na przykładzie funkcji main()

    public static
    void main(String[] args]){

        MyApp app = new MyApp();
        new Thread(app).start();
    }
}
```

Interfejs `Runnable` zawiera tylko metodę `public void run()`. Nasza klasa implementując ten interfejs musi zdefiniować tylko tę metodę. Zatem sensowne jest dodanie tej metody do jakiejś większej klasy zawierającej dane i funkcje, z których mógłby korzystać wątek (oczywiście trzeba dodać też klauzulę `implements Runnable`). Nowy wątek można wtedy utworzyć konstruktorem `Thread(Runnable r)` tak: `new Thread(this)`. Następnie uruchamia się go metodą `start()`.

Uwaga: będzie wykonywana metoda `run()` klasy implementującej interfejs, a nie klasy `Thread`.

3.3.3 Klasa anonimowa

```
class ThreadUse {

    Object field;

    void fun(final Object parameter){

        final Object variable = new Object();

        new Thread(
            new Runnable(){
                public void run(){
                        // ew. użycie zmiennych
                        // z funkcji otaczającej
                    Object v = variable;
                    Object p = parameter;
                        // albo z klasy
                        // (można bez final)
                    Object f = field;
                }
            }
        ).start();
    }
}
```

Szczególnym - często stosowanym - przypadkiem drugiego schematu jest stworzenie wątku na bazie klasy anonimowej implementującej interfejs `Runnable` (a więc dostarczającej metodę `run()`). Ten sposób jest użyteczny, gdy chcemy utworzyć dokładnie jeden wątek wykonujący dane zadanie. Nie chcemy produkować wtedy nowej klasy dedykowanej temu zadaniu, ani też dodawać do istniejącej klasy funkcji, która zostanie użyta tylko raz.

Przypomnienie: jeśli zamierzamy używać w metodzie `run()` takiego wątku zmiennych zadeklarowanych w funkcji otaczającej klasę anonimową, to trzeba ich deklaracje poprzedzić specyfikatorem `final`.

3.4 Zatrzymywanie wątków

```
class ThreadStop implements Runnable {

    public static void main(String[] args){
        ThreadStop ts = new ThreadStop();
        Thread thr = new Thread(ts);
        thr.start();
            // ....
        thr.interrupt();
    }
    public void run(){

        boolean done = false;
        while(!done){
            done = doSomething();
            try {
                Thread.sleep(czas);
            }
            catch(InterruptedException){
                if(done)
                    return;
            }
        }
    }
}
```

Naturalne zakończenie pracy wątku następuje po wyjściu z jego metody `run()`. Jeśli zawiera ona nieskończoną pętlę, to przeważnie sprawdza się wewnątrz tej pętli jakiś warunek (np. zmienną) rozstrzygający, czy można już zakończyć jej wykonywanie.

Jeśli wątek jest uśpiony (funkcja `Thread.sleep(long)`) albo wstrzymany (np. na synchronizatorze), to należy najpierw mu wydać polecenie `interrupt()`, aby wyszedł z tego stanu. Spowoduje to, że po uaktywnieniu odbierze on wyjątek `InterruptedException`. Wtedy można go w normalny (jak wyżej) sposób zakończyć.

Metoda `stop()` z klasy `Thread`, która służyła do kończenia wątków we wcześniejszych wersjach Javy jest niebezpieczna i nie należy jej używać.

Wątku nie można ponownie uruchomić.

3.5 Synchronizowanie

W Javie, aby zagwarantować wyłączność dostępu do sekcji krytycznej używa się słowa kluczowego `synchronized`. Synchronizowane (tzn. poprzedzone tym słowem) mogą być bloki instrukcji lub funkcje (które właściwie też są blokami instrukcji).

Jak wspomniano przy okazji omawiania implementacji wątków Javy, przy wejściu do synchronizowanego bloku wątek uzgadnia z pamięcią główną procesu wartości współdzielonych zmiennych, natomiast kończąc wykonywanie tego bloku zapisuje w pamięci głównej wartości zmiennych, które się zmieniły podczas jego wykonania. Operacje te nie są atomowe (jako całość): oznacza to, że wątek może zostać wstrzymany w trakcie takiej operacji. W efekcie dane w wątku i pamięci głównej mogą przez pewien czas być różne, co może mieć fatalne konsekwencje jeśli nie zapewni się właściwej synchronizacji (przykład poniżej).

3.5.1 Synchronizowanie bloków

Instrukcja synchronizowana ma postać `synchronized(o){ /* instrukcje */ }`, gdzie `o` jest odniesieniem do obiektu (klasy `Object`), nazywanego *synchronizatorem*, *blokadą*, *ryglem* albo *monitorem*. Taka zmienna powinna być zadeklarowana jako `final`. Jeśli jakiś wątek rozpocznie wykonywanie bloku instrukcji (umieszczonego w nawiasach klamrowych po słowie `synchronized`), to każdy inny wątek, który napotka taką instrukcję z tym samym synchronizatorem zostanie wstrzymany do momentu, gdy pierwszy zakończy wykonywanie tego bloku

(albo wywoła metodę `wait()` na rzecz tego samego rygla - o czym dalej). Ze wszystkich wstrzymanych w ten sposób wątków zostanie uruchomiony jeden wybrany przez zarządcę. Uzyskujemy w ten sposób gwarancję, że w każdej chwili co najwyżej jeden wątek będzie wykonywał instrukcje bloku synchronizowanego. Sposób wyboru wątku przez VM jest sprawiedliwy, tzn. nie dopuszczający do zagłodzenia żadnego z wątków.

```
class SynBlock implements Runnable {
    int buf = 0;
    volatile int threadId = 1;
    final Object lock = new Object();

    public void run(){
        int local = threadId++;
        while(true){
            synchronized(lock){     // !
                buf = local;
                    // możliwe wywłaszczenie
                if(buf != local)
                    System.out.println(
                        local + " " + buf
                    );
            }
        }
    }
}
```

Poniższy przykład pokazuje jak używać bloków synchronizowanych dla zapewnienia wzajemnego wykluczania w dostępie do współdzielonych zasobów. Jeśli usuniemy synchronizację, program będzie wypisywał pary liczb! Dzieje się tak dlatego, że wątek może zostać wywłaszczony zaraz po przypisaniu `buf = local`. Po wznowieniu uzgodni swoją kopię zmiennej `buf` z jej globalnym odpowiednikiem i okaże się, że są one różne (bo inny wątek zmodyfikował `buf`).

```
public static void main(String[] args){
    SynBlock sblk = new SynBlock();
    new Thread(sblk).start();
    new Thread(sblk).start();
    new Thread(sblk).start();
}
```

3.5.2 Synchronizowanie metod

Metoda synchronizowana to metoda zadeklarowana ze specyfikatorem `synchronized`: `synchronized Type fun(...){}`. Taką deklarację kompilator niejawnie zastępuje przez:

```
Type fun( ... )
{
    synchronized(lock){
        // instrukcje
    }
}
```

Przy czym:

- jeśli `fun()` **nie jest** funkcją statyczną, to `lock` jest odnośnikiem `this`.
- jeśli `fun()` **jest** funkcją statyczną w klasie `Klasa`, to `lock` jest literałem klasowym `Klasa.class`.

A zatem synchronizowane funkcje są po prostu zwykłymi funkcjami, których ciała są blokami instrukcji synchronizowanymi przez konkretne rygle. Podobne są również zasady dostępu wątków do takich funkcji.

Jeśli funkcja jest synchronizowana, to może być wywołana tylko przez jeden wątek na rzecz danego obiektu. Jeśli zostanie ona wywołana (na rzecz tego samego obiektu) z innych wątków, to zostaną one wstrzymane do czasu zakończenia wykonywania pierwszego z nich (lub wywołania w nim metody `wait()` na rzecz obiektu synchronizującego). Podobnie, jeśli na rzecz tego samego obiektu zostanie wywołana inna metoda synchronizowana, zanim zakończy się pierwsze wywołanie, to wołający wątek zostanie wstrzymany. Oczywiście można niezależnie wywoływać te metody na rzecz różnych obiektów (będą wtedy różne rygle).

3.5.3 Przykład braku synchronizacji

```
class SynchShow {
    int v = 0;
    int w = 1;

    synchronized
    void foo(){
        v = 2;
        w = 3;
    }
    void bar(){
        System.out.println(
            " v = " + v +
            " w = " + w
        );
    }
}
```

Załóżmy, że wątek **F** rozpoczął wykonywanie funkcji `foo()`, a wątek **B** - funkcji `bar()` na rzecz tego samego obiektu klasy `SynchShow`. Dodatkowo przyjmijmy, że **F** został wstrzymany przy wyjściu z `foo()` podczas odsyłania wartości zmiennych do pamięci głównej (`foo()` jest synchronizowana !). Ponieważ `bar()` nie jest synchronizowana, więc **B** może być wznowiony i odczytać wartości zmiennych `v, w`. Jakie one będą? Niestety, mogą być dowolne, tzn. `v == 0` lub `v == 2` jak również `w == 1` lub `w == 3`. Wynika to z faktu, że kolejność uzgadniania kopii zmiennych w wątku z pamięcią główną przy wejściu do i wyjściu z bloku synchronizowanego nie ma nic wspólnego z kolejnością wykonywanych przypisań w kodzie. Jednak, jeśli `bar()` będzie funkcją synchronizowaną, to do takiej sytuacji nie dojdzie, ponieważ **B** nie będzie uruchomiony, zanim **F** całkowicie nie zakończy wykonywania `foo()`.

Oczywiście prawdopodobieństwo, że metoda `bar()` pokaże wyniki `v = 0` lub `w = 1` jest bardzo małe i zależy od implementacji konkretnej VM, jednak specyfikacja języka to dopuszcza.

Wniosek: wszystkie funkcje w klasie (dowolnej), które mają dostęp do składowych tej klasy powinny być synchronizowane. Zagwarantuje to, że w każdej chwili tylko jeden wątek będzie miał do nich dostęp. W przeciwnym wypadku może dojść do powyższej anomalii.

3.5.4 Przykład złej synchronizacji

Początkowo tworzone są dwa obiekty klasy `FooBar`: `foo` i `bar`, na bazie których powstają dwa wątki. Metoda `run()` wątków wywołuje funkcję synchronizowaną `oops()` tej klasy, która wykonuje dwie instrukcje. Najpierw usypia na krótki okres czasu bieżący wątek, a następnie wywołuje tę samą metodę `oops()` na rzecz innego obiektu (`other`) tej klasy.

Wątek oparty na `foo` rozpoczynając wykonywanie metody `run()` od wywołania `this.oops()` powoduje zajęcie rygla `this` (`== foo`). Następnie zostaje uśpiony, co umożliwia wątkowi opartemu na `bar` rozpoczęcie wykonywania metody `run()`. On także na początku zajmuje rygiel `this` (`== bar`), wywołując `this.oops()` - po czym zasypia. Oba synchronizatory są od tej pory zajęte. Wtedy budzi się pierwszy wątek i próbuje wykonać drugą instrukcję: `other.oops()`. Ale tu `other == bar`, a metoda `oops()` jest synchronizowana i może być wywoływana tylko przez jeden wątek z obiektu `bar`, co właśnie zrobił drugi z nich. Zatem pierwszy wątek (oparty na `foo`) wisi na ryglu `bar` przed wywołaniem `bar.oops()`. Wtedy budzi się z uśpienia drugi wątek (oparty na `bar`) i również próbuje wywołać `other.oops()`, ale tu `other == foo`. Niestety, rygiel `foo` jest już zajęty przez pierwszy wątek i `bar` musi czekać aż zostanie on zwolniony.

Każdy z wątków potrzebuje dwóch zasobów do działania, a ma tylko jeden. Musi zatem czekać aż inny wątek zwolni drugi zasób. To jest klasyczny przykład blokady. Wątki przestają się wykonywać i trzeba je siłą usunąć.

```
class LockShow {

    public static void main(String[] args){
        new LockShow();
    }
    FooBar foo = new FooBar("foo");
    FooBar bar = new FooBar("bar");

    LockShow(){
        foo.other = bar;
        bar.other = foo;
        new Thread(foo).start();
        new Thread(bar).start();
    }
    void sleep(String who){
        try {           // uśpienie na 1 ms
            Thread.sleep(1);
        }
        catch(InterruptedException e){
        }
        System.out.println(who);
    }
}
```

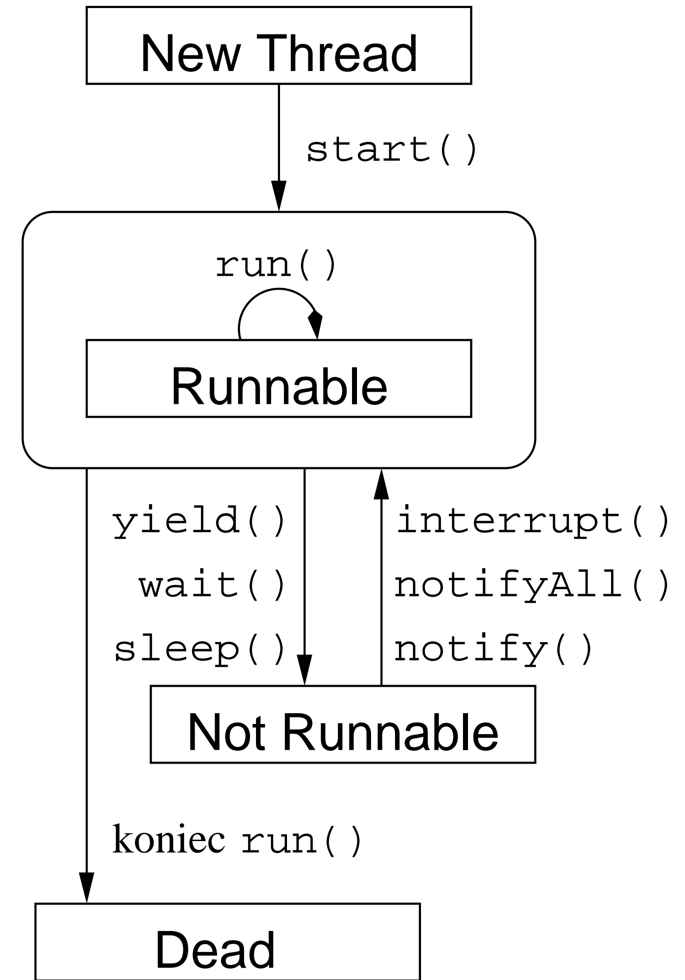
```
class FooBar implements Runnable {

    String me;
    FooBar other;

    FooBar(String id){
        me = id;
    }
    public void run(){
        this.oops();
            // to się nie wykona:
        System.out.println(me+" done");
    }
    synchronized void oops(){
        sleep(me);
        other.oops();   // blokada !
    }
}
```

Uwaga: uśpienie jest potrzebne tylko, aby przekazać sterowanie do drugiego wątku.

3.6 Stany wątków

Wątek może znajdować się w jednym z czterech stanów: *nowy*, *uruchamialny*, *uśmiercony*, *zablokowany*.

1. Nowy (*New*): po utworzeniu obiektu wątku, ale przed uruchomieniem - wywołaniem metody `start()`. Taki wątek jeszcze się nie wykonuje.
2. Uruchamialny (*Runnable*): wątek jest gotowy do działania - wywołano metodę `start()`. Zostanie uruchomiony (zapewne wielokrotnie), kiedy tylko zostanie mu udostępniony procesor.
3. Uśmiercony (*Dead*): po zakończeniu wykonywania metody `run()`.
4. Zablokowany (*Not Runnable*): wątek oczekuje na zajście jakiegoś zdarzenia. Zostanie wznowiony kiedy to zdarzenie nastąpi.



Wątki zablokowane

Wątek może być zablokowany z pięciu powodów:

1. Uśpienie po wywołaniu metody `sleep(long)`. Wątek zostanie wznowiony po upływie przewidzianego czasu lub gdy zostanie mu wydane polecenie `interrupt()` (odbierze wtedy wyjątek `InterruptedException`).
2. Zawieszenie po wywołaniu metody `suspend()`. Taki wątek zostanie wznowiony gdy inny wątek wywoła z niego metodę `resume()`. Metody te nie są zalecane w obecnej wersji Javy.
3. Zawieszenie po wywołaniu metody `wait()`. Zostanie wznowiony po wywołaniu `notify()` lub `notifyAll()` na rzecz synchronizatora, na którym został zawieszony.
4. Oczekiwanie na zakończenie blokującej operacji (np. wejście-wyjście).
5. Zawieszenie na synchronizatorze przed wejściem do sekcji krytycznej, np. po wywołaniu metody synchronizowanej.

Po zakończeniu wykonywania metody `run()` obiekt wątku może jeszcze istnieć, jednak wywoływanie na jego rzecz metody `start()` jest niedozwolone. Innymi słowy wątek nie może przejść ze stanu *uśmiercony* do żadnego innego stanu.

3.7 Sterowanie wykonaniem

Mimo iż o tym, któremu wątkowi zostanie przydzielony procesor decyduje zarządca, to jednak za pośrednictwem pewnych mechanizmów można wpływać na jego decyzje. Jest to ważny element programowania wielowątkowego.

3.7.1 Zawieszanie i wznawianie

W klasie `Object` zdefiniowano metody służące do wzajemnej koordynacji wątków wewnątrz sekcji krytycznej (czyli metody albo bloku synchronizowanego). Metoda `wait()` służy do zawieszenia wątku na obiekcie synchronizatora - przechodzi on do stanu *zablokowany*. Metody `notify()` i `notifyAll()` służą do wznawiania wątków zawieszonych na synchronizatorze czyli takich, które wywołały przedtem `wait()` na rzecz tego samego synchronizatora. Pierwsza z tych metod wznawia jeden wybrany przez system wątek zawieszony na blokadzie (po wywołaniu `wait()`). Druga wznawia wszystkie - przechodzą one ze stanu *zablokowany* do *uruchamialny*.

Po wywołaniu `o.wait()` wątek zwalnia blokadę `o`, aby umożliwić innym wątkom wejście do bloku przez nią synchronizowanego. Następnie, po wywołaniu `o.notify()` lub `o.notifyAll()` z innego wątku, ponownie ją zajmuje (o ile został wybrany przez zarządcę) i rywalizuje wraz z innymi wątkami o ponowne wejście do tego bloku. Kiedy po przydzieleniu mu procesora,

rozpocznie działanie - opuści metodę `wait()` i będzie kontynuował wykonanie bloku od instrukcji następującej po `o.wait()`. Może się zdarzyć, że po wyjściu z `wait()` wątek nie od razu otrzyma procesor i inne wątki dostaną się przed nim do sekcji krytycznej.

Powyższe metody mogą być wywołane tylko na rzecz tego obiektu synchronizatora, który był wcześniej użyty jako blokada sekcji krytycznej. W przeciwnym wypadku program skompiluje się, ale w czasie wykonania zostanie zgłoszony wyjątek `IllegalMonitorStateException`. Tak samo stanie się w przypadku wywołania tych metod poza sekcją krytyczną (tzn. blokiem albo funkcją synchronizowaną).

Ten sposób zawieszania i wznawiania wątków różni się od omówionego wcześniej (zawieszenie na blokadzie przed wejściem do sekcji krytycznej) tym, że po wywołaniu `wait()` wątek może zostać wznowiony tylko przez inny wątek - znajdujący się w sekcji krytycznej - poprzez wywołanie `notify()` lub `notifyAll()`. Umożliwia również zatrzymanie wykonywania wątku wewnątrz bloku synchronizowanego (np. w oczekiwaniu na zajście jakiegoś zdarzenia).

Są jeszcze dwie wersję metody `wait()`: pobierają one jako argument(y) czas, po upływie którego wątek zostanie automatycznie wznowiony, o ile wcześniej nie stanie się to na skutek wywołania `notify()` lub `notifyAll()`. Metody `wait()` są użyteczne w sytuacjach, gdy będąc w sekcji krytycznej, wątek musi zaczekać na zajście jakiegoś zdarzenia nie opuszczając jej.

Przykład koordynacji

Klasa `ProdConsDemo` demonstruje użycie funkcji `wait()` i `notify()`. Obiekty klasy wewnętrznej `Producer` produkują coś i umieszczają w buforze `buffer`. Liczbę porcji dostarczanych przez danego producenta podaje się w konstruktorze. Po umieszczeniu porcji w buforze producent informuje o tym konsumentów poprzez `lock.notify()`. Konsument - obiekt klasy `Consumer` - pobiera tyle porcji z bufora, ile potrzebuje (argument konstruktora). Jeśli w buforze brak wystarczającej liczby porcji, to zawiesza się (wywołując `lock.wait()`) w oczekiwaniu na wyprodukowanie kolejnych porcji. Po wznowieniu ponownie sprawdza czy może przystąpić do konsumpcji, i jeśli tak - robi to.

Gdyby w instrukcji po `//!` zamiast pętli `while` znalazła się instrukcja warunkowa `if`, to mogłoby dojść do pobrania z bufora większej liczby porcji niż się tam znajduje. Wynika to z faktu, że po wznowieniu wątku konsumenta (na skutek `notify()`), zanim rozpocznie on wykonywanie kolejnej instrukcji (tu `buffer -= volume`), inny wątek konsumenta może dostać się do sekcji krytycznej i skonsumować pewną liczbę porcji.

Bufor jest zasobem współdzielonym między wszystkich producentów i konsumentów (których można stworzyć dowolną liczbę), dlatego dostęp do niego musi odbywać się w sekcji krytycznej (strzeżonej przez `lock`).

```java
class ProdConsDemo {

    int buffer = 0;
    final Object lock = new Object();

    class Consumer implements Runnable {
        int volume;
        Consumer(int v){
            volume = v;
        }
        public void run(){
            while(true){
                synchronized(lock){
                        // !
                    while(buffer < volume){
                        try {
                            lock.wait();
                        }
                        catch(InterruptedException e){
                        }
                    }
                    buffer -= volume;
                }
            }
        }
    }
```

```java
    class Producer implements Runnable {
        int volume;
        Producer(int v){
            volume = v;
        }
        public void run(){
            while(true){
                synchronized(lock){
                    buffer += volume;
                    lock.notify();
                }
            }
        }
    }
    ProdConsDemo(){
        new Thread(new Producer(1)).start();
        new Thread(new Producer(2)).start();
        new Thread(new Consumer(1)).start();
        new Thread(new Consumer(2)).start();
        new Thread(new Consumer(3)).start();
    }
    public static void main(String[] args){
        new ProdConsDemo();
    }
}
```

Lepszy przykład

W poprzednim przykładzie obowiązek koordynacji wątków spoczywał na użytkowniku - programiście. Java umożliwia przerzucenie tego obowiązku na barki projektanta klasy. Jak wiadomo, możliwa jest taka implementacja klasy, że tylko jeden wątek będzie mógł w danej chwili używać konkretnego jej obiektu - będzie ona wątkowo bezpieczna (*thread safe*). Oznacza to, że programista nie będzie musiał się przejmować wzajemnym wykluczaniem wątków przy dostępie do współdzielonego zasobu zawartego w obiekcie. Zrobią to za niego odpowiednio zaprojektowane metody takiej klasy. Programista używa tych metod w zwykły sposób. Cała synchronizacja zachodzi w ich wnętrzu. Wiele standardowych klas Javy jest domyślnie synchronizowanych, metody tych klas są wątkowo bezpieczne (które to są - opisuje dokumentacja).

Jak to zrobić? Przede wszystkim metody danej klasy - przynajmniej te, które odwołują się do jej składowych - muszą być synchronizowane. Czasami może być potrzebna dodatkowa koordynacja, co pokazuje poniższy przykład. Oczywiście synchronizacja jest niezbędna wtedy, gdy przynajmniej jedna metoda modyfikuje jakąś składową obiektu.

Poniższy przykład ilustruje tę technikę: obowiązek koordynacji wątków przejmuje bufor - obiekt klasy `Buffer`. Obiekty klas `Producer` i `Consumer` po prostu wywołują (synchronizowane) funkcje przez niego dostarczone: `get()` do pobierania z bufora i `put()` w celu wstawiania do bufora.

```
class Buffer {
    int amount = 0;
    synchronized void put(int v){
        amount += v;
        notifyAll();   // !
    }
    synchronized int get(int v)
            throws InterruptedException {
        while(amount < v)
            wait();
        amount -= v;
        return amount;
    }
}
class Producer implements Runnable {
    int volume;
    Buffer buffer;
    Producer(int v, Buffer b){
        volume = v;
        buffer = b;
    }
    public void run(){
        while(true)
            buffer.put(volume);
    }
}
```

```
class Consumer implements Runnable {
    int volume;
    Buffer buffer;
    Consumer(int v, Buffer b){
        volume = v;
        buffer = b;
    }
    public void run(){
        while(true){
            try {
                buffer.get(volume);
            }
            catch(InterruptedException e){
                return;
            }
        }
    }
}
class ProdConsSynchr {
    public static void main(String[] args){
        Buffer buf = new Buffer();
        new Thread(new Producer(1, buf)).start();
        new Thread(new Consumer(2, buf)).start();
        new Thread(new Consumer(3, buf)).start();
    }
}
```

3.7.2 Usypianie

Często stosowaną metodą jest `static void sleep(long)` z klasy `Thread`. Pozwala ona uśpić wątek (dowolny - bo jest statyczna) na czas podany jako argument tej metody. Oczywiście w tym czasie procesor jest przydzielony innym wątkom, ale po jego upływie zostanie zwrócony wątkowi uśpionemu.

3.7.3 Przekazywanie sterowania

Wątek może zaczekać na zakończenie wykonywania innego wywołując na jego rzecz metodę `join()` (zarządca przydziela wtedy procesor innemu wątkowi).

Wątek może dobrowolnie zrezygnować z procesora wywołując z klasy `Thread` metodę `static void yield()`. Jest to użyteczne, w systemach nie wspierających wywłaszczania, albo robiących to źle. Można w ten sposób uniknąć ryzyka zawłaszczenia procesora przez jeden wątek. Po ponownym przydzieleniu sterowania temu wątkowi przez zarządcę, rozpocznie on wykonywanie instrukcji za wywołaniem tej funkcji.

Uwaga: metoda `yield()` spowoduje przekazanie sterowania do innego wątku, tylko gdy oczekuje na nie jakiś wątek o priorytecie (o priorytetach niżej) równym lub większym od priorytetu wywołującego `yield()`.

3.7.4 Blokujące wejście-wyjście

Oprócz zawieszania wątków za pomocą mechanizmów języka (lub metod), znane są też inne sposoby na wstrzymanie wątku. Do najważniejszych należy oczekiwanie na zakończenie operacji przesyłania danych. Jeśli wątek próbuje odczytać metodą `read()` dane ze strumienia w momencie gdy nie są one jeszcze dostępne (celowy brak danych albo opóźnienia spowodowane obciążeniem dysków, sieci itp.), to zostanie wstrzymany do momentu, gdy się pojawią. Ta technika bywała w przeszłości używana do synchronizacji procesów.

3.8 Priorytety

Wątki mogą mieć priorytety, będące liczbą z przedziału [1..10]. Zarządca zadań chętniej uruchamia te z wyższym priorytetem. Oznacza to, że będą one miały więcej czasu procesora niż te o niższych priorytetach. Do manipulacji priorytetami służą metody `getPriority()` i `setPriority()` z klasy `Thread`. Początkowo wątek główny ma domyślny priorytet równy 5 (stała `Thread.NORM_PRIORITY`). Wątek oddelegowany do obsługi zdarzeń i interakcji z GUI ma priorytet równy 6. Operacje wejścia-wyjścia wykonywane są z priorytetem 4. Nowy wątek dziedziczy priorytet po wątku, który go stworzył. Ze względu na możliwość zagłodzenia wątków o niskich priorytetach zarządca nie gwarantuje, że do procesora dopuszczony będzie wątek o najwyższym priorytecie z oczekujących.

Uwaga

W systemach operacyjnych wspierających wątki (lub je emulujących) - czyli właściwie wszystkich obecnie używanych - VM mapuje priorytety wątków Javy na priorytety wątków (procesów) systemowych. Ponieważ w różnych systemach są różne liczby priorytetów może to prowadzić do nieoczekiwanych zachowań. Na przykład w systemie Windows NT jest 7 priorytetów wątków. Oznacza to, że kilka priorytetów Javy jest mapowanych na jeden priorytet NT. Co gorsza - jądro systemu NT może zwiększać priorytet wątków nie informując o tym maszyny wirtualnej. W efekcie zachowanie się programu może być różne od oczekiwanego. Dla odmiany w systemie Solaris mamy 2^{31} priorytetów i w związku z tym różne możliwości mapowania.

Wniosek: nie należy uzależniać logiki programu od priorytetów wątków. Powinny one służyć głównie do dostrojenia wykonywania się programu w danym środowisku.

3.9 Rodzaje wątków

Są dwa rodzaje wątków o różnym przeznaczeniu: demony i nie-demony. Do manipulowania rodzajami wątków służą funkcje `isDaemon()` i `setDaemon(boolean)` z klasy `Thread`.

3.9.1 Zwykłe (nie-demony)

Są to normalne wątki, wykonujące podstawowe zadania. Program działa, dopóki istnieje co najmniej jeden aktywny zwykły wątek. W momencie zakończenia ostatniego takiego wątku program kończy się.

3.9.2 Demony

Są to wątki, które działają w tle, wykonując pracę niezwiązaną bezpośrednio z główną częścią programu. Zwykle świadczą określone usługi na rzecz pozostałych wątków. Program może zakończyć się (w naturalny sposób), nawet jeśli istnieją działające wątki-demony. Program może działać, nawet gdy wszystkie wątki demony zakończą się, ale musi istnieć wtedy przynajmniej jeden wątek niedemoniczny.

Bycie demonem jest dziedziczne: każdy wątek utworzony przez demona jest demonem i na odwrót. Można to potem zmienić metodą `setDaemon(boolean)`.

Przykład

Wątki zwykłe klasy `Printer` drukują określoną liczbę znaków na konsoli, po czym kończą się. Klasa `Daemon` reprezentuje wątek demoniczny, który w pętli wypisuje liczbę na konsoli. Po zakończeniu się wszystkich wątków `Printer` on również zostanie zakończony.

```
class DaemonDemo {
    public static void main(String[] args){
        new Daemon();
        new Printer('*', 10000);
        new Printer('+', 10000);
        new Printer('#', 10000);
    }
}
class Daemon extends Thread {
    Daemon(){
        setPriority(Thread.MIN_PRIORITY);
        setDaemon(true);
        start();
    }
    public void run(){
        int alive = 0;
        while(true){
            System.out.print(alive++ + "\t");
            if(alive%10 == 0)
                System.out.println();
            // może być konieczne:
            yield();
        }
    }
}
```

```
class Printer extends Thread {
    char chr;
    int count;
    Printer(char c, int cnt){
        chr = c;
        count = cnt;
        setPriority(Thread.MAX_PRIORITY);
        setDaemon(false);
        start();
    }
    public void run(){
        for(int i = 0; i < count; i++){
            System.out.print(chr);
            if(i%80 == 0) {
                System.out.println();
                // może być przydatne:
                yield();
            }
        }
        System.out.println(
            "Printer(" + chr + ")-->DONE"
        );
    }
}
```

3.10 Komunikacja

Wątki mogą się ze sobą komunikować (przesyłać dane) przy pomocy potoków. Do czytania służą obiekty klasy `PipedReader` (znakowo) i `PipedInputStream` (bajtowo) a do wysyłania `PipedWriter` i `PipedOutputStream`. Obiekty czytające muszą być połączone z piszącymi poprzez argumenty konstruktorów.

Następny przykład pokazuje, jak przy pomocy takich strumieni wątki mogą przesyłać sobie liczby. Potoki `PipedInputStream` i `PipedOutputStream` zostały opakowane w strumienie do przesyłania typów pierwotnych: `DataInputStream` i `DataOutputStream`. Dzięki temu wątki mogą sobie przesyłać dane tych typów sposób niezależny od platformy.

```
import java.io.*;
class ProdConsPipe {
    public static void main(String[] args){
        PipedInputStream pis = new PipedInputStream();
        PipedOutputStream pos;
        try {
            pos = new PipedOutputStream(pis);
        }
        catch(IOException e){
            return;
        }
        DataInputStream dis = new DataInputStream(pis);
        DataOutputStream dos = new DataOutputStream(pos);
        new Thread(new PipedConsumer(dis)).start();
        new Thread(new PipedProducer(dos)).start();
    }
}
```

```
class PipedProducer
            implements Runnable {
    DataOutputStream out;
    PipedProducer(DataOutputStream dos){
        out = dos;
    }
    public void run(){
        int data = 0;
        while(true){
            try {
                out.writeInt(data++);
                Thread.sleep(500);
            }
            catch(InterruptedException e){
                return;
            }
            catch(IOException e){
                return;
            }
        }
    }
}
```

```
class PipedConsumer
            implements Runnable {
    DataInputStream in;
    PipedConsumer(DataInputStream dis){
        in = dis;
    }
    public void run(){
        int msg = 0;
        while(true){
            try {
                // blokujące wejście
                msg = in.readInt();
            }
            catch(IOException e){
                return;
            }
            // użycie msg
        }
    }
}
```

Konsument - obiekt klasy `PipedConsumer` - czyta liczbę ze strumienia danych. Jeśli jej tam nie ma, to czeka aż się pojawi. Jeśli jest ich więcej - czyta dopóki zarządca go nie wywłaszczy.

Producent - obiekt klasy `PipedProducer` - w pętli wpisuje liczbę do strumienia danych, a następnie zasypia na chwilę aby pokazać, że konsument musi czekać na pojawienie się kolejnej.

Zmieniając czas uśpienia można obserwować zachowanie zarządcy zadań - im będzie on krótszy tym większą liczbę obrotów pętli będzie mógł wykonać każdy wątek (bez wywłaszczenia).

Maksymalna ilość danych, które można umieścić w potoku bez blokowania zapisu, jest określona stałą `PipedInputStream.PIPE_SIZE`.

3.11 Grupy wątków

Wątki można łączyć w grupy. Służy do tego klasa `ThreadGroup`. Grupa wątków również może być elementem grupy. Zatem tworzą one drzewo, którego korzeniem jest grupa systemowa.

Podczas tworzenia wątek zostaje przypisany do jakiejś grupy (domyślnie systemowej) i nie może potem tego zmienić.

Grupowanie umożliwia zdefiniowanie wzajemnego dostępu wątków do siebie. Wątek może modyfikować inny, tylko jeśli należy do tej samej grupy lub grupy potomnej. Grupy umożliwiają również przeprowadzanie pewnych działań równocześnie na zestawie wątków np. ustalenie maksymalnego priorytetu dla członków grupy.

4 API - szczegóły

Oto zestawienie najważniejszych klas i metod związanych z wątkami. Wszystkie klasy i interfejsy są zdefinowane w pakiecie `java.lang`, który jest domyślnie importowany.

4.1 Interfejs `Runnable`

Zawiera tylko metodę `public void run()`. Jeśli w oparciu o obiekt klasy implementującej ten interfejs utworzymy nowy wątek, to będzie ona wywołana po jego uruchomieniu.

4.2 Klasa `Object`

```
void wait()
void wait(long)
void wait(long,
          int)
```

Trzy warianty metody, służącej do zawieszania wątku wewnątrz monitora identyfikowanego przez `this`. Argumenty określają maksymalny czas przez jaki wątek będzie oczekiwał wznowienia poprzez `notify()` lub `notifyAll()`. Po upływie tego czasu zostanie on normalnie wznowiony.

```
void notify()
void notifyAll()
```

Metody te służą do wznawiania wątków zawieszonych w monitorze po wywołaniu `wait()`. Pierwsza wznawia jeden, druga wszystkie wątki zawieszone na tej blokadzie.

4.3 Klasa `Thread`

4.3.1 Najważniejsze konstruktory

`Thread()`	Tworzą obiekt wątku nie uruchamiając go.
`Thread(Runnable r)`	Argument typu **`String`** jest nazwą wątku (domyślnie jest to
`Thread(String s)`	**`"Thread-"+`*n*** - gdzie n jest liczbą).
`Thread(ThreadGroup g, Runnable r)`	Argument typu **`ThreadGroup`** jest grupą, do której zostanie przyłączony wątek. Domyślnie jest to grupa systemowa.

4.3.2 Metody sterujące

`void start()`	Uruchamia wątek - wywołuje metodę **`run()`**.
`static void yield()`	Oddaje procesor innym wątkom.
`void join()`	Oczekuje na zakończenie innego wątku.
`void join(long)`	Jak wyżej, ale nie dłużej niż podany czas.
`static void sleep(long)`	Usypia wątek na podany (w milisekundach) czas.
`void interrupt()`	Ustawia flagę **`interrupted`** wątku na **`true`**. Jeśli był on uśpiony lub wstrzymany, to zostanie wznowiony i odbierze wyjątek **`InterruptedException`**.

4.3.3 Metody pomocnicze

`void setPriority(int)`	Ustalanie priorytetu.
`int getPriority()`	Pobieranie priorytetu.
`void setName(String)`	Ustalanie nazwy.
`String getName()`	Pobieranie nazwy.
`void setDaemon(boolean)`	Przejście pomiędzy wątkiem użytkownika i demonem.
`boolean isDaemon()`	Czy wątek jest demonem.
`boolean isAlive()`	Czy wątek może się jeszcze wykonywać.
`boolean isInterrupted()`	Dostarcza wartość flagi `interrupted`.
`ThreadGroup getThreadGroup()`	Pobiera grupę wątku.
`static Thread curentThread()`	Zwraca odniesienie do aktualnie wykonywanego wątku.

4.3.4 Metody zaniechane

Poniższych metod nie należy używać, gdyż mogą spowodować nieoczekiwane zachowanie (blokada) i dlatego zostały usunięte z nowszych wersji Javy. Znajdują się w klasie `Thread` ze względu na stare programy, które mogą ich jeszcze używać.

`void stop()`	Zatrzymuje wątek.
`void suspend()`	Zawiesza wątek.
`void resume()`	Wznawia wątek zawieszony przez `suspend()`.

4.4 Wyjątki

`IllegalMonitorStateException` jest zgłaszany przy próbie wywołania `wait()`, `notify()` lub `notifyAll()` poza blokiem synchronizowanym.

`InterruptedException` jest zgłaszany po wywołaniu `interrupt()` przez inny wątek na rzecz tego wątku (odbierającego wyjątek), podczas gdy oczekuje on na zajście jakiegoś zdarzenia (np. podczas uśpienia lub blokującego wejścia-wyjścia).

5 AWT i Swing

W AWT i Swingu obsługa wszystkich zdarzeń - interakcja z GUI - wykonywana jest tylko w specjalnie do tego przeznaczonym wątku zdarzeniowym (utworzonym przez VM). (dzięki czemu nie może dojść do przerwania obsługi jednego zdarzenia przez wystąpienie innego). Podstawowym wymaganiem biblioteki Swing jest, aby po zrealizowaniu komponentu (uczynieniu go zdolnym do wyświetlenia na skutek wywołania którejś z metod `setVisible()`, `show()` lub `pack()`) jego modyfikacje odbywały się wyłącznie w tym wątku. Co zrobić, jeśli zachodzi potrzeba zmodyfikowania GUI po jego uwidocznieniu, nie będąca skutkiem zajścia zdarzenia?

5.1 Kolejkowanie zadań

Należy skorzystać ze statycznych metod `invokeLater()` lub `invokeAndWait()` klasy `SwingUtilities` z pakietu `javax.swing`. Pozwalają one na wykonywanie zadań w wątku zdarzeniowym.

5.1.1 Kolejkowanie asynchroniczne

Funkcja `invokeLater(Runnable r)` umieszcza zadanie w wątku zdarzeniowym nie czekając na jego wykonanie. Może być wywołana z wątku zdarzeniowego czyli z funkcji obsługującej jakieś zdarzenie. Zadanie jest obiektem klasy implementującej interfejs `Runnable`. Zostanie ono wykonane po obsłużeniu wszystkich oczekujących zdarzeń.

```
Runnable doWork = new Runnable(){

    public void run(){
        // operacje na GUI
        // ale nie tylko
    }
};
SwingUtilities.invokeLater(doWork);
```

Powyższa metoda nie pozwala na poznanie rezultatu wykonania zadania. Co zrobić, jeśli chcemy z innego wątku pobrać np. zawartość pola tekstowego nie wykonując fizycznej akcji? Trzeba poczekać na zakończenie wykonywania zdarzenia, które to zrobi i zwróci wynik.

5.1.2 Kolejkowanie synchroniczne

Funkcja `SwingUtilities.invokeAndWait(Runnable r)` umieszcza zadanie w wątku zdarzeniowym czekając na jego wykonanie.

Podobnie jak poprzednio, zadanie jest obiektem klasy implementującej interfejs `Runnable` i zostanie ono wykonane po obsłużeniu wszystkich oczekujących zdarzeń. Różnica w stosunku do poprzedniej funkcji jest taka, że `invokeAndWait()` wstrzymuje wykonanie wątku, z którego została wywołana do czasu wykonania zadania. Konsekwencją jest zakaz wywoływania jej z wątku zdarzeniowego, bo może to zablokować obsługę innych zdarzeń. Jeśli jest to konieczne należy powołać do życia nowy wątek, który zaczeka na jej zakończenie - jak w przykładzie.

```
final Runnable doWork = new Runnable(){
    public void run(){
        // zrób coś
    }
};
new Thread(){
    new Runnable(){
        public void run(){
            try {
                SwingUtilities.invokeAndWait(doWork);
            }
            catch(InterruptedException e){
            }
            catch(InvocationTargetException e){
            }
            System.out.println(
                "zakończono wykonywanie zadania"
            );
        }
    }
}.start();
```

5.2 Generowanie zdarzeń

Innym sposobem, szczególnie użytecznym kiedy chodzi o wykonywanie powtarzalnych czynności w wątku zdarzeniowym jest użycie klasy `Timer` z pakietu `javax.swing`.

Obiekt klasy `javax.swing.Timer` generuje zdarzenie `ActionEvent` w ustalonych odstępach czasu, jeden raz lub wielokrotnie. Natępnie propaguje to zdarzenie do zarejestrowanych słuchaczy, powodując wywołanie u nich metody `actionPerformed()`.

Użycie generatora zdarzeń polega na sfabrykowaniu jego obiektu i wydaniu mu polecenia `start()`. Argumentami konstruktora jest częstość generowania zdarzenia i słuchacz zdarzenia `ActionEvent`. Można dodać większą liczbę słuchaczy metodą `addActionListener()`. Zatrzymanie generatora polega na wydaniu mu polecenia `stop()`.

```
class TimerUse {
    public static void main(String[] args){
        TimerUse tuse = new TimerUse();
        tuse.task.start();
        try {                               // w tym czasie
            Thread.sleep(10000);            // słuchacze
        }                                   // cyklicznie
        catch(Exception e){                 // reagują na
        }                                   // zdarzenia
        tuse.task.stop();
    }
    class Ticker implements ActionListener {
        int count = 0;
        int id = (int)(10*Math.random());
        public void actionPerformed(ActionEvent evt){
            System.out.println(id + "-->" + count++);
        }
    }
    final int delay = 500;
    Timer task = new Timer(delay, new Ticker());
    TimerUse(){
        task.addActionListener(new Ticker());
    }
}
```

6 Podsumowanie

Programowanie współbieżne jest trudne, ponieważ (w przeciwieństwie do programów sekwencyjnych) nigdy nie możemy mieć pewności, że nasz program będzie działał poprawnie niezależnie od danych wejściowych oraz kolejności aktywowania wątków lub procesów. Należy wykluczać (a przynajmniej minimalizować) ryzyko wystąpienia blokady i zagłodzenia, co jest zadaniem niebanalnym. Każde wykonanie programu współbieżnego (wielowątkowego) jest inne, ze względu na niemożliwe do przewidzenia momenty wznawiania i wstrzymywania wątków, oraz ich kolejność. Stawia to szczególne wymagania fazie testowania oprogramowania.

Pisanie wielowątkowych programów w Javie jest wygodne dzięki nowoczesnym i przejrzystym mechanizmom synchronizacji wątków. Jednak napisanie niezależnego od platformy (co jest istotą Javy), programu współbieżnego jest bardzo trudne. Przyczyną jest fakt, że maszyna wirtualna korzysta z modelu wielowątkowości dostarczanego przez system operacyjny, który w każdym przypadku jest inny. Zatem napisanie przenośnego, wielowątkowego programu w Javie wymaga przyjęcia dwu sprzecznych założeń:

- System **będzie** wywłaszczał nasz wątek.
- System **nie będzie** wywłaszczał naszego wątku (i w związku z tym sami musimy zadbać o przekazanie sterowania innym).

6.1 Podstawowa zasada programowania współbieżnego

UNIKAJ WSPÓŁBIEŻNOŚCI JEŚLI TO TYLKO MOŻLIWE.

6.2 Dobre rady

6.2.1 Nakazy

Jeśli powyższego nie da się uniknąć to:

- Synchronizuj wszystkie funkcje w klasie, które mają dostęp do jej składowych.
- Używaj jak najmniejszej liczby synchronizatorów.
- Nie zagnieżdżaj sekcji krytycznych (bloków, wywołań funkcji synchronizowanych), jeśli nie jesteś w 150% pewien konsekwencji tego, co robisz.
- Pamiętaj, że każdy wątek może być zawieszony w dowolnym miejscu kodu, nawet tam gdzie się tego nie spodziewasz (np. w trakcie przypisania `long l = cokolwiek;`).
- Używaj synchronizowanych metod i klas API jeśli mają być dostępne z wielu wątków. Synchronizowane wersje kolekcji można uzyskać metodami klasy `Collections` o nazwach `synchronizedXXX(XXX)`.

6.2.2 Zasady dobrego stylu

- Używaj minimalnej liczby wątków.
- Nie synchronizuj bloków lub metod wykonujących nieskończone pętle.
- Staraj się używać jak najmniejszej liczby dzielonych zasobów (zmiennych) jednocześnie. Po użyciu zwalniaj zasób tak szybko, jak to możliwe.
- Unikaj stosowania metod: `stop()`, `suspend()` i `resume()`, gdyż zwiększają one ryzyko wystąpienia blokady.
- Używaj kostrukcji dobrowolnie oddających sterowanie: `yield()`, `sleep()` jeśli chcesz zagwarantować przenośność programu.
- Nie ufaj przesadnie funkcji `yield()` - jej działanie zależy od systemu operacyjnego.
- Nie traktuj zbyt poważnie priorytetów.
- Nie przyjmuj założeń co do liczby procesorów w systemie.
- Testuj program na różnych maszynach: zmiana szybkości procesora może spowodować zadziwiające efekty.