Concurrency

Spis treści

```
Jak jeden wątek to za mało, to można zrobić kilka rzeczy na raz.
```

Współbieżność / Concurrency - teoria

Po co to wszystko jest i o co w tym chodzi?

Czym jest współbieżność w kontekście programowania? Generalnie to piękne słowo oznacza wykonywanie wielu rzeczy w tym samym czasie i tak samo w naszym przypadku: *o współbieżności mówimy, gdy komputer wykonuje równocześnie więcej niż jedną instrukcję*. I tutaj zachodzi pytanie: *po co w ogóle się to stosuje i po co mi ta wiedza*? Jesteś już przecież w stanie napisać program, robiący to, co chcesz i po co dodawać do tego jakąś współbieżność…

Otóż jest ona istotna z punktu widzenia efektywności. Wyobraź sobie, że gotujesz zupę: wstawiłeś/-aś już wodę do zagotowania i pozostały ci warzywa do umycia i pocięcia. Oczywiście nie czekasz na zagotowanie się wody (skończenie pierwszej instrukcji), ale przygotowujesz warzywa od razu. Zyskujesz czas, a sam proces gotowania staje się bardziej wydajny.

Tak samo jest w przypadku komputerów - nie muszą one czekać na wykonanie konkretnej operacji, a

wykonują wiele rzeczy "naraz". Komputer nie musi czekać na wykonanie jednej instrukcji (która może być czasochłonna: na przykład dostęp do wiekowych i wypieranych już dysków talerzowych: HDD), a w międzyczasie może zająć się czymś innym.

Przekładając to na twoją naukę: po przyswojeniu tej części materiału będziesz w stanie pisać programy, w których komputer (a dokładnie mówiąc procesor) będzie wykonywał wybrane fragmenty kodu w tym samym czasie.

Twórcy Javy zrobili wiele w temacie współbieżności i przygotowali gotowe rozwiązania dla programistów. Jest to m.in. *Concurrency API* (dla dokładności: to API zostało dodane do Javy w wersji nr 5) i tym będziemy się tutaj zajmować. API to jest potężne i zawiera naprawdę wiele klas: także zaczynamy.

Współbieżność istnieje w Javie od zawsze, a od czasu powstania języka wszystko jest cały czas rozwijane.

Definicja pojęcia **concurrency** z Wikipedii wygląda tak:

> (...) the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome (...)

W odniesieniu do Javy zagadnienie współbieżności pokrywa kilka powiązanych ze sobą pojęć:

- Multithreading (wielowątkowość)
- Concurrency (współbieżność)
- Parallelism (równoległość)

Zanim o nich porozmawiamy, najpierw przypomnijmy sobie podstawowe terminy takie jak **proces** i **wątek**:

Czym jest wątek? Czym jest proces? Czym się różnią?

Powyższe słowa każdy rozumie intuicyjnie. Każdy przecież ma do czynienia z jakimiś procesami w życiu, a wiele wątków ma np. każdy film. I co do zasady te intuicyjne definicje, jakie znasz, są dość blisko tych w kontekście programowania w Javie, ale warto je sobie na początek omówić, żeby mieć pewność, że każdy myśli dokładnie o tym samym.

Wątek (ang. ***thread***) to najmniejsza (w kontekście współbieżności) część programu, która może zostać wykonana. Po angielsku: ***unit of execution***, jest to również klasa w Javie, ale o tym za chwilę! Co to oznacza? Otóż w danym czasie, w ramach jednego wątku może się wykonywać tylko jedna instrukcja.

Z tego miejsca łatwo już wyjść do tego: *czym jest **proces**?* Otóż nazwiemy tak grupę wątków, które współpracują ze sobą w jednym środowisku. Wracając do naszego porównania z filmem, można powiedzieć, że historia w takim filmie to właśnie odpowiednik procesu. Historia jest jedna i można opowiedzieć ją od początku do końca, musi składać się z chociaż jednego wątku, a zazwyczaj ma ich wiele...

Natomiast w wątku filmowym, którego naszym odpowiednikiem jest **thread**, na raz dzieje się tylko jedna rzecz. Natomiast wszystkie wątki przenikają się (w naszym kontekście: współpracują) i tworzą

jeden film (**proces**).

We wcześniejszych notatkach pojawiały się już stwierdzenia **proces** oraz **wątek** i były one wyjaśniane w taki sposób:

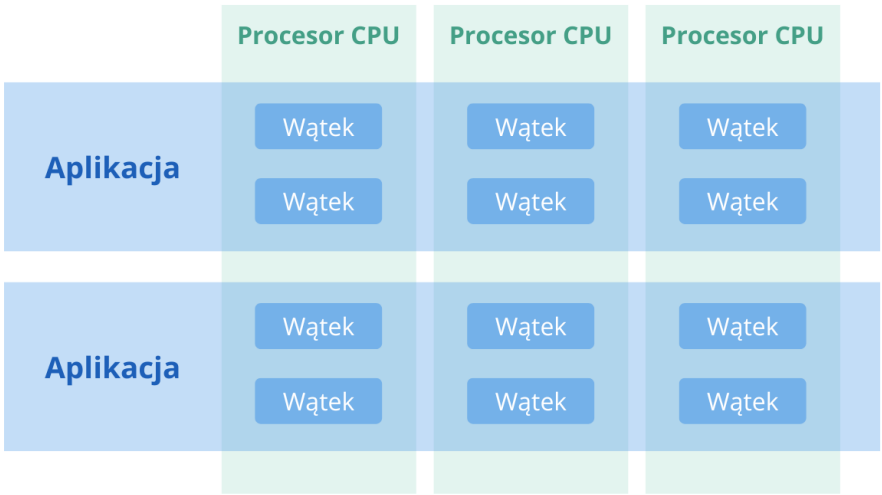
- **Proces** — mówi się, że jest egzemplarzem wykonywanego programu. Gdy odpalamy naszą aplikację, może ona uruchomić jeden proces w naszym systemie operacyjnym, ale może też uruchomić takich procesów wiele. W systemach operacyjnych każdy proces ma swoje unikalne PID (*process identifier*), który jest jednoznacznym identyfikatorem procesu. Gdy uruchamiamy program, system operacyjny takiemu procesowi przydziela zasoby, m.in. pamięć RAM, czas procesora.
- **Wątek** — W ramach jednego **procesu** możemy stworzyć jeden lub więcej **wątków**, które będą wykonywały jakieś części programu. Wątki współdzielą zasoby zarezerwowane dla procesu, oprócz czasu procesora. Ten jest przydzielany indywidualnie do każdego wątku.

Mam nadzieję, że po takim porównaniu wiesz już, czym jest **wątek** i **proces** i czym się od siebie różnią. Tutaj od razu zachodzi pytanie: *czy można stworzyć proces tylko z jednym wątkiem?* Odpowiedź brzmi: *oczywiście, że tak*. W zasadzie to już właśnie takie programy pisałeś/-aś do tej pory, czyli wykonujące się w jednym wątku. Czas zobaczyć jak działać na wielu wątkach.

Wiedząc już o tym, należy napisać, że jak uruchamiamy aplikację w Javie przez wywołanie metody `main()`, automatycznie tworzymy jeden wątek i na nim operujemy. Wątki możemy też tworzyć sami i wykonywać pewne czynności współbieżnie.

I na koniec odrobina nomenklatury: o procesie możemy powiedzieć, że jest *jedno-* bądź *wielowątkowy* (*single-* vs *multi-threaded process*). I tak tutaj bez niespodzianek: znaczenie jest trywialne.

Multithreading (wielowątkowość)



Obraz 1. Multithreading (wielowątkowość)

Multithreading — oznacza, że w aplikacji może istnieć więcej niż jeden wątek, gdzie każdy z nich odpowiada za wykonywanie różnych części kodu w tym samym czasie.

Jakie są zalety współbieżności?

OK, powiesz: *wiem już, czym jest ta cała współbieżność, ale co to znaczy, że jest efektywna? Jakie są jej konkretne zalety?*

I tutaj wracamy do przykładu z zupą. Pierwszą konkretną zaletą jest:

Lepsze wykorzystanie posiadanych zasobów

Jak wiesz sercem komputera jest procesor, z angielskiego **CPU** (***c**entral **p**rocessing **u**nit* - nazwa już wiele mówi w kontekście informacji, które właśnie pozyskałeś/-aś o procesie). **CPU** wykonuje program, jednak w zależności od programu, zazwyczaj w wielu sytuacjach musi sięgnąć po inne zasoby (jak np. wczytanie pliku z dysku, czyli Java NIO). Zazwyczaj takie operacje są kosztowne czasowo (wspomnieliśmy już prędkość działania dysków twardych, szczególnie tych starych: talerzowych). Oczekiwanie na wczytanie pliku z dysku to idealny moment dla procesora (który w tym czasie tylko czeka na odpowiedź) na zajęcie się czymś innym, niezależnym. Tym samym czas wykonania całego programu ulega skróceniu. Tak samo, jak w przykładzie z zupą i krojeniem warzyw w trakcie gotowania wody.

Zwiększenie responsywności

Na pewno masz w życiu wspomnienie związane z oczekiwaniem załadowanie jakiejś gry. Większość gier obecnie robi to lepiej i udostępnia graczowi jakąś mini-grę (albo chociażby pasek postępu) w trakcie wczytywania z dysku pełnej lokacji zamiast pokazywania przez długie minuty statycznego ekranu. Być może pamiętasz grę "*Mass Effect*" - tam duże lokacje przedzielone były windami, gdzie główni bohaterowie spędzali czas na pogaduchach (tym dłuższych im wolniejszy dysk twardy posiadał gracz).

Jeśli nie jesteś graczem: nic strasznego. Wyobraź sobie sytuację, że pobierasz z internetu jakiś duży plik przy użyciu jedno-wątkowego programu i... **NIC SIĘ NIE DZIEJE**, a przynajmniej ty tak stwierdzasz, bo nie widzisz, żeby cokolwiek się działo na ekranie. Tak naprawdę program pobiera się, ale przez to, że procesor wykonuje jeden wątek, w którym posiada wyraźną instrukcję o pobraniu pliku - jest zajęty i nie może aktualizować widoku, na który patrzysz. Nic nie możesz kliknąć - wszystko wygląda na zawieszone. Procesor wykonuje program i czeka na pobranie... Już pewnie widzisz jak tutaj wykorzystanie wielu wątków (jeden do obsługi interfejsu użytkownika, a drugi do zadań backendowych) podniosłoby ogólną jakość programu.

Efektywna obsługa wielu użytkowników

Do wytłumaczenia powyższego punktu użyłem przykładu aplikacji desktopowej (gra komputerowa), to jest używanej zazwyczaj na komputerze przez pojedynczą osobę. Do wytłumaczenia tego punktu znacznie lepiej przysłuży się aplikacja webowa, tj. taka do której zazwyczaj dostajesz się przez przeglądarkę, owiedzmy aplikacja twojego banku.

Załóżmy, że klikasz button: "wszystkie produkty" w GUI swojego banku. Oznacza to wysłanie zapytania do serwerów bankowych. Bank je otrzymuje i kompletuje wszystkie informacje o kartach, saldach, ubezpieczeniach itd. jakie posiadasz (najprawdopodobniej twój bank sięga teraz do swoich baz danych i łączy się przez sieć z wieloma odległymi systemami), komponuje odpowiedź i wysyła do twojej przeglądarki w celu jej wyświetlenia na ekranie. Brzmi prosto. Co się zatem stanie, gdy w międzyczasie ktoś inny wykona takie samo zapytanie? Albo inne zapytanie: np. o specjalnie spersonalizowane oferty

kredytu? Tak jest, ten inny ktoś będzie musiał poczekać, aż serwer banku wykona wszystkie opisane wyżej operacje i wyśle odpowiedź dotyczącą twoich produktów z powrotem do Ciebie. Niby nie brzmi źle, przecież to tylko chwila. Zdecydowanie gorzej zrobi się, gdy zamiast prostych operacji mówimy o takich, które mogą trwać dobrych kilka sekund. Albo, gdy zwiększymy liczbę użytkowników do powiedzmy: kilkuset…

Mam nadzieję, że widzisz już, że takie oprogramowanie nie sprawdziło by się najlepiej. Dzięki wielowątkowości procesor może dzielić czas na wielu użytkowników. Czy twoje zapytanie o wszystkie produktay zajmie więcej czasu? Owszem, serwer będzie zajęty przyjmowaniem i obsługą nowych zapytań, ale dzięki temu kolejne zapytania innych ludzi zajmą znacznie mniej czasu bo nie będą musiały czekać w kolejce.

Do tematyki wielowątkowości w kontekście aplikacji webowych i w kontekście serwerów przejdziemy w warsztacie, w którym po raz pierwszy będziemy przygotowywali aplikację uruchamianą na serwerze.

A jakie są wady?

Po zapoznaniu się z powyższym pewnie myślisz, że współbieżność to absolutne must-have i powinno się tego używać wszędzie. Mimo wszystko pamiętaj, że to są również koszty, to znaczy:

Context switch

Procesor nie umie przeskakiwać między wątkami bez extra kosztu. Dokładnie tak jak człowiek potrzebuje chwili, gdy przechodzi od rąbania drewna do mycia naczyń, potrzeba chwili na przeorganizowanie się. Taką zmianę między wątkami wykonywaną przez procesor nazywamy właśnie "*context switch*". W trakcie context switcha procesor musi zapisać informacje ze starego wątku i załadować dane z nowego wątku, by móc na nich pracować. Są to operacje, które naturalnie nie występują w jednowątkowej aplikacji.

Zasobożerność

Wspomniałem właśnie o context switch'u, jak się okazuje, kosztuje on nie tylko czas, ale również zasoby. W końcu gdzieś trzeba zapisać ten context wątku. Nic nie ma za darmo…

Poziom skomplikowania

Na koniec dochodzimy do skomplikowania kodu. Po pierwsze współbieżność jest kolejnym tematem, który trzeba opanować. Po drugie wprowadza również do twojej nauki programowania nowe problemy, których musisz nauczyć się unikać, ale również rozpoznawać ich symptomy i naprawiać. A niestety rozpoznawanie i odtwarzanie błędów, z którymi będziesz mógł/-ą się tutaj spotkać nie zawsze będzie najprostsze. Do tych problemów jeszcze zdecydowanie wrócimy.

Współbieżność (*concurrency*) vs równoległość (*parallel execution*)

Brzmi podobnie prawda? W zasadzie czy w ogóle jest jakaś różnica, czy te dwa pojęcia znaczą dokładnie

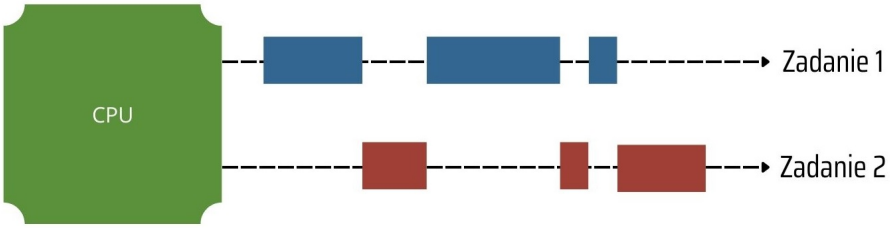
to samo (i tu myślisz, że pewnie nie skoro w ogóle zaczynam o tym gadać) - otóż nie.

Concurrency (współbieżność)

Concurrency — oznacza, że aplikacja jest w stanie robić postępy w więcej niż jednym zadaniu jednocześnie. Czy faktycznie jest tak, że dzieje się to w dokładnie tym samym momencie, zależy od tego, ile rdzeni ma procesor na maszynie, gdzie program jest uruchamiany. Sama współbieżność oznacza, że więcej niż jedno zadanie jest realizowane w tym samym czasie, w trakcie działania naszej aplikacji.

Wypadałoby podać jakiś przykład z życia. Jesteś w stanie jednocześnie jeść i mówić? Dokładnie w tym samym czasie? Mama zawsze mówiła, żeby tak nie robić, bo się udławisz ☺. Można powiedzieć, że jesteś w stanie realizować na raz konsumowanie posiłku i prowadzenie konwersacji, ale dokładnie w tym samym czasie przełykać posiłku i wymawiać słowa nie możemy. Czyli musisz przełknąć, potem możesz coś powiedzieć, potem możesz przełknąć, potem możesz coś powiedzieć. Nie da się tego zrobić jednocześnie. Realizujemy jakiś proces/zadanie (czyli jedzenie i rozmawianie) w tym samym czasie i idą one do przodu (robimy w nich progress), czyli jedzenia jest coraz mniej i zmierzamy w tej rozmowie do końca, ale nie jesteśmy w stanie tego robić jednocześnie.

Zatem wykonujemy kilka zadań w tym samym czasie, ale niekoniecznie są one wykonywane równocześnie. Spójrz na poniższą grafikę, żeby lepiej poukładać to sobie w głowie.

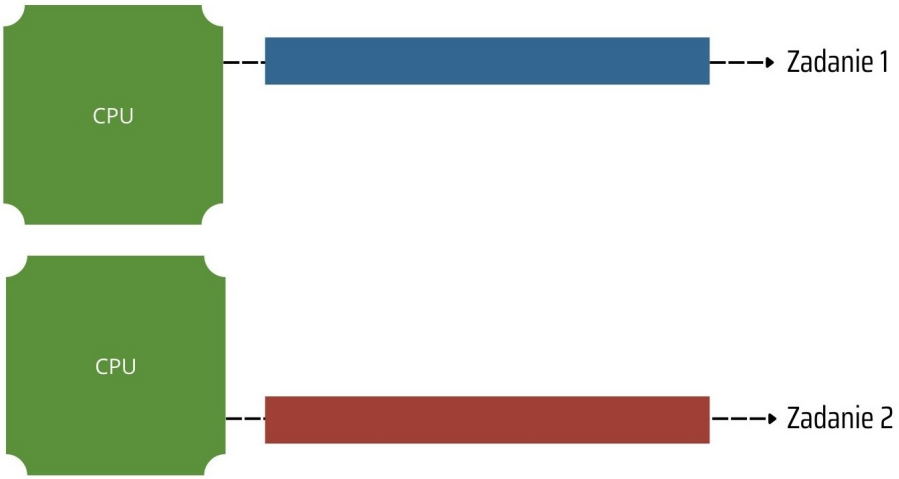


Obraz 2. Concurrency

Parallel execution (równoległość/równoległe wykonanie)

O współbieżności już sporo powiedzieliśmy: procesor przełącza się pomiędzy wątkami, wykonując wiele wątków *"jednocześnie"*. Pracuje nad wieloma wątkami w ramach jednego procesu. Skupmy się zatem na równoległym wykonaniu.

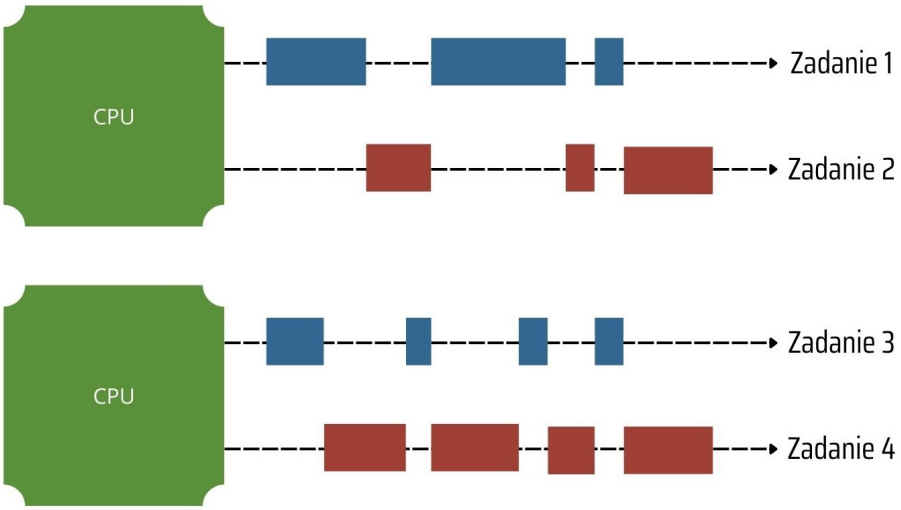
Załóżmy, że mamy do dyspozycji komputer z 2 procesorami. Pierwsze co przychodzi do głowy, by poprawić efektywność działania programu to właśnie wykonanie programu na kilku procesorach jednocześnie. Spójrz na poniższą grafikę:



Obraz 3. Parallel execution

Opisane powyżej działanie określa się właśnie mianem równoległego wykonania (ang. *parallel execution*).

Oczywiście takie wykonanie możemy łączyć ze współbieżnością: w takim wypadku program byłby wykonywany przez 2 procesory jednocześnie, a w ramach każdego z pracujących procesorów następowałoby przełączanie pomiędzy wątkami. Spójrz na poniższą grafikę:



Obraz 4. Parallel Concurrent execution

W tym przypadku możemy mówić o *Parallel Concurrent execution*, jak to przetłumaczyć na polski? Równoległe współbieżne wykonanie.. chyba...

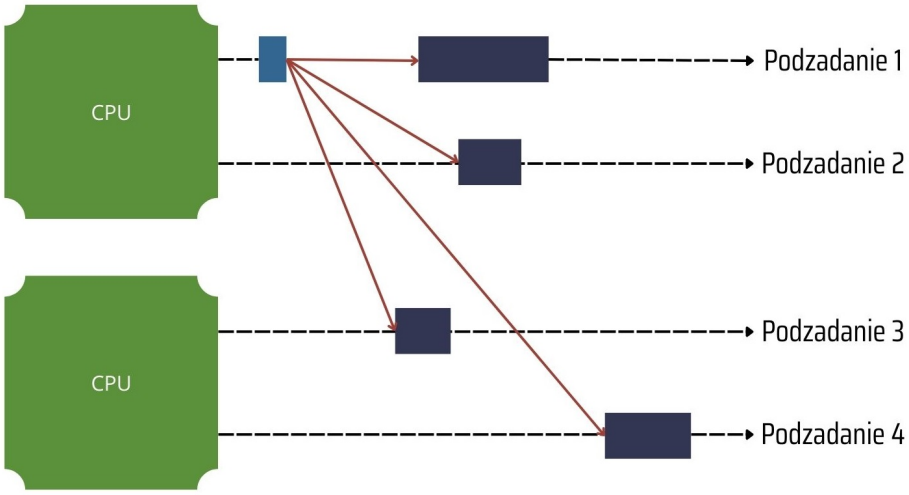
Parallelism

Parallelism — oznacza, że aplikacja jest w stanie podzielić to, co ma zostać wykonane na mniejsze fragmenty i faktycznie wykonać je równolegle. Aby tak mogło się stać, wymagane jest posiadanie więcej niż jednego CPU, wtedy każde z nich jest w stanie wykonywać jakieś zadanie jednocześnie.

Czym różni się to od *Parallel Concurrent execution*? Otóż na pełną równoległość musimy patrzeć szerzej - zakłada ona podział danego zadania otrzymanego przez procesor na zestaw mniejszych pod-zadań i ich pełne równoległe wykonanie. Można zatem powiedzieć, że równoległe wykonanie jest elementem równoległości jako takiej.

Znowu przykład z życia, to by mogło zadziałać, jakbyśmy mieli obok siebie 2 usta. Jedno by jadło, a drugie rozmawiało w tym samym czasie. Wtedy akurat mogłoby się to dziać w tym samym czasie.

Dla lepszego zrozumienia, spójrz na poniższą grafikę, która przedstawia podział zadania na mniejsze pod-zadania i równoległe wykonanie.



Obraz 5. Parallelism

OK, była to całkiem potężna piguła teorii na początek, możemy więc ruszać do pierwszych implementacji...

Praktyka

Możemy teraz przejść do zabawy w kodzie.

Tworzenie i rozpoczynanie wątków (threads)

Jeśli do tej pory uważałeś/-aś to możesz już się domyślać, że tak naprawdę umiesz już tworzyć wątek... Tworzysz go za każdym razem pisząc metodę `main()` swojego programu. Dobrze więc, spytasz w takim razie jak utworzyć drugi wątek i uczynić swoją aplikację wielowątkową? Nic prostszego, wykorzystamy do tego klasę `Thread` i jej konstruktor.

```
public static void main(String[] args){
    Thread myVeryFirstThread = new Thread();
}
```

I proszę, mamy już "własny" wątek, niestety nie jest on zbyt użyteczny. Po prawdzie to nawet się nie uruchamia. Zobaczmy, co możemy z tym zrobić. Na początek wydzielmy go do osobnej klasy:

```
public class MyVeryOwnThreadImplementation extends Thread {

    @Override
    public void run(){
        System.out.println("Siemanko MyVeryOwnThreadImplementation");
    }
}
```

Teraz możemy go odpalić:

```
public static void main(String[] args){
    Thread myVeryFirstThread = new MyVeryOwnThreadImplementation();
    myVeryFirstThread.start(); // uruchamiamy wyżej zdefiniowany wątek
    System.out.println("Siemanko main");
}
```

Gdy uruchomisz powyższy kod - nie będzie niespodzianek. Zobaczysz dwie linijki tekstu w konsoli. To, co będzie wyjątkowe i nowe w porównaniu z twoimi poprzednimi programami, to fakt, że ich wypisanie odbędzie się w osobnych wątkach. Oznacza to również że program wywołując metodę `start()` nie będzie czekał na jej wykonanie tylko od razu pójdzie dalej. Nie jesteśmy więc w stanie powiedzieć, który z powyższych tekstów wypisze się pierwszy.

Poza rozszerzeniem klasy `Thread` istnieje jeszcze druga metoda stworzenia własnego wątku: poprzez implementację interfejsu `Runnable`. Spróbujmy:

```
public class RunnableThreadImplementation implements Runnable {

    public void run(){
        System.out.println("Siema po raz drugi z runnable'a");
    }
}
```

I uruchomienie:

```
public static void main(String[] args){
    Runnable myRunnable = new RunnableThreadImplementation();
    Thread thread = new Thread(myRunnable); ①
    thread.start();
}
```

① Używamy konstruktora, który jako argument przyjmuje interfejs `Runnable`.

Nie różni się to zbyt od rozszerzenia klasy `Thread`, za to w tym przypadku możemy uprościć sprawę używając lambdy, np. o tak:

```
public static void main(String[] args){
    Thread thread = new Thread(() -> System.out.println("Siema po raz trzeci z lambdy"));
    thread.start();
}
```

Teraz całość zmieściła się w jednej klasie. Mechanizm jest ten sam: program wywołuje metodę `run()` w klasie rozszerzającej `Thread` lub w twojej implementacji `Runnable`. W przypadku gdy wywołasz `start()` bezpośrednio w klasie `Thread` - to jego metoda `run()` zostanie wywołana: robi ona dokładnie zupełnie nic.

W tym momencie pewnie się zastanawiasz: po co mam odpalać wątek przez metodę `start()` skoro jej głównym zadaniem jest wywołanie mojej metody `run()`, przecież mogę wywołać metodę `run()` sam bezpośrednio na stworzonym wątku:

```
public static void main(String[] args){
    Thread thread = new Thread(() -> System.out.println("Siema po raz trzeci z lambdy"));
    thread.run(); // tak nie rób
}
```

Niby wszystko ok, "panie to się kompiluje", nawet zobaczysz `println()` w konsoli. Natomiast wszystko wykona się w jednym wątku, drugi wątek nigdy nie zostanie utworzony. Musisz o tym pamiętać.

Chciałoby się napisać, że kwestia wyboru: rozszerzenie klasy `Thread` lub `Runnable` to kwestia osobistej preferencji i tak w zasadzie jest. Jednak generalnie, raczej polecam używanie `Runnable`. Dlaczego? Jak dobrze pamiętasz, w Javie nie ma wielodziedziczenia (problem diamentu!), dlatego implementując interfejs, nie blokujemy sobie możliwości dziedziczenia po jakiejś innej klasie. Daje to po prostu nam samym większą elastyczność. No i nie zapominajmy o możliwości użycia lambdy - wygodne i nadal przejrzyste, jeśli nasza implementacja metody `run()` nie jest za długa.

I dalej patrząc na to z bardziej architektonicznej perspektywy, jest lepiej, bo oddzielasz "wykonawcę" (czyli `Thread`) od instrukcji do wykonania (która znajduje się w `Runnable`) - gdy rozszerzasz wątek, wszystko trzymasz w jednej klasie.

Nazwy wątków i ich wyświetlanie

Tworzone wątki mają swoje nazwy. Za każdym razem, gdy tworzysz wątek, jest do niego przypisywana nazwa w takim stylu: *thread-0*, *thread-1*, *thread-2*, ... i tak dalej.

Później możesz te nazwy wątków odczytywać, co może być przydatne, chociażby przy debuggingu. Do tego celu użyjesz, niespodzianka, metody `getName()` z klasy `Thread`.

```
System.out.println(Thread.currentThread().getName());
```

Zwróć uwagę, najpierw używam metody statycznej `currentThread()` by zdobyć wątek właściwy dla danej linijki kodu, a potem już mogę wyciągnąć jego nazwę (w Stringu). Całość elegancko przekazuję do konsoli. Oczywiście możesz użyć metody `getName()` bezpośrednio na zmiennej z klasy `Thread` o ile istnieje ona w kontekście, w którym się aktualnie znajdujesz (lub inaczej: w której znajduje się linijka kodu, którą chcesz napisać).

Własne nazywanie wątków

Powiesz: *ok, ale co gdy chcę nazwać swój wątek powiedzmy: "thread-siemanko" żeby było mi jeszcze łatwiej je odróżnić?* Nic prostszego, klasa `Thread` udostępnia nam taką możliwość przy pomocy konstruktorów:

```
Thread firstThread = new Thread("thread-siemanko"); ①
Thread secondThread = new Thread(myRunnable, "thread-siemanko-z-runnabla") ②
```

① Tutaj możesz użyć oczywiście również swojej klasy rozszerzającej klasę `Thread`.

② Jest również wersja konstruktora z implementacją `Runnable` i nazwą w Stringu.

To nie koniec - masz też możliwość zmiany nazwy wątku już po jego utworzeniu. Klasa udostępnia w tym celu standardowy setter:

```
secondThread.setName("thread-siemanko-by-setter");
```

Oczywiście, gdy będziesz nazywał/-a swoje wątki w trakcie pracy, zasady są takie same jak przy nazywaniu klas, metod czy zmiennych: wybieraj nazwy, które mają znaczenie. Im prościej będzie programiście z ulicy zrozumieć, za co dany wątek odpowiada, tym lepiej. W celach debbugingowych możesz używać nazw jakich chcesz ☺.

Pauzowanie wątków

Kolejną przydatną możliwością (i znowu szczególnie w debuggingu) jest "usypianie" wątku, a mówiąc wprost, pauzowanie na określony czas.

Służy do tego metoda `sleep()` przyjmująca jako argument liczbę milisekund (w `longu`) przez jaką wątek będzie "zawieszony", zanim przejdzie do wykonywania kolejnych linii kodu. Robimy to w poniższy sposób:

```
try {
    Thread.sleep(5000L); ①
} catch (InterruptedException ex) {
    // obsłużenie wyjątku
}
```

① Wątek zamiera na 5 sekund.

Metoda ta deklaruje rzucanie wyjątku, który musimy obsłużyć (jest to *checked exception*), może do niego dojść, gdy jakiś inny wątek wezwie metodę `interrupt()` na naszym wątku.

Problemy wielowątkowości

OK, wiesz już jak tworzyć wątki i przekazywać im instrukcje do wykonania. Wszystko wydaje się (póki co) łatwe i przyjemne. Jednak w akapicie o wadach wspomniałem o nowych problemach, jakie pojawiają się przy tworzeniu wielowątkowego kodu. Każdy programista programujący z użyciem tych technik powinien (musi!) zdawać sobie z nich sprawę. Dodatkową motywacją niech będzie fakt, że pytania rekrutacyjne z tego zakresu są popularne i trudne. Także, starając się o pracę w zawodzie, zdecydowanie możesz spodziewać się pytań o **race condition** i **deadlock** (spokojnie, omówimy dokładnie te pojęcia).

Java a zarządzanie pamięcią (memory model)

Zanim pójdziemy dalej z tematem, musimy się na chwilę zatrzymać i pogadać o czymś pozornie zupełnie niezwiązanym. Spokojnie, przyda ci się to zdecydowanie i to już zaraz. Mianowicie jak wygląda zarządzanie pamięcia w Javie (i w ogóle jak wygląda pamięć komputera).

Temat ten był już poruszany na etapie Bootcampu, natomiast teraz wrócimy do tego tematu w kontekście wielu wątków.

Powiedzieliśmy już sobie, że wykonywaniem wątków (czyli samym wykonywaniem kodu) zajmuje się procesor. Nie ma on jednak wiedzy o wartościach przypisanych do zmiennych. Ty czytając kod i widząc nazwę jakiejś zmiennej, musisz poscrollować sobie i znaleźć miejsce w kodzie, gdzie dana zmienna miała przypisaną wartość - i już wiesz.

Procesor również "nie wie" jaka wartość stoi za daną zmienną. Musi ją pobrać z RAMu. RAM (eng. *random-access memory*), jest bardzo szybką pamięcia, jaką dysponuje komputer, ale jest pamięcią "podręczną" tzn. jest czyszczona w momencie restartu komputera i stanowi uzupełnienie dla twardego dysku (dużo wolniejszego, ale trzymającego dane na stałe, do tego znacznie tańszego, jeśli chodzi o pojemność).

W trakcie działania programu wirtualna maszyna Javy (JVM) zapisuje wszystko, co tworzysz i na czym pracujesz w kodzie aplikacji w RAMie komputera. Każde stworzenie zmiennej, przypisanie jej wartości, jej zmiana czy odczyt oznacza interakcję procesora z RAMem... no prawie zawsze: żeby cały proces jeszcze usprawnić, pomiędzy RAMem a procesorem znajduje się **cache procesora**. Jest to pamięć jeszcze szybsza niż RAM, gdzie procesor lokuje najczęściej używane wartości po to, żeby zaoszczędzić czas i energię potrzebną do komunikacji z RAMem. Może tam na przykład trafić zmienna, której wartość jest częścią warunku w pętli while.

Generalnie wszystkie typy pamięci można w uproszczeniu opisać tak:

- **Cache procesora** - najszybszy, nadroższy, najmniejszy (bo najdroższy). Super-podręczna pamięć do przechowywania aktualnie używanych danych,
- **RAM** - średnio szybki (dużo szybszy niż dysk twardy, ale wolniejszy od cache), średnio drogi, średnio duży. W RAMie informacje mogą być trzymane tak długo, jak komputer jest włączony, reset komputera równa się wyczyszczeniu całego RAMu. W kontekście Javy: trzymane są tam wszystkie dane w trakcie trwania programu (jeśli coś jest trzymane w cache - to kopia danych z RAMu),
- **Twardy dysk** - najwolniejszy, najtańszy, największy. Chociaż dyski SSD są o wiele szybsze od dysków HDD to nadal dużo wolniejsze od RAMu. Na dysku twardym trzymasz to, co chcesz, żeby przetrwało dłużej niż nagłe wyłączenie prądu. Czyli np. wpisy w lokalnej bazie danych (albo nie w kontekście programowania: stare faktury i save'y do gier).

Java a zarządzanie pamięcią ciąg dalszy - shared resources

Szczególnie ciekawy w kontekście wielowątkowości jest RAM, ponieważ nie wszystko, co Java przechowuje w RAM, jest przechowywane w ten sam sposób. Mamy tam:

- Stosy (*stack*) - każdy wątek ma swój niezależny stack (po polsku stos lub sterta, ale prawdopodobnie

częściej usłyszysz *stack*). Ile odpalonych wątków tyle stacków.

- Sterta (*Heap*) - jeden wspólny worek dla wszystkich wątków w programie, "pamięć współdzielona".

Tutaj trzeba od razu zauważyć, że z punktu widzenia hardware'u jest to wszystko jedno - wszystko jest trzymane w RAMie i ładowane do wewnętrznej pamięci procesora (innymi słowy *cache'u*) w ten sam sposób. Podział na stack i heap jest stricte software'owy i stricte javowy.

Java w jasno określony sposób (w zależności od typu przechowywanych danych) część informacji trzyma w stackach (czytaj: są niezależne od siebie i przypisane do określonych wątków), a część w heapie (czytaj: współdzielone między wszystkimi wątkami, właśnie takich resourcach powiemy, że są "shared").

Co więc trafia gdzie? Otóż:

1. Zmienne lokalne (zmienne zadeklarowane wewnątrz metody).
2. Parametry metody (przyjęte z "zewnątrz", w ramach tej metody).
3. Parametry w klauzalach `catch` (czyli wszystkie exceptiony jakie "łapiesz"/obsługujesz).

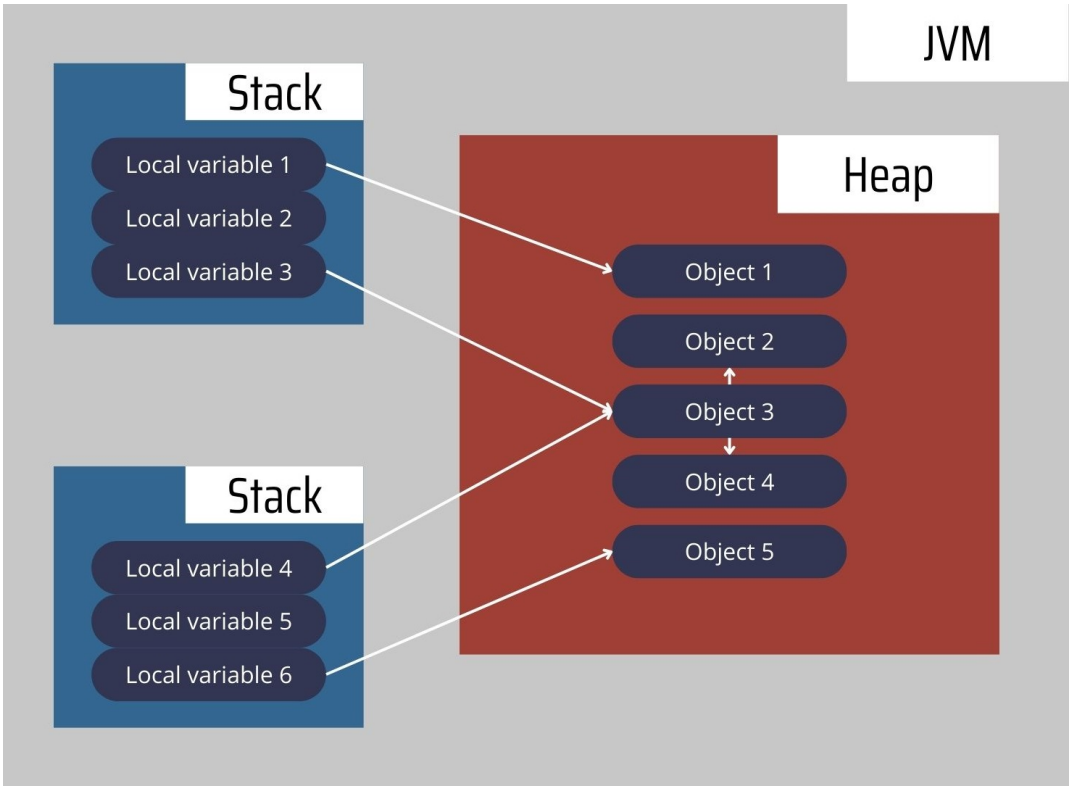
Wymienione wyżej zawsze trafiają do stacku! Oznacza to, że np. każdy wątek ma "swoje" zmienne lokalne i pozostałe wątki nie mają do nich dostępu.

Natomiast np.:

1. Pola klasy
2. Zmienne statyczne
3. Elementy dodawane do list

trafiają do heapa, czyli są "shared", wspólne dla wszystkich wątków.

Na podstawie tego, co zostało napisane, spójrz na poniższą grafikę, żeby móc sobie to wszystko poukładać w głowie.



Obraz 6. Model pamięci Java

OK, może już widzisz, co tu się święci... I w tym momencie możemy przejść dalej, czyli do problemów, jakie przynosi nam wielowątkowość aplikacji w Javie.

Race condition

Głównym problemem, jaki pojawia się w wielowątkowych aplikacjach jest *race condition*.

Brzmi tajemniczo, ale geneza problemu jest bardzo prosta: gdy mamy zasób (np. zmienna, ale również tabela w bazie danych), do której dostęp ma kilka wątków (czyli, jak już sobie powiedzieliśmy: taki który znajduje się w heapie aplikacji), a od kolejności ich dostępu zależy wynik całej operacji, to może pojawić się problem.

Lub jeszcze inaczej na przykładzie: mamy 1 zmienną i 2 wątki, które operują na tej zmiennej: jeden nadpisuje jej wartość, a drugi odczytuje. I teraz wartość odczytana przez drugi wątek może się różnić. Jak to możliwe? Otóż wiesz, że wątki działają niezależnie - robią swoje nie oglądając się na siebie. W zależności czy pierwszy zdąży nadpisać wartość zmiennej przed drugim wątkiem, wątek numer 2 może pobrać wartość starą bądź nową - zaktualizowaną. Jak widzisz mamy tutaj taki "wyścig" tzn. *race*.

Race condition można podzielić na dwa główne typy, czyli:

1. *Read-modify-write*
2. *Check-then-act*

Read-modify-write

Z pierwszym przypadkiem możemy mieć do czynienia, gdy np. piszemy kod generujący numery do faktur. Być może wiesz, że numer faktury powinien być unikalny (a może nie musi? nie jestem

księgowym, ale dla dobra przykładu załóżmy, że tak musi być ☺), powiedzmy, że napisaliśmy taki kod do ich generowania:

```
public class InvoiceNameGenerator {

    private static final String INVOICE_NAME_PATTERN = "FV_";

    private int lastInvoiceNumber = 0; ①

    public String generateNewInvoiceName() {
        lastInvoiceNumber = lastInvoiceNumber + 1; ②
        return INVOICE_NAME_PATTERN + lastInvoiceNumber;
    }
}
```

① Prymitywy nie mogą mieć `null`a, a 0 to dla inta wartość domyślna, czyli "= 0" jest nadmiarowe i jego zapis należy do osobistej preferencji.

② Tutaj mamy: read, modify i write, stąd nazwa scenariusza.

Kod na pierwszy rzut oka wygląda ok, ale to, co może się stać w przypadku wielowątkowej aplikacji to sytuacja, gdy dwa wątki będą wykonywać metodę `generateNewInvoiceName()` "w tym samym czasie". Może dojść do sytuacji, w której drugi wątek pobierze wartość `lastInvoiceNumber`, zanim pierwszy zapisze jego zaktualizowaną wartość. Skutkować to będzie dwoma fakturami o tym samym numerze. Sytuacja, której zdecydowanie chcemy uniknąć.

Check-then-act

Drugi przypadek *Check-then-act* wiąże się z `if`em, w którego ciele (części kodu, do której wejdziemy w przypadku gdy wynikiem warunku będzie `true`) będzie kod, który zmieni wynik tego `if`a. Dość trudno wyobrazić sobie taki przypadek na podstawie samego opisu, więc spójrz na ten prosty kod:

```
if (list.contains("test")) { // check
    list.remove("test"); // then act
}
```

Scenariusz jest inny, ale analogiczny do poprzedniego przykładu, wątki wchodzą w warunek w tym samym czasie i usunięcie elementu listu następuje po tym, jak drugi wątek już sprawdził ten warunek. Następnie drugi wątek próbuje usunąć już usunięty wcześniej element - co skutkuje naszym ulubionym `NullPointerException`.

Skoro wiesz już, czym jest problem *race condition* to możemy w takim razie przejść do metod, jak się przed nim zabezpieczyć. Pierwsze co może przyjść do głowy to zagwarantowanie, żeby kod, który jest na to podatny (taki właśnie kod nazywamy po angielsku **critical section**, czyli kod, który operuje na współdzielonych zasobach z heapa: *shared resources*) powinien być wykonywany *atomowo* tzn. tylko przez jeden wątek jednocześnie. Jak to zapewnić? Cierpliwości, pociągnijmy temat dalej...

Odporność na race condition czyli bycie thread-safe.

O właśnie takim kodzie, odpornym na *race condition* (czy inaczej: taki w którym race condition **NIE MA PRAWA** zaistnieć) w przypadku użycia go przez wiele wątków powiemy, że jest **Thread-safe**. Samo

thread-safe można zdefiniować na wiele sposobów (mniej lub bardziej się rozpisać), natomiast wszystko sprowadza się do jednego: kod thread-safe może, być bezpiecznie wykonany przez wiele wątków (jak sama nazwa wskazuje). Bezpiecznie z punktu widzenia działania aplikacji i przede wszystkim wyniku jej działania.

Pierwszym krokiem do posiadania kodu, który jest **thread-safe**, jest używanie zasobów, z jakich korzystasz w sposób, który nie pozwoli na *race condition*. O jakich zasobach mówimy?

I tutaj wracamy do akapitu o pamięci w Javie. **Wszystko, co jest przechowywane w stacku jest thread-safe, wszystko co jest przechowywane w heapie nie jest thread-safe** (jest wspólne dla wielu wątków). Dla utrwalenia i rozwinięcia tematu:

- zmienne lokalne są zawsze thread-safe. Jeśli jakaś zmienna została zadeklarowana w metodzie to znaczy, że nigdy nie będzie współdzielona między wątkami (każdy wątek trzyma ją osobno)! I choć zawsze jest to prawda, to jednak w przypadku typów obiektowych możesz się zaskoczyć efektem finalnym... owszem, zmienne nie są współdzielone, ale ich wartości tak. Trzeba pamiętać, że sama zmienna zawiera tylko referencję do obiektu... i tak jak same referencje są trzymane osobno przez wątki, tak obiekty siedzą już sobie w heapie i są współdzielone. Trzeba na to uważać. Dlatego nie przekazuj nigdy lokalnej referencji do obiektu "na zewnątrz" metody, bo tym samym udostępnisz ją do użytku wielu wątkom. W przypadku prymitów oczywiście - zmienne lokalne są thread safe (serio zawsze).
- pola w klasie nie są thread-safe, co widać na przykładzie z fakturami. Pola, podobnie jak całe obiekty - są trzymane w heapie.

No dobra, to jak w końcu jak zabezpieczyć się przed tym *race condtion* konkretnie w kodzie?

Odporność na race condition - Synchronized

OK, powiedzieliśmy już sobie, że jedną z metod zabezpieczenia przed race condition jest zapewnienie, że fragmenty naszego kodu, które stanowią *critical section* będą wykonywane tylko przez jeden wątek naraz. Na szczęście jest to bardzo proste, bo java dostarcza nam w tym celu już gotowe narzędzia. I właśnie takim narzędziem jest słowo kluczowe (element syntaxu) `synchronized`.

Weźmy nasz przykład z generowaniem numerów faktur i odrobinę go zmodyfikujmy:

```
public class InvoiceNameGenerator {

    private static final String INVOICE_NAME_PATTERN = "FV_";

    private int lastInvoiceNumber = 0;

    public synchronized String generateNewInvoiceName() {
        lastInvoiceNumber = lastInvoiceNumber + 1;
        return INVOICE_NAME_PATTERN + lastInvoiceNumber;
    }
}
```

I... załatwione! Kod już jest **thread-safe**, wspomniany problem z *race condition* (ten sam numer dla dwóch faktur nie może już wystąpić!). Pierwszy wątek wywołując metodę `generateNewInvoiceName()`, blokuje jej wywołanie dla wszystkich innych wątków na tak długo aż z niej nie wyjdzie, czyli w tym

przypadku nie zwróci Stringa. Bajka!

I tutaj dla jasności warto sobie jeszcze wspomnieć o dwóch rzeczach. Oczywiście takie blokowanie działa tylko w ramach jednej instancji klasy. Gdy w osobnych wątkach stworzymy sobie osobne InvoiceNameGeneratory, to nadal ten kawałek kodu będzie mógł być wywoływany dla tych dwóch obiektów współbieżnie. Technicznie oczywiście żaden *race condition* nie zachodzi: mamy 2 zupełnie niezależne generatory z dwoma zupełnie niezależnymi polami lastInvoiceNumber (race condition jak pamiętasz, dotyczy tylko wspólnych zasobów). Patrząc na to funkcjonalnie, to jednak efekt jest ten sam: powtarzające się numery faktur, więc w tym konkretnym przypadku warto zadbać, żeby nasz InvoiceNameGenerator miał tylko jedną instancję w całym programie (czy to dobry design to osobna kwestia, ale ten przykład dobrze i prosto pokazuje ten problem) ALBO pole lastInvoiceNumber możemy przerobić na zmienną statyczną:

```
public class InvoiceNameGenerator {

    private static final String INVOICE_NAME_PATTERN = "FV_";

    private static int lastInvoiceNumber = 0;

    public static synchronized String generateNewInvoiceName() {
        lastInvoiceNumber = lastInvoiceNumber + 1;
        return INVOICE_NAME_PATTERN + lastInvoiceNumber;
    }
}
```

Jak pamiętasz: pola i metody statyczne nie są związane z żadną instancją a bezpośrednio z klasą, której dotyczą. Teraz nieważne ile będziemy mieli instancji naszego InvoiceNameGeneratora, metoda generateNewInvoiceName() nie będzie wykonywać się współbieżnie na innych wątkach.

Drugą rzeczą, o której warto wspomnieć to fakt, że jeśli w klasie będziemy mieć kilka metod oznaczonych przez synchronized, to wejście wątku do jakiejkolwiek metody zablokuje innym wątkom możliwość wywołania WSZYSTKICH tych metod. Mały szczegół a DOŚĆ istotny.

Poza oznaczaniem metod słowem synchronized jest jeszcze możliwość oznaczania mniejszych bloków kodu, wewnątrz metod:

```
public String generateNewInvoiceName() {
    System.out.println("Ten print będzie wykonywany współbieżnie");
    synchronized(this) {
        System.out.println("... a ten nie");
        lastInvoiceNumber = lastInvoiceNumber + 1;
        return INVOICE_NAME_PATTERN + lastInvoiceNumber;
    }
}
```

Zwróć uwagę, że tutaj obok słowa kluczowego mamy nawiasy i parametr: tak jakbyśmy wywoływali metodę. Obiekt, który przekazujemy w nawiasie nazywamy **monitor object**. Brzmi groźnie, ale znaczy ni mniej, ni więcej, że wykonanie kodu w klamrach jest zablokowane dla innych wątków, ALE W RAMACH DANEJ INSTANCJI KLASY.

Dla pewności i podsumowania rozpiszmy to sobie wszystko:

Znana nam już bardzo dobrze klasa InvoiceNameGenerator

```
public class InvoiceNameGenerator {
    private static final String INVOICE_NAME_PATTERN = "FV_";

    private int lastInvoiceNumber = 0;

    public String generateNewInvoiceName() {
        synchronized(this){
            lastInvoiceNumber = lastInvoiceNumber + 1;
            return INVOICE_NAME_PATTERN + lastInvoiceNumber;
        }
    }
}
```

Klasa InvoiceThread

```
import java.util.stream.IntStream;

public class InvoiceThread extends Thread { ①

    InvoiceNameGenerator invoiceNameGenerator;

    public InvoiceThread(InvoiceNameGenerator invoiceNameGenerator) { ②
        this.invoiceNameGenerator = invoiceNameGenerator;
    }

    @Override
    public void run() { ③
        IntStream.range(0, 3)
            .forEach(iteration -> System.out.println(invoiceNameGenerator.generateNewInvoiceName())); ④
    }
}
```

① Nasza implementacja wątku.

② Tworzymy własny konstruktor, żeby być pewnym, że wątek nie powstanie bez przypisanego generatora faktur.

③ Wygenerujmy trochee nazw dla faktur.

④ Print do konsoli, każdy wątek niech wygeneruje trochę tych numerów.

Klasa Main

```
public class Main {
    public static void main (String[] args) {
        InvoiceNameGenerator invoiceNameGenerator = new InvoiceNameGenerator();

        Thread firstThread = new InvoiceThread(invoiceNameGenerator);
        Thread secondThread = new InvoiceThread(invoiceNameGenerator); ①

        firstThread.start();
        secondThread.start();
    }
}
```

① Zwróć uwagę, że mamy jedną instancję naszego generatora - jest to zasób współdzielony, 2 wątki

będą generować faktury, ale nazwy się nie powtórzą.

Odpal powyższy kod i zobacz wynik, w konsoli powinieneś/-aś zobaczyć wygenerowane 6 nazw dla faktur - wszystkie unikatowe od `FV_1` do `FV_6` (mało profesjonalnie, ale już ustaliliśmy, że to nie kurs księgowy), a potem poeksperymentuj na bazie tego wszystkiego, co tutaj omówiliśmy: stwórz dla drugiego wątku osobny obiekt `InvoiceNameGenerator` (`synchronized` nie działa pomiędzy instancjami, zobaczysz więc co prawda 6 numerów, ale powtarzających się), przenoś oznaczenie `synchronized` i dodawaj printy tam gdzie chcesz.

Podsumowując, `synchronized` gwarantuje nam, że zmiany, które wykonuje jeden wątek w danym bloku kodu, będą już dostępne w pamięci (w heapie) w momencie, gdy następny wątek będzie je pobierać. *Race condition* zażegnane. Za jaką cenę? Oczywiście wydajności, na szczęście narzut jest w tym przypadku wart swojej ceny.

Odporność na race condition - immutable

Wiesz już, że kod wykonywany przez wiele wątków powinien być **thread-safe** (żeby jego działanie było przewidywalne). Wiesz także, że *race condition* może zajść na współdzielonych przez wątki zasobach, w sytuacji, gdy te wątki jednocześnie zapisują i odczytują dany zasób.

Z tego wynika, że najprostszym sposobem na tworzenie kodu **thread-safe** jest zabranie możliwości update'owania tego zasobu. Brzmi jak zbyt proste rozwiązanie skomplikowanego problemu, ale jest ono jak najbardziej poprawne i często wykorzystywane!

O obiektach, które nie mają możliwości update'u powiemy że są immutowalne (ang. **immutable**) - raz stworzone, nie mogą już być nigdy zmienione. **Obiekty immutable zawsze są thread-safe**. Do tego naprawdę łatwo napisać immutowalną klasę, wystarczy pamiętać o paru rzeczach.

1. ŻADNYCH SETTERÓW.
2. Wszystkie pola ustawiane w konstruktorze.
3. Wszystkie pola oczywiście prywatne.
4. Klasa oznaczona jako final (nie możemy jej rozszerzyć).
5. Wszystkie pola finalne (ustawione raz i na zawsze).

I już! Np. taka klasa jest immutowalna, po jej stworzeniu nie ma już opcji zmian stanu:

Klasa MyImmutableClass

```
public final class MyImmutableClass {
    private final String fieldOne;
    private final long fieldTwo;

    public MyImmutableClass(String fieldOne, long fieldTwo) {
        this.fieldOne = fieldOne;
        this.fieldTwo = fieldTwo;
    }

    public String getFieldOne() {
        return fieldOne;
    }
```

```
    public long getFieldTwo() {
        return fieldTwo;
    }
}
```

W tym momencie zapala ci się lampka, no dobra, ale co gdy MUSZĘ zmodyfikować obiekt. Powiedzmy, chcę zupdate'ować jakieś jedno pole w obiekcie, czy tak się da programować w ogóle? Nic prostszego: tworzysz nowy obiekt tej samej klasy, kopiujesz do niego wszystkie wartości ze starego obiektu, z wyjątkiem tego jednego pola...

Widzę teraz twoją minę i tak: jest to mniej wygodne, bardziej zasobożerne i generalnie mniej wydajne od zwykłego settera. Na szczęście mamy coraz szybsze komputery, więc możesz spać spokojnie ☺.

Podsumowując, jest to naprawdę dobra metoda radzenia sobie z problemami wielowątkowości i do tego (tu uwaga!) szeroko wykorzytywana we współczesnych projektach w branży. Najlepsze jest to, że umiesz już na tym etapie stworzyć obiekt immutable przy wykorzystaniu jednej adnotacji:

Klasa MyImmutableClass z wykorzystaniem Lomboka

```
@Value ①
@With ②
@Builder ③
public final class MyImmutableClass {
    String fieldOne;
    long fieldTwo;
}
```

① Dzięki zastosowaniu adnotacji `@Value`, stworzysz obiekt immutable.

② Adnotacja `@With` pozwoli Ci na tworzenie nowego obiektu immutable ze zmodyfikowanym tylko jednym polem.

③ Adnotacja `@Builder` pozwoli Ci w wygodny sposób stworzyć instancję klasy. Adnotacja ta nie jest konieczna, żeby stworzyć obiekt immutable, natomiast jest ona często używana z poprzednimi dla wygody.

Odporność na race condition - Lock

Czym jest "lock"? W tym przypadku również jedno słowo wyraża więcej niż tysiąc. Jest to narzędzie / technika w Javie, dzięki której możemy "blokować" kawałki kodu do użycia przez jeden wątek.

Przypomnij sobie jak rozmawialiśmy o *database lock*. Tutaj działa to na analogicznej zasadzie.

Zastosowanie jest takie samo jak w przypadku `synchronized` - **ustalenie wyłącznego dostępu dla konkretnego wątku do współdzielonego zasobu**. Technicznie jest to interfejs Java, wprowadzony do API Javy w wersji 5. Mamy już gotowych kilka implementacji tego interfejsu do użycia, jednak nic nie stoi na przeszkodzie, żebyśmy sami zaimplementowali ten interfejs. Same locki bazują na znanym Ci już mechanizmie `synchronized`, nie jest to więc coś całkowicie nowego.

Zaraz... skoro zastosowanie i mechanizm jest ten sam co w przypadku `synchronized` to po co w ogóle `lock`? Otóż jest to rozwiązanie bardziej zaawansowane, bardziej elastyczne. Przyjrzyjmy mu się bliżej.

Spójrzmy na najprostsze zastosowanie, użyjmy starego przykładu z `synchronized` i lekko go zmodyfikujmy:

Klasa InvoiceNameGenerator

```
public class InvoiceNameGenerator {
    private static final String INVOICE_NAME_PATTERN = "FV_";

    private Lock lockExample = new HypoteticalLockImplementation(); ①
    private int lastInvoiceNumber = 0;

    public String generateNewInvoiceName() {
        lockExample.lock();
        int calculatedLastInvoiceNumber = lastInvoiceNumber + 1;
        lockExample.unlock();
        return INVOICE_NAME_PATTERN + calculatedLastInvoiceNumber;
    }
}
```

① Tworzymy obiekt locka w momencie tworzenia klasy.

Jak widzisz blok `synchronized` zastąpiliśmy użyciem `locka` - co do zasady nic się nie zmieniło. Natomiast sam `return` znajduje się już poza "bezpiecznym" (jednowątkowym) fragmentem kodu, a jeszcze nadal w *critical section*. Przeskoczyliśmy ten problem w prosty sposób: deklarując nową zmienną lokalną `calculatedLastInvoiceNumber`. Zmienne lokalne, jak pamiętasz, zawsze są **thread-safe**. Nie ma już więc ryzyka, które moglibyśmy mieć w przypadku pozostawienia jednej zmiennej globalnej: aktualizację `lastInvoiceNumber` w czasie gdy inny wątek skleja sobie *return statement*.

A samo użycie locka? Jak widzisz: banalne... użyliśmy dwóch metod: `lock()` oraz `unlock()`, myślę, że w tym przypadku zastosowanie jest oczywiste.

Warto tutaj jeszcze zwrócić uwagę na kawałek kodu, który ujrzymy jak tylko wejdziemy na stronę interfesju `Lock` w oficjalnej dokumentacji Javy (kod pochodzi z dokumentacji:

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Twórcy zwracają w ten sposób uwagę na zagrożenia, jakie niesie ze sobą użycie locka. W przeciwieństwie do użycia bloku "synchronized", gdzie zdjęcie blokady z zasobu mamy już "w pakiecie" tak w powyższym przypadku sami musimy pamiętać o zdjęciu locka. Kod, który może rzucać wyjątkami musi być przed nimi zabezpieczony, a *lock* zawsze zdjęty - inaczej narażamy się na "więczną" blokadę danego zasobu.

Na koniec tematu locków: mamy teraz świetną sposobność, by rozwinąć temat kwestii *reentrance*. Rozwinąć, gdyż już wspominałem o tym mechanizmie podczas dyskusji nt. *synchronized*. Otóż, jak pamiętasz, jeśli mamy w jednej klasie kilka metod `synchronized` to wejście w jedną z nich przez wybrany wątek blokuje dla niego wszystkie inne metody `synchronized` w danej klasie.

Samą nazwę *reentrance* najłatwiej wytłumaczyć na prostym przykładzie: Gdy wątek wejdzie już w synchronizowane na danej instancji fragmenty kodu, wtedy ten wątek (i tylko ten wątek) może wejść stamtąd (re-enter) w dowolny inny fragment kodu już zsynchronizowany na tej instancji (poza metodami `synchronized` pamiętaj również o `synchronized(this)`! - reentrance działa zawsze na *monitor object*).

I tak jak `synchronized` zapewnia *reentrance* od razu, tak przy implementacji własnego locka musimy sami to zapewnić. Na szczęście Java API udostępnia nam gotową implementację interfejsu `Lock` wraz z reentrance. Jest to klasa `ReentrantLock`. Jak stwierdza sama dokumentacja Javy, jest to klasa o takim samym zachowaniu jak mechanizm `synchronized`, natomiast dająca rozszerzone możliwości. Jednym z nich jest zapewnienie bycia "fair" - na razie pewnie brzmi tajemniczo, ale zaraz o tym porozmawiamy więcej.

Odporność na race condition - ThreadLocal

Kolejnym przepisem na pisanie kodu *thread-safe* jest stosowanie klasy `ThreadLocal`. Klasa ta jest bardzo przemyślna. Zakładając sytuację, że mamy jeden obiekt `ThreadLocal`, na którym pracuje wiele wątków, wartości, które dane wątki przypisują do tej zmiennej, są trzymane osobno: per wątek. Nigdy nie dojdzie tutaj do żadnego *race condition*, bo pomimo jednej zmiennej w kodzie, wartości są zawsze odseparowane.

Działanie klasy `ThreadLocal` jest bardzo ładnie wyjaśnione w dokumentacji, którą cytujemy poniżej.

> This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).
>
> For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes ThreadId.get() and remains unchanged on subsequent calls.
>
> (...)
>
> Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

Wygodnie nie? Spójrzmy na przykład pokazujący użycie tej klasy.

Powiedzmy, że chcemy mieć wartość typu `Integer`, która będzie powiązana z określonym wątkiem. W tym celu tworzymy instancję `ThreadLocal`.

```
ThreadLocal<Integer> threadLocal = new ThreadLocal<>();
```

Gdy chcemy użyć tej wartości typu `Integer` powiązanej z określonym wątkiem, operujemy na metodach

`set()` oraz `get()`. Inaczej mówiąc, możemy sobie wyobrazić, że `ThreadLocal` przechowuje dane wewnątrz mapy z wątkiem jako kluczem. Czyli, gdy wywołamy metodę `get()` na zmiennej `threadLocal`, otrzymamy wartość `Integer` dedykowaną dla wątku.

W poniższym przykładzie będziemy operować na obiekcie klasy `MyObject`.

Klasa MyObject

```
class MyObject {
    private Integer counter;

    public MyObject(Integer counter) {
        this.counter = counter;
    }

    public Integer getCounter() {
        return counter;
    }

    public MyObject withCounter(String who, Integer counter) {
        System.out.printf("Changing value by: [%s] to value: [%s]%n", who, counter);
        this.counter = counter;
        return this;
    }

    @Override
    public String toString() {
        return "MyObject{counter=%s}".formatted(counter);
    }
}
```

Klasa CounterRunnable

```
class CounterRunnable implements Runnable {

    private final ThreadLocal<MyObject> threadLocal = new ThreadLocal<>(); ①
    private final MyObject myObject = new MyObject(0);

    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        justWait(threadName); ②
        if (threadLocal.get() != null) { ③
            MyObject previous = threadLocal.get();
            threadLocal.set(previous.withCounter(threadName, previous.getCounter() + 1));
        } else {
            threadLocal.set(new MyObject(0));
        }

        myObject.withCounter(threadName, myObject.getCounter() + 1); ④

        System.out.printf("name: [%s] - threadLocal.get(): [%s]%n", threadName, threadLocal.get());
        System.out.printf("name: [%s] - counter: [%s]%n", threadName, myObject);
    }

    private void justWait(String who) {
        try {
            int toWait = new Random().nextInt(1000);
```

```
            System.out.printf("Thread: [%s] waiting for: [%s]%n", who, toWait);
            Thread.sleep(toWait);
        } catch (InterruptedException ex) {
            System.err.println("ERROR: " + ex.getMessage());
        }
    }
}
```

① Celem `ThreadLocal` jest tworzenie oddzielnych instancji obiektu dla każdego wątku. Każdy wątek będzie miał własną kopię zmiennej `ThreadLocal` i powiązanego obiektu. Dlatego zaleca się utworzenie zmiennej `ThreadLocal` osobno dla każdego wątku, zamiast udostępniania jej między wątkami. Udostępnianie zmiennej `ThreadLocal` między wątkami zniweczyłoby cel lokalnego przechowywania wątków. Cała idea `ThreadLocal` polega na dostarczaniu danych lub obiektów specyficznych dla wątku, umożliwiając każdemu wątkowi posiadanie własnej, izolowanej instancji bez ingerencji w inne wątki.

② Czekamy - niech wszystkie wątki zaktualizują swoje wartości. `Random` jest tutaj dodany, żeby każdy wątek wykonał swoje zadanie w innym momencie w czasie.

③ W tym fragmencie modyfikujemy obiekt przy wykorzystaniu `ThreadLocal`.

④ W tym fragmencie modyfikujemy obiekt bez wykorzystania `ThreadLocal`.

Klasa ThreadLocalCounterExample

```
public class ThreadLocalCounterExample {

   public static void main(String[] args) {
      CounterRunnable commonObject = new CounterRunnable();

      Thread t1 = new Thread(commonObject);
      Thread t2 = new Thread(commonObject);
      Thread t3 = new Thread(commonObject);
      Thread t4 = new Thread(commonObject);

      t1.start();
      t2.start();
      t3.start();
      t4.start();
   }
}
```

Na ekranie zostanie wydrukowane coś w stylu:

```
Thread: [Thread-0] waiting for: [177]
Thread: [Thread-2] waiting for: [832]
Thread: [Thread-1] waiting for: [437]
Thread: [Thread-3] waiting for: [207]
Changing value by: [Thread-0] to value: [1]
name: [Thread-0] - threadLocal.get(): [MyObject{counter=0}]
name: [Thread-0] - counter: [MyObject{counter=1}]
Changing value by: [Thread-3] to value: [2]
name: [Thread-3] - threadLocal.get(): [MyObject{counter=0}]
name: [Thread-3] - counter: [MyObject{counter=2}]
Changing value by: [Thread-1] to value: [3]
name: [Thread-1] - threadLocal.get(): [MyObject{counter=0}]
name: [Thread-1] - counter: [MyObject{counter=3}]
```

```
Changing value by: [Thread-2] to value: [4]
name: [Thread-2] - threadLocal.get(): [MyObject{counter=0}]
name: [Thread-2] - counter: [MyObject{counter=4}]
```

Oznacza to, że zmienna opakowana (można patrzeć na `ThreadLocal` również jak na opakowanie izolujące wartości między wątkami) w `ThreadLocal` pracowała na kopii niezależnej od innych wątków. Z kolei, zmienna `myObject` była współdzielona przez wszystkie wątki.

Kiedy tworzysz zmienną `ThreadLocal`, każdy wątek uzyskujący do niej dostęp będzie miał własną niezależną kopię. Wszelkie zmiany dokonane w zmiennej `ThreadLocal` lub przechowywanym przez nią obiekcie będą miały wpływ tylko na określony wątek uzyskujący do niej dostęp, zapewniając bezpieczeństwo wątków i eliminując potrzebę jawnej synchronizacji.

Najlepszą praktyką jest utworzenie oddzielnej zmiennej `ThreadLocal` dla każdego wątku, gdy chcesz udostępnić dane lub obiekty specyficzne dla wątku. Pozwala to na izolację i niezależność między wątkami, zapewniając prawidłowe i wydajne wykorzystanie mechanizmu `ThreadLocal`.

Można powiedzieć, że jest to kolejny sposób na osiągnięcie *thread safety* bez tworzenia klas *immutable*.

Gdy przejdziemy do omówienia środowiska WEB, poruszone zostanie zagadnienie: *jak ma się ThreadLocal do transakcji?*

Sporo powiedzieliśmy sobie o *race condition* i jak mu zapobiegać, przejdźmy do dalszych problemów związanych z wielowątkowością:

1. Deadlock
2. Livelock
3. Starvation

Pomówmy sobie trochę o każdym z nim (i obiecuję, że nie będzie więcej problemów, przynajmniej w tym temacie).

Deadlock

Co to *deadlock*? Proste, określenie sytuacji, gdy 2 (albo więcej) wątki się nawzajem "blokują". Najprostszy przykład (oczywiście zaistnieje tylko i wyłącznie w przypadku gdy kod jest napisany w bardzo konkretny, poniższy sposób):

Mamy 2 wątki, dwa zasoby (A i B, mogą być zmienne, może być dostęp do bazy, dowolny obiekt itd.). Wątki uruchamiają współbieżnie dwa różne zsynchronizowane metody / bloki kodu.

1. Pierwszy wątek uruchamia zsynchronizowany blok kodu blokujący dostęp do zasobu A.
2. Drugi wątek uruchamia zsynchronizowany blok kodu blokujący dostęp do zasobu B.
3. Pierwszy wątek dochodzi do momentu w swoim bloku kodu (zsynchronizowanym!), w którym potrzebuje dostępu do zasobu B (dostępu zsynchronizowanego, a więc musi zablokować ten zasób...). Ten jest jednak zablokowany przez drugi wątek... Procesor w tym wątku musi więc czekać na wątek numer 2.
4. Drugi wątek przechodzi do momentu w swoim zsynchronizowanym bloku kodu, w którym

potrzebuje dostępu do zasobu A, który jest zablokowany przez pierwszy wątek - musi więc czekać.

I to jest właśnie *deadlock* - dwa wątki czekają na siebie nawzajem, przy czym przez wzajemne zależności będą czekać w nieskończoność. Oczywiście mogą istnieć bardziej skomplikowane deadlocki, angażujące znacznie większą liczbę wątków - jednak co do zasady działają na tej samej zasadzie co ten prosty przykład.

W takim przypadku trzeba uważać, jak został napisany kod: jeśli mamy jakąś zsynchronizowaną metodę zawsze trzeba sprawdzić, czy nie wymaga ona dostępu do zasobów potencjalnie blokowanych przez inne wątki. Jeśli tak to już nam się powinno zapalić lampka: trzeba sprawdzić te pozostałe wątki. I to nie tylko czy nie wymagają dostępu do zasobów blokowanych w pierwotnym wątku, ale też, czy nie zachodzi bardziej skomplikowana relacja, np. oczekują jeszcze innych zasobów blokowanych przez inne wątki, które z kolei mogą czekać na ten nasz pierwotny wątek - już widzisz, że takich sytuacji należy unikać.

Spytasz zatem: *JAK?!* 3 złote zasady, bo Java sama zezwala na takie sytuacje i w żaden sposób im nie zapobiega, ani nawet nie ostrzega użytkownika: *panie to się skompiluje, nie ma problemu*:

1. Jak pewnie zauważyłeś/-aś na tym prostym przykładzie powyżej, obydwa wątki chciały zablokować dla siebie te same zasoby, jednak zrobiły to w odwrotnej kolejności. Gdyby obydwa wątki zaczynały od tego samego zasobu, do sytuacji nigdy by nie doszło. Czyli pierwza metoda to odpowiednia (ta sama) kolejność blokowania zasobów przez różne wątki.
2. **Timeout**, czyli nazwa, która pojawiała się już wcześniej. Zamiast czekać w nieskończoność, wątek powinien mieć ustalony czas, po którym odpuści czekanie i się "zrollbackuje" tzn. zwolni zasoby, które zablokował, poczeka i spróbuje jeszcze raz.
3. **Deadlock detection**, czyli ostatnia metoda, która zakłada, że każde zablokowanie zasobu zostaje odnotowane w globalnym "rejestrze" (np. w mapie odnotowującej kto, (czyli który wątek) co dokładnie zablokował). Jeśli *thread* nie może założyć locka, to zagląda do rejestru w poszukiwaniu, kto blokuje potrzebny mu zasób. Co z tym robi dalej to już kwestia uznaniowa: może np. postąpić podobnie jak w przypadku *timeoutu* tzn. zwolnić zasoby, poczekać i spróbować jeszcze raz. Jeśli program jest bardzo skomplikowany, to w takiej sytuacji może dochodzić często, więc po ten sposób sięgnałbym, tylko jeśli dwa pierwsze z jakiegoś powodu są poza zasięgiem. Warto by również określić mniej priorytetowe wątki - te które się "rollbackują", tak żeby te naprawdę te istotne nigdy nie musiały na nic czekać. Jednak i na to rozwiązanie trzeba uważać (o czym za chwilę przy okazji *starvation*).

Czasami o wiele łatwiej jest zrozumieć kod niż słowa pisane, spójrz zatem na poniższy przykład, który skończy się *deadlockiem*:

Klasa DeadlockExample

```
class DeadlockExample {

    public static final Object lock1 = new Object();
    public static final Object lock2 = new Object();

    public static void main(String[] args) {
        Member1 member1 = new Member1();
        Member2 member2 = new Member2();
        member1.start();
        member2.start();
    }
```

```
    private static class Member1 extends Thread {
        public void run() {
            synchronized (lock1) {
                System.out.println("Member 1: Inside lock 1...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Member 1: Before lock 2...");
                synchronized (lock2) {
                    System.out.println("Member 1: Inside lock 1 & 2...");
                }
            }
        }
    }

    private static class Member2 extends Thread {
        public void run() {
            synchronized (lock2) {
                System.out.println("Member 2: Inside lock 2...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Member 2: Before for lock 1...");
                synchronized (lock1) {
                    System.out.println("Member 2: Inside lock 1 & 2...");
                }
            }
        }
    }

}
```

Livelock

Sytuacja dość podobna do *deadlocka*, skutkująca tym samym: przynajmniej dwoma wątkami wzajemnie się blokującymi.

W przeciwieństwie jednak do *deadlocka* gdzie wątki po prostu w nieskończoność na siebie czekają i nie robią nic, to tutaj dzieje się bardzo dużo. Wręcz za dużo, a mianowicie wątki reagują wzajemnie na siebie w taki sposób, że się blokują tak samo, gdyby nie robiły nic. W konsekwencji wątki nie są w stanie wykonać do końca swoich bloków kodu. Idealnym, często podawanym przykładem *livelocka* z życia codziennego jest próba minięcia kogoś w ciasnym korytarzu. Na pewno miałeś/-aś w życiu taką sytuację, że próbując kogoś minąć przesunąłeś się w tym samym kierunku co ta druga osoba. ☺

Użyjmy poprzedniego, zmodyfikowanego przykładu, żeby to wszystko dokładnie prześledzić:

1. Pierwszy wątek uruchamia zsynchronizowany blok kodu blokujący dostęp do zasobu A.
2. Drugi wątek uruchamia zsynchronizowany blok kodu blokujący dostęp do zasobu B.
3. Pierwszy wątek dochodzi do momentu w swoim bloku kodu (zsynchronizowanym!), w którym potrzebuje dostępu do zasobu B (dostępu zsynchronizowanego, a więc musi zablokować ten zasób...

). Ten jest jednak zablokowany przez drugi wątek...

JAK NA RAZIE WSZYSTKO JAK W DEADLOCKU...

4. Pierwszy wątek widzi, że nie może zdobyć dostępu do zasobu B, zwalnia więc blokadę na swoim zasobie A i próbuje jeszcze raz od początku...
5. W międzyczasie drugi wątek próbuje uzyskać dostęp do zasobu A... nie może się dostać, więc zwalnia blokadę i próbuje wykonać swoją pracę jeszcze raz...

I TAK DALEJ...

Generalnie, *livelock* jest mniej powszechnym i znanym zjawiskiem niż *deadlock*. Natomiast zasady walki z nim są takie same jak przyjęte w przypadku wyżej. Przede wszystkim odpowiednia kolejność blokowania, *timeouty* itd.

Czasami o wiele łatwiej jest zrozumieć kod niż słowa pisane, spójrz zatem na poniższy przykład, który skończy się *livelockiem*:

Klasa LivelockExample

```
package pl.zajavka.java.workshop18_concurrency.examples;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LivelockExample {

    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    public static void main(String[] args) {
        LivelockExample example = new LivelockExample();
        new Thread(example::operation1, "Thread1").start();
        new Thread(example::operation2, "Thread2").start();
    }

    public void operation1() {
        while (true) {
            if (!tryLock(lock1, 1000)) {
                print("Cannot acquire lock1");
            }
            print("Lock1 acquired, trying to acquire lock2");
            sleep(1000);

            if (lock2.tryLock()) {
                print("Lock2 acquired");
            } else {
                print("Releasing lock1 - cannot acquire lock2");
                lock1.unlock();
                continue;
            }

            print("Operation1");
            break;
        }
        lock2.unlock();
```

```
            lock1.unlock();
        }

        public void operation2() {
            while (true) {
                if (!tryLock(lock2, 1000)) {
                    print("Cannot acquire lock2");
                }
                print("Lock2 acquired, trying to acquire lock1");
                sleep(1000);

                if (lock1.tryLock()) {
                    print("Lock1 acquired");
                } else {
                    print("Releasing lock2 - cannot acquire lock1");
                    lock2.unlock();
                    continue;
                }

                print("Operation2");
                break;
            }
            lock1.unlock();
            lock2.unlock();
        }

        public void print(String message) {
            System.out.printf("Thread: %s, msg: %s%n", Thread.currentThread().getName(), message);
        }

        public void sleep(long millis) {
            try {
                Thread.sleep(millis);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        public boolean tryLock(Lock lock, long millis) {
            try {
                return lock.tryLock(millis, TimeUnit.MILLISECONDS);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return false;
        }

    }
```

Starvation (i fairness)

Czyli po polsku *głód* - całkiem dobrze oddaje, o co chodzi w tym problemie. Wracając do *deadlocka*: wspomniany był tam przykład bardziej priorytetowych wątków jako jedno z rozwiązań problemu *deadlocka*. I taka jest właśnie geneza *starvation*: to sytuacja, w której wątek nie jest przez nic blokowany, ale nie może się dobić do wykonania jakiegoś bloku kodu, bo ciągle jest "wyprzedzany" przez inne wątki. Może to być efekt celowej priorytetyzacji wątków - wtedy procesor poświęca cały czas na inne wątki, a dla tych o mniejszym priorytecie brakuje już czasu (nie poruszają się one do przodu w żaden sposób w

rozumieniu wykonania programu).

Druga możliwość to... przypadek: warto zwrócić uwagę, że przed wykonaniem zsynchronizowanego bloku nie ma żadnej kolejki FIFO dla wątków - istnieje więc szansa, że jakiś wątek będzie miał "pecha" i będzie czekał w nieskończoność w momencie, gdy inne wątki bez przerwy będą blokować ten kod.

Odpowiedzią na problem starvation jest koncepcja *fairness*. Zakłada ona bycie *fair*, czyli poświęcanie czasu i dopuszczanie wszystkich wątków do wykonania. OK, w takim razie, na czym polega implementacja tej koncepcji w kodzie?

Po pierwsze pamiętaj, że przed blokiem `synchronized`, nie ma żadnej kolejki FIFO, w zasadzie nie ma żadnej kolejki! Wątki rzucają / wciskają się na oślep!

Wystarczy więc... zaimplementować swój mechanizm, który zagwarantuje trzymanie kolejki w sposób *fair*, przetrzymujący listę, na którą wpisujemy oczekujące wątki. Na szczęście taki *fair lock* mamy już zaimplementowany w Javie (`ReentrantLock` - wspominałem o tym), poniżej jego użycie:

Klasa FairnessExample

```
public class FairnessExample{
    private final ReentrantLock reentrantLock = new ReentrantLock(true); ①

    public void fairNotSynchronizedMethod() { ②
        reentrantLock.lock(); ③
        try {
            // critical section kodu
        } finally {
            reentrantLock.unlock() // nie zapomnij zwolnić na koniec
        }
    }
}
```

① Używamy konstruktora przyjmującego jako argument (*boolean fair*) - w ten sposób gwarantujemy pierwszeństwo dla najdłużej czekającego wątku.

② Rezygnujemy ze słówka `synchronized`.

③ Zamiast tego blokujemy przy użyciu naszego *fair locka*.

Na koniec tego tematu, oczywiście takie zabezpieczenie ma swój koszt w wydajności. Trzeba mieć to na uwadze podczas pisania, bo zawsze jest to jakiś tradeoff. W sytuacji, gdy kod już działa powoli, powinniśmy zastanowić się dwa razy czy gra jest warta świeczki (jako, że *starvation* jest rzadziej występującym problemem).

Volatile

Kolejnym elementem syntaxu Java, jaki poznamy, obok nowego słowa `synchronized`, jest słowo `volatile`. W przeciwieństwie do `synchronized` - używanego do metod czy bloków kodu, `volatile` możemy używać do deklarowania zmiennych. Tylko i wyłącznie do deklarowania zmiennych. Spójrz:

```
private volatile int insignificantNumber = 5;
```

Za co odpowiada `volatile`? Dosłownie, to słowo oznacza *ulotny, ulatniający się*. Poprzez dodanie tego słowa do deklaracji zmiennej instruujemy JVM, żeby ładował i zapisywał jej wartość nie do / z pamięci podręcznej procesora (cache), ale z RAMu. Dlaczego to istotne?

Wracamy teraz do modelu pamięci. Procesory mają swój cache. Cache jest bardzo szybki, co oznacza, że wątki będą tam przechowywać wartości zmiennych - w celach optymalizacyjnych. Oznaczając zmienną jako `volatile` upewniamy się, że odczyt i zapis będzie się odbywać każdorazowo poprzez RAM. Gwarantujemy sobie w ten sposób, że każdy wątek zobaczy wersję zapisaną w RAMie i od razu będzie zapisywał do RAMu (czyli do stacku bądź heapa).

Czy słowo `volatile` zabezpiecza nas przed *race condition* tak jak `synchronized`? Absolutnie nie! Nadal może dojść do scenariuszy opisywanych wcześniej, gdyż nadal odczyt danych może nastąpić w momencie, gdy inny wątek już na nich pracuje i zaraz je zmieni. Czyli samo **volatile NIE CZYNI NASZEGO KODU THREAD-SAFE**.

Patrząc na to z innej strony: `synchronized` "blokuje" nam dostęp do danego zasobu, dopóki pracuje na nim dany wątek. Słówko `volatile` nic nie blokuje, upewnia się jedynie, że zapis wartości zmiennej i jej odczyt będzie się odbywał natychmiast, a nie w momencie, gdy JVM uzna to za optymalne. Pamiętasz scenariusz *read-update-write*? No właśnie.

OK, w takim razie w ogóle, po co używać `volatile`, zapytasz? Skoro `synchronized` daje mi pewność, a z `volatile` może być różnie. Odpowiedź jest najłatwiej przedstawić na przykładzie kodu.

Użycie `volatile` nic nie blokuje, natomiast praktycznie wyklucza użycie przez procesor cache'a w odniesieniu do konkretnej zmiennej (`synchronized` nie zabrania użycia cache'a, wymusza jedynie jego "flush", czyli wypchnięcie wartości do pamięci w momencie zdejmowania locka / kończenia bloku kodu `synchronized`).

Poprawność działania programu jest bardziej istotna niż jego wydajność, dlatego znajdą się scenariusze, gdy użycie `volatile` w programie będzie potrzebne. Jednym z nich może być np. coś takiego:

Klasa VolatileUsageExample

```
public class VolatileUsageExample {

   public volatile boolean someVariableUpdatedByManyThreads = true;

    public void executeExample(){
        while(someVariableUpdatedByManyThreads) {
            System.out.println("Nadal ustawione na true");
        }
        System.out.println("Inny wątek zmienił na false");
    }
}
```

Załóżmy, że zmienna globalna `someVariableUpdatedByManyThreads` jest, jak nazwa sama wskazuje, zaktualizowana przez inny wątek na `false` w trakcie działania tej pętli. Jednak jej aktualizacja nie przerwie pętli... jest to spowodowane tym, że JVM nie spodziewa się zmiany wartości tej zmiennej w trakcie działania pętli i trzyma jej wartość w cache'u - co oczywiście doprowadziłoby w tej sytuacji do dobrze nam znanej *infinite loop*.

Poprzez oznaczenie `someVariableUpdatedByManyThreads` słowem kluczowym `volatile`, gwarantujemy, że

wartość zmiennej zostanie każdorazowo (przy każdej iteracji pętli `while`) pobrana z RAMu i w momencie aktualizacji jej wartości przez inny wątek - `pętla` zostanie przerwana już w pierwszej iteracji po zmianie.

Gwarancja "HAPPENS BEFORE"

Wspomnieliśmy, że słowo `volatile` daje gwarancję odczytu i zapisu zmiennej bezpośrednio do pamięci RAM z pominięciem cache procesora, ale to nie jedyna gwarancja, jaką dostajemy w pakiecie z `volatile`. Kolejną jest *happens before*, czasami nazywana *relacją happens before*. Co to oznacza?

JVM jest w stanie zmieniać kolejność wykonywania instrukcji twojego programu... tak jest(!!!11!11!). Co więcej, taką moc ma nawet sam procesor. Wchodzimy tutaj w dość niskopoziomowe tematy, ale warto wiedzieć, że tak długo, jak nie wpływa to na funkcjonalne działanie programu, to JVM może przesuwać kolejność wykonania instrukcji w celach optymalizacyjnych. Słówko `volatile` zmienia nam jednak trochę sytuację. Spójrz na poniższy przykład:

Klasa InstructionsReorderingExample

```
public class InstructionsReorderingExample {

   private int firstVariable;
   private int secondVariable;

    public void executeExample(){
        firstVariable = 2;
        secondVariable = 5;
        firstVariable = firstVariable++;
    }
}
```

Obydwie zadeklarowane zmienne nie wchodzą ze sobą w interakcje, dlatego JVM mógłby wykonać powyższe instrukcje w tej kolejności:

Klasa InstructionsReorderingExample

```
public class InstructionsReorderingExample {

   private int firstVariable;
   private int secondVariable;

    public void executeExample(){
        // (...)
        firstVariable = 2;
        firstVariable = firstVariable++;
        secondVariable = 5;
        // (...)
    }
}
```

Jak widzisz, dla funkcjonalnej strony programu nic to nie zmiana, a z punktu widzenia JVM, jest to pewna optymalizacja działania (`firstVariable` jest trzymane w cache'u i wszystkie wymagane przez program operacje są na nim wykonywane "od razu").

OK, ale co to ma wspólnego z *happens before*? Otóż oznaczenie zmiennej `volatile` nakłada pewne

ograniczenia na JVM w zakresie, właśnie, przesuwania instrukcji programu związanych z daną zmienną (odczytu i zapisu):

Zapis do zmiennej, który w kodzie znajduje się przed zapisem do zmiennej `volatile`, zawsze wydarzy się wcześniej (*happens before*!), przed zapisem do tej zmiennej `volatile` (JVM nie może przesunąć tych instrukcji na później, jak zrobił to w naszym przykładzie ze zmienną `secondVariable`, gdyby zmienna ta była oznaczona `volatile` - JVM nic nie mógłby zrobić).

I analogicznie dla odczytu (tylko tutaj oczywiście odczyt ma precedens): wszelkie pobranie wartości zmiennej `volatile`, które w kodzie znajdują się przed odczytami innych zmiennych, zawsze wydarza się przed (*happens before*) odczytami innych zmiennych. Wszystko dla zabezpieczenia przed niespodziewanymi zapisami / odczytami przez inne wątki.

Niby wszystko oczywiste, szczególnie w kontekście zapisu do RAMu który już sobie omówiliśmy. Warto jednak pamiętać, że taka gwarancja istnieje i co ona w praktyce oznacza.

Thread-safety w gotowych narzędziach.

Jak mogłeś/-aś się domyślić: nie tylko ty piszesz kod, który nie jest *thread-safe*. W samej Javie znajdziemy klasy, które nie są napisane z myślą o obsłudze przez wiele wątków i wynik ich działania w takich warunkach będzie po prostu często błędny. Do tego dochodzi wiele innych gotowych rozwiązań: biblioteki mniejsze i większe, frameworki itd. Przyjrzyjmy się najważniejszych przypadkom:

Thread-safe w Java API.

Przykładem takiej klasy (nie *thread-safe*) jest jedna ze znanych Ci już: `StringBuilder`.

`StringBuilder` jest klasą, która powstała w celu wygodnej obsługi Stringów. O ile sam String jest **immutable**, tak `StringBuilder` już nie - co dość ułatwia pracę na nim (sam widziałeś/-aś) no i jest bardziej wydajny. Jeśli musisz to wróć do momentu gdzie rozmawialiśmy o immutowalności - tam jest jeszcze wspomniana wydajność. Oczywiście tak jest i w tym przypadku.

No dobra, ale co w przypadku gdy piszę kod współbieżny? Muszę zrezygnować ze `StringBuildera` i robić wszystko na `Stringach`? Tak i nie: Java udostępnia *Thread-safe* alternatywę tzn. `StringBuffer`.

Czymże jest `StringBuffer`? Otóż można powiedzieć, że jest "klonem" `StringBuildera` (tyle, że powstałym wcześniej): posiada wszystkie te same metody co z `StringBuilder` - jedyna różnica jest taka, że wszystkie jego metody są zsynchronizowane, a więc *Thread-safe*. Minusem będzie oczywiście wydajność, a konkretnie jej spadek. Z racji` że obydwie klasy dostarczają dokładnie to samo robiące metody łatwo porównać ich czas wykonania. Wyniki zdecydowanie przemawiają za `StringBuilderem`, więc w jednowątkowych aplikacjach to zawsze powinna być domyślna opcja.

Na koniec o tych klasach to warto też pamiętać, że choć sam `StringBuilder` nie jest thread-safe, to może być używany w ten sposób, np. z zsynchronizowanego bloku kodu. Nie będziemy mieli korzyści wydajnościowych w tym przypadku, ale jest to raczej zalecana forma, bo używamy synchronizacji tylko, kiedy jest to potrzebne.

Innymi przykładami klas thread-safe są: `Vector` i `Hashtable`. Są to stare klasy, wprowadzone jeszcze w pierwszej wersji Java. Obydwie są już "obsolete" (przestarzałe). Dlaczego? Bo szybko dostały swoich

następców, o których pewnie pamiętasz tzn.: `ArraList` i `HashMap` - klasy bardzo podobne, stosowane w tym samym celu, ale podobnie jak `StringBuilder` NIE ZSYNCHRONIZOWANE. Są przez to o wiele bardziej wydajne.

Wygląda na to, że dawno temu, w czasach pierwszej Javy jej twórcy chcieli uszczęśliwić wszystkich na siłę i tworzyć wszędzie synchronizowane, thread-safe API. Kurs się jednak szybko zmienił i teraz mamy do dyspozycji o wiele szybsze API, a kwestia zabezpieczenia kodu przed *race condition* spoczywa już w pełni na nas.

Hibernate a thread-safety

Na koniec pogadajmy sobie trochę o temacie wielowątkowości w kontekście połączeń z bazą danych. Konkretnie pomówmy więc sobie jeszcze o thread-safe w Hibernate'cie. Chciałbym tutaj poruszyć kwestię dwóch interfejsów dostarczanych przez ten framework i są to klasy specyficzne dla Hibernate'a (nie znajdziecie ich w JPA). Mianowicie:

1. `SessionFactory`
2. `Session`

`SessionFactory`, czyli interfejs odpowiadający za tworzenie obiektów `Session` dla aplikacji. Jest on *thread-safe*. Jak stwierdza oficjalna dokumentacja Hibernate: zalecane jest, aby aplikacja miała jedną `SessionFactory` w całym swoim cyklu życia, a więc żeby była współdzielona między wątkami. Jest to w 100% bezpieczne i zdecydowanie zalecane podejście. Jako ciekawostka można powiedzieć, że thread-safe, jest w tym przypadku zapewnione przez immutowalność obiektu.

`Session`, czyli główny interfejs do komunikacji pomiędzy Javą a Hibernate'm, a co za tym idzie również bazą danych. Przypomnijmy, że to właśnie dzięki `Session` możemy wykonywać operacje CRUD na bazie z poziomu Javy.

I tutaj znowu odwołując się do najlepszego źródła, czyli dokumentacji: Klasa `Session` nie jest thread-safe. Twórcy zwracają uwagę, że jej implementacja również nie powinna być thread-safe. W tym przypadku każdy wątek powinien mieć swoją sesję bazy danych (utworzonych przez wspólne `SessionFactory`).

A więc tak powinien być używany Hibernate w wielowątkowych aplikacjach. Tak to zostało zaprojektowane przez twórców i trzeba przyznać, że jest to ładny i sensowny design. Dlatego właśnie, nie należy używać tego w inny sposób. Z drugiej strony poznaliśmy już Spring Data i tam nie używamy ręcznie ani `SessionFactory` ani `Session`.

To tematu wielowątkowości w kontekście pisania aplikacji WEB będziemy jeszcze wracać.

Może Ci się teraz zaświecić nad głową lampka z myślą: *przecież z aplikacji w Internecie jednocześnie może korzystać setki użytkowników, pewnie jest to jakoś powiązane z wielowątkowością*. Jest, jeszcze do tego przejdziemy.

Executor Service jako przykład Thread poola

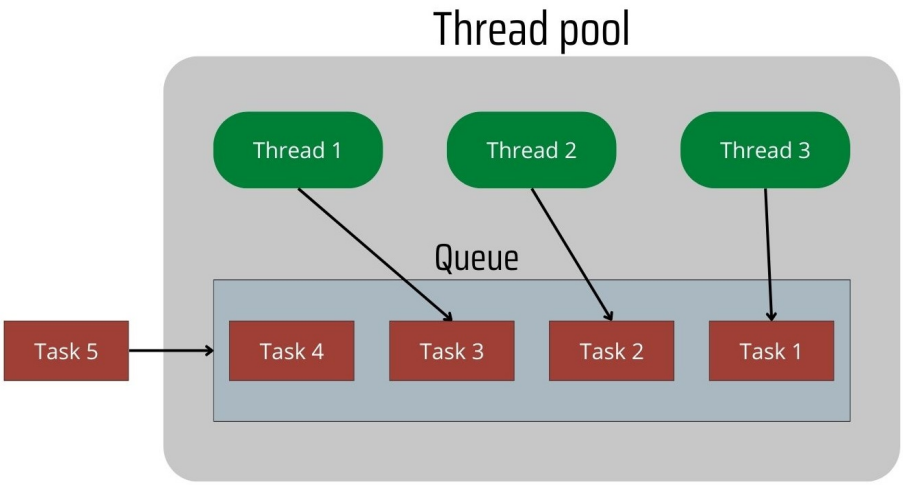
Omówiliśmy sobie wszystkie problemy, jakie możemy spotkać przy programowaniu współbieżnym oraz metody zapobiegania im. Teraz wrócimy to tematu... tworzenia wątków! A w zasadzie do alternatywy tworzenia nowych wątków, jakim jest wykorzystanie *thread poola*, czyli puli wątków. Swoją drogą, nie jest to pierwsza pula, z jaką mamy do czynienia.

Wcześniej rozmawialiśmy o:

- *String pool* - czyli, dla przypomnienia, przestrzeń w pamięci do przetrzymywania zdefiniowanych Stringów na wypadek ich przyszłego użycia.
- *Connection pool* - czyli, dla przypomnienia, pula połączeń do bazy danych, żeby móc takie połączenia wykorzystywać ponownie, zamiast tworzyć za każdym razem od nowa.

Podobna sytuacja ma miejsca dla wątków, *thread pool* w tym przypadku odgrywa analogiczną rolę. Jest to zbiór wątków, które czekają na wykorzystanie w celu wykonania przeznaczonych im zadań. Zamiast samemu tworzyć nowy wątek przy pomocy konstruktora, można posłużyć się wątkiem z *thread poola*, a po wykonaniu zadania zwrócić go tam z przeznaczeniem do wykonania kolejnych pojawiających się zadań. Za takim podejściem przemawia wyższa wydajność (w końcu nie tworzymy nowych obiektów, tylko korzystamy cały czas z tych samych) oraz wygoda. Podobnie jak w przypadku *fair locka* (czy szukając przykładów z innych obszarów: listy) można zaimplementować własny *thread pool*, natomiast Java dostarcza nam już gotowy interfejs `ExecutorService` wraz z dokumentacją i wygodnym zestawem metod.

Jak możemy sobie wyobrazić taki thread pool? O tak:



Obraz 7. Schemat puli wątków

Wysyłanie zadań

Sam `ExecutorService` jest co prawda interfejsem, ale w pakiecie dostajemy również klasy go implementujące. Jak pozyskać taką klasę? Do tego posłużymy się klasą `Executors` (która realizuje *factory design pattern*). Stworzymy z jej użyciem najprostszą możliwą implementację `ExecutorService`.

```
public class Main {
    public static void main (String[] args) {
        ExecutorService executorService = null;

        try {
            executorService = Executors.newSingleThreadExecutor();
            executorService
                .execute(() -> System.out.println("Pierwsze zadanie z użyciem ExecutorService"));
            executorService
                .execute(() -> System.out.println("... i drugie"));
        } finally {
            if(executorService != null) {
                executorService.shutdown();
            }
        }
    }
}
```

Przykład jest bardzo prosty:

1. Tworzymy nowy, prosty `executorService` (implementacja: `newSingleThreadExecutor()`). Ten executor jest jednowątkowy, to znaczy, że wykona wszystkie instrukcje w kolejności, w jakiej zostały mu przekazane, bo jest tu tylko jeden wątek. Ten String: *Pierwsze zadanie z użyciem ExecutorService* **ZAWSZE** pokaże się w konsoli jako pierwsze. Czy to znaczy, że nasza aplikacja jest *single-thread app*? Oczywiście nie, metoda `main()` to już jeden wątek i tworzymy osobny dla `executorService`. Mamy więc 2 wątki.
2. Wysyłamy nowo stworzonemu `executorService` zadania przy użyciu metody `execute()`. Jak już wspomniałem: zadania wykonają się jedno po drugim. Zadania są dostarczone w formie lambdy implementującej interfejs `Runnable`. Sama metoda `execute()` zwraca `void`, działa to więc w sposób asynchroniczny - wysyłamy zadanie i nie interesuje nas już jego wynik i tym bardziej nie będziemy na ten wynik czekać.
3. Używamy konstrukcji `try-finally`, aby upewnić się, że zamykamy serwis (wszystkie aktywne wątki w ramach stworzonego `executorService`) - do tego służy nam metoda `shutdown()`. Blokuje ona wysyłanie nowych requestów, czeka na już rozpoczęte, a następnie zamyka cały `executorService`. Interfejs udostępnia więcej metod do zamykania, warto pamiętać o:
    - `shutdownNow()` - działa jak zwykły `shutdown()`, ale próbuje również stopować już rozpoczęte zadania. Metoda ta zwraca listę zadań oczekujących na wykonanie w formie `List<Runnable>`,
    - `awaitTermination(long timeout, TimeUnit unit)` - jest to "najbardziej miękka" metoda, która blokuje nam zamknięcie wątku do czasu wykonania wszystkich zadań. Tutaj oczywiście musimy wyspecyfikować *timeout* określający, jak długo ma trwać taka blokada. Metoda zwraca `true` w przypadku gdy uda się zamknąć `executorService` i `false`, gdy się to nie uda i pierwszy nastąpi timeout.

Na koniec warto pamiętać o dwóch metodach niesłużących bezpośrednio do zamykania, ale również przydatnych. Są to:

- `isShutdown()` - informująca (poprzez zwrócony `boolean`) czy `executorService` został zamknięty,
- `isTerminated()` - informująca czy `executorService` jest zamknięty oraz, czy wykonał już wszystkie powierzone mu zadania (może więc dojść do sytuacji, gdy użyliśmy już metody `shutdown()`, metoda `isShutdown()` zwróci `true`, ale wszystkie zadania jeszcze nie zostały wykonane, zatem `isTerminated()`

zwróci `false`).

Metoda submit()

Jak do tej pory wszystko powinno być logiczne, ale widzisz już pewnie jedną wadę metody `execute()`, mianowicie nie daje ona żadnej informacji zwrotnej o wykonywanym zadaniu. Jeśli interesuje nas wynik, powinniśmy użyć metody `submit()`. Jej sygnatura to:

Sygnatura metody submit()

```
Future<?> submit(Runnable runnable)
```

W tej sytuacji otrzymujemy w zwrotce obiekty generycznej klasy `Future`, dzięki której możemy "śledzić" postęp naszego zadania. Poza tym powyższa metoda nie różni się wcale od `execute()`. Co więcej, to `submit()` jest o wiele powszechniej wykorzystywaną metodą.

Odbieranie zadań

Spójrzmy teraz na obiekt `Future<V>` który posłuży nam do sprawdzenia co dzieje się z naszym zadaniem. Służy nam do tego kilka bardzo przydatnych metod:

- `isDone()`
- `isCancelled()`
- `cancel()`

Wszystkie z powyższych robią dokładnie to, co wynika z ich nazw. Dodatkowo wszystkie z nich zwracają `boolean` dokładnie w ten sposób, w jaki sobie wyobrażasz. Do samego pobrania wyniku posłużą nam poniższe metody:

- `get()`
- `get(long timeout, TimeUnit timeUnit)`

Metoda `get()` zwróci nam wynik działania zadania. Typ oczywiście będzie zależny od tego, co dany `Future` przechowuje, a więc od spodziewanego przez nas wyniku danego taska. Jeśli wynik działania nie będzie dostępny, metoda `get()` będzie działać w nieskończoność. Zanim ją wywołamy, warto więc sprawdzić status zadania przy użyciu `isDone()`. Możemy również użyć metody `get()`, która jako parametr przyjmuje określoną przez nas wartość *timeoutu*.

Powiesz teraz pewnie... zaraz, ale przecież każda implementacja `Runnable` musi być zawsze `void`. Co wtedy? No cóż, wywołanie `get()` na naszym obiekcie `Future` zawsze w takim przypadku zwróci nam `null`a. Jeśli nasz scenariusz wymaga sprawdzania wyników działania wysłanych zadań (np. odnosząc się do przykładu z generatorem nazw faktur: chcemy zwrócone nazwy wysłać komuś mailem) również użyjemy metody `submit()`, natomiast w innym wariancie... I tutaj cały na biało wchodzi interfejs `Callable`.

Callable

Znasz już interfejs `Runnable` wraz z jego metodą `run()`, którą przesłaniasz używając lambdy w metodzie `ExecutorService.submit()`. Znasz też jego największą wadę - metoda `run()` zwraca typ `void`.

Korzystając z interfejsu `Callable` mamy możliwość zwrócenia wyniku. `Callable`, podobnie do `Runnable` jest interfejsem funkcyjnym, więc będziemy mogli do jego implementacji użyć wygodnej lambdy. Natomiast jego jedyna metoda wygląda następująco:

Definicja interfejsu Callable

```
V call() throws Exception
```

Metoda `call()` jest w stanie wyrzucać *checked Exception*, co już jest bardzo przydatne. Natomiast co najważniejsze, metoda zwraca również typ generyczny, co daje Ci w zasadzie nieograniczone możliwości jej użycia. Jak użyć interfejsu `Callable` w `ExecutorService`? Nic prostszego:

Definicja metody submit() w ExecutorService

```
<T> Future<T> submit(Callable<T> callable)
```

`ExecutorService` udostępnia przeciążoną metodą `submit()`, która pozwala na użycie interfejsu `Callable`. Świetnie! Teraz używając metody `get()` (po wykonaniu zadania) dostaniemy obiekt z wynikiem naszego działania. Spójrzmy na prosty przykład:

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

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

        final String TEST_STRING = "Mój kod jest bezbłędny";
        final String NEW_STRING = "czasami ";

        ExecutorService executorService = null;

        try {
            StringBuilder stringBuilder = new StringBuilder(TEST_STRING);
            executorService = Executors.newSingleThreadExecutor();
            System.out.println("Tutaj rozpoczynam");
            Future<StringBuilder> expectedResult = executorService
                .submit(() -> stringBuilder.replace(13, 16, NEW_STRING));
            printValueWhenReady(expectedResult);
        } catch (ExecutionException | InterruptedException exception) {
            // obsługa wyjątku
        } finally {
            if (executorService != null) {
                executorService.shutdown();
            }
        }
    }
```

```
    private static void printValueWhenReady(
        Future<StringBuilder> stringBuilderFuture
    ) throws ExecutionException, InterruptedException {
        while (!stringBuilderFuture.isDone()) {
            System.out.println("Czekam na wykonanie zadania!");
        }
        System.out.println(stringBuilderFuture.get().toString());
    }
}
```

Prześledź sobie, co tu się dzieje, z wykorzystaniem debbugera. Uruchamiając program z pewnością zauważysz, że zanim zobaczysz wynik działania swojego zadania, w konsoli kilka (kilkadziesiąt) razy zobaczysz *Czekam na wykonanie zadania!*. Idealny przykład ilustrujący wielowątkowość naszego programu. Zadanie stworzenia nowego Stringa przy użyciu StringBuildera zostało wydelegowane przez `executorService`, natomiast samo wywołanie metody `main()` poszło dalej do kolejnej metody `printValueWhenReady()`.

Parallel streamy

Spotkałeś/-aś już się ze streamami. Stream API dołączyło do Javy w wersji 8 i od razu stało się jednym z ulubionych narzędzi programistów - ze względu na swoją wygodę. Co ciekawe obok streamów, z jakimi miałeś/-aś do czynienia do tej pory, istnieją również stramy równoległe (ang. *parallel*) streamy.

Jak sama nazwa wskazuje, praca na kolekcjach nie odbywa się tam jak w przypadku "normalnych" streamów - jeden po drugim (sekwencyjnie), ale właśnie "równolegle". Oczywiście, żeby było to możliwe, *parallel streamy* używają pod maską wielu wątków. Potencjalnie może przyspieszyć to pracę na wielu kolekcjach (czasami bardzo mocno). Przyjrzyjmy się im zatem (tym stream'om).

Tworzenie parallel streamów

Są dwie metody tworzenia parallel streamów. Przerobienie już istniejącego "sekwencyjnego" streama na *parallel stream* oraz stworzenie parallel streama z Javowej kolekcji.

Pierwsza metoda

```
Stream<Integer> exampleOfRegularStream = Stream.of(1, 2, 3);
Stream<Integer> exampleOfParallelStream = exampleOfRegularStream.parallel();
```

Jak widzisz *parallel stream* nie posiada innego typu, jest to dokładnie ten sam typ danych z jedynie "włączoną" obsługą równoległą jego elementów.

I druga wspomniana metoda:

```
Stream<Integer> anotherExampleOfParallelStream = Arrays.asList(1, 2, 3).parallelStream(); ①
```

① Przypomnienie z innej beczki, lista stworzona z użyciem `Arrays.asList()` będzie **fixed-size**! Czyli możesz zmienić wartość elementów w istniejących w tej liście (przy pomocy metody `set()`), ale nie możesz dodać nowego elementu przy wykorzystaniu metody `add()`. Gdy przejdziesz do dokumentacji

metody `Arrays.asList()`, znajdziesz takie zdanie: *Returns a fixed-size list backed by the specified array.*

To wszystko - nic trudnego, druga metoda wydaje się bardziej zwięzła, ale tak naprawdę obydwie są równie mile widziane.

Procesowanie parallel streamów

I co można zrobić z takim parallel streamem? Jak już zauważyłeś/-aś, jest to dokładnie ten sam typ danych co Stream, możemy więc z nim robić wszystko to samo co ze zwykłymi Streamami, lista dostępnych metod jest przecież ta sama. To na co musimy uważać to efekty działania naszego streama. Najbardziej oczywisty przykład:

```
Stream<Integer> anotherExampleOfParallelStream = Arrays.asList(1, 2, 3).parallelStream();
anotherExampleOfParallelStream.forEach(element -> System.out.print(element + ","));
```

W przypadku sekwencyjnego streama, `println()` w konsoli będzie wyglądał następująco: `1,2,3`. **Natomiast w przypadku parallel streama nie możemy być pewni kolejności**. Przykład jest bardzo prosty, ale uruchamiając go parę razy, powinieneś/-aś zauważyć różnicę. Podkreślam więc: możesz używać wszystkich metod Stream API na parallel streamach, miej jednak zawsze na uwadze, że wynik tych operacji nie powinien być ze sobą powiązany.

Metoda reduce()

Chciałbym przywołać jeden przykład w odniesieniu do nieprzewidywalności otrzymanego rezultatu, w przypadku gdy stosujemy parallel streams.

Poznaliśmy już wcześniej operacją terminującą streamy - `reduce()`. Operacja ta służyła do zredukowania całego Streama do jednego finalnego obiektu. Operacja ta mogła przyjmować do trzech parametrów:

- *identity* - pierwszy parametr, który służył do "rozpoczęcia" budowania zredukowanego obiektu,
- *accumulator* - paramter określający funkcję dołączającą kolejny element do wyniku,
- *combiner* - funkcja złączająca elementy wyniku, musi być kompatybilna z funkcją *accumulator*.

Wiem, że nadal jest to mgliste, więc posłużmy się przykładem:

```
String reduce = Stream.of('z', 'a', 'j', 'a', 'v', 'k', 'a')
    .reduce(
        "",
        (previous, next) -> previous + next,
        (left, right) -> left + right
    );
System.out.println(reduce);
```

Zmienne `previous` oraz `next` służą do rozróżnienia, która zmienna reprezentuje ciąg znaków, który został już zbudowany, a która reprezentuje kolejny znak, który jest dołączany. Zwróć uwagę, że trzeci argument, czyli *combiner* jest typu `BinaryOperator`, podczas, gdy *accumulator* jest typu `BiFunction`.

W przypadku streamów sekwencyjnych wynik jest przewidywalny, a co w przypadku streamów

parallel?

W przypadku parallel streams, metoda `reduce()` działa w ten sposób, że przygotowuje sobie w locie wartości pośrednie, a następnie łączy ze sobą te wartości pośrednie, by uzyskać ostateczny wynik. Oznacza to, że o ile w przypadku sequential stream, wynikowy String *zajavka* zostałby zbudowany znak po znaku, to w przypadku parallel stream, mogłyby zostać utworzone ciągi: *za*, *ja* oraz *vka*, a następnie połączone w wynik końcowy.

Pojawia się wtedy natomiast kolejny problem - *kolejność*. Jeżeli elementy będą łączone ze sobą w złej kolejności, możemy otrzymać rezultat: *zavkaja* lub *vkajaza*. Java Stream API daje nam możliwość przeprowadzania redukcji przy wykorzystaniu parallel streams, tak, żeby wynik został złączony w stosownej kolejności. Musimy jednak wtedy pamiętać o kilku zasadach:

1. Parametr *identity* musi być zdefiniowany w taki sposób, żeby dla wszystkich elementów w streamie, wykonanie operacji `combiner.apply(identity, streamElement)` było równe `streamElement`,
2. Operator wykorzystywany w parametrze *accumulator* musi być asocjacyjny i bezstanowy, czyli wynik operacji:

```
(elementA operator elementB) operator elementC
```

będzie równy:

```
elementA operator (elementB operator elementC)
```

3. Operator wykorzystywany w parametrze *combiner* również musi być asocjacyjny i bezstanowy, w taki sposób, że wynik operacji:

```
combiner.apply(streamElement1, accumulator.apply(identity, streamElement2))
```

będzie równy:

```
accumulator.apply(streamElement1, streamElement2)
```

Jeżeli będziesz przestrzegać tych zasad podczas tworzenia parametrów wywołania metody `reduce()`, możesz wtedy wykonać taką redukcję na parallel stream, a rezultat tego wywołania zachowa się tak, jakby zachował się w przypadku sequential stream.

Zasady te również mają zastosowanie w przypadku streamów sekwencyjnych, natomiast tam nie widzieliśmy żadnych efektów ubocznych, ze względu na to, że te streamy zawsze wykonywały się w określonej kolejności. W przypadku parallel stream kolejność nie jest gwarantowana, dlatego nietrzymanie się wspomnianych zasad w kontekście operacji `reduce()` może skutkować efektami niepożądanymi.

Można powiedzieć, że przedstawiony wcześniej przykład jest zgodny ze wspomnianymi zasadami. A jak wyglądałby przykład, który tych zasad nie spełnia? Operator odejmowania nie jest asocjacyjny, dlatego właśnie poniższy przykład skończy się w sposób nieprzewidywalny, jeżeli zaczniemy go uruchamiać

przy wykorzystaniu parallel stream:

```
Integer reduce = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .parallelStream()
    .reduce(0, (a, b) -> a - b);
System.out.println(reduce); ①
```

① Nawet jeżeli w Twoim przypadku efekt działania `stream()` i `parallelStream()` jest ten sam, to twórcy języka nie dają gwarancji, że będzie tak przy każdym ponownym uruchomieniu tego kodu.

Spójrzmy jeszcze na przykład, w którym parametr *identity* nie spełnia wspomnianych reguł:

```
String reduce = Stream.of('z', 'a', 'j', 'a', 'v', 'k', 'a')
    .parallel()
    .reduce(
        "_",
        (previous, next) -> previous + next,
        (left, right) -> left + right
    );
System.out.println(reduce);
```

Tak samo, jak w poprzednim przypadku, powyższy kod może wydrukować na ekranie: *_z_a_j_a_v_k_a*, natomiast zgodnie z założeniami twórców, taki zapis może skutkować nieprzewidzianym rezultatem.

Zarówno dwu- jak i trzy- argumentowa wersja metody `reduce()` wspiera przetwarzanie przez parallel stream. Zaleca się stosowanie metody `reduce()` z trzema argumentami podczas pracy z parallel streams. Jeżeli podamy parametr *combiner*, JVM może wtedy bardziej wydajnie podzielić wykonywane operacje.

Parallel stream vs thread pool

Ktoś może dojść do wniosku: *po co stosować thread pool, jak można na danym zbiorze zrobić parallelStream() i efekt będzie ten sam?*

Jak to bywa w programowaniu - *to zależy*. `ExecutorService` i thread poole dają Ci większą możliwość konfiguracji parametrów, np. narzucenia ilości wątków w puli, określenie timeoutów itp. Stosując parallel stream, JVM może stworzyć tyle wątków, ile mu jest potrzebne do przetworzenia danego streama. W przedstawionym zapisie `parallelStream()` nie mamy nad tym kontroli.

Dlatego można wysnuć wniosek, żeby stosować parallel stream do niezbyt skomplikowanych operacji, podczas gdy te bardziej złożone powinny być realizowane przy wykorzystaniu puli wątków.

Klasy Atomic

Dochodzimy już do końca tej notatki, natomiast warto wspomnieć o częstym błędzie popełnianym przez deweloperów. Zacznijmy od pytania: *czy operatory są thread safe?* Albo zadajmy to pytanie inaczej, *czy kod, który wykorzystuje np. operator dodawania jest thread safe?*

Potrafimy już stworzyć wiele wątków, które mają dostęp do tych samych obiektów w pamięci, w takim

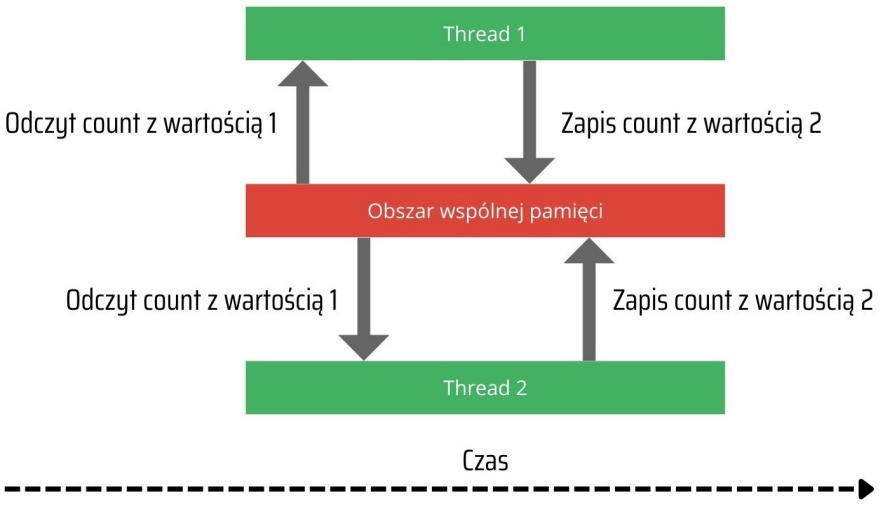
razie w wielu przypadkach musimy być pewni, że dostęp do tych danych jest zorganizowany w taki sposób, że działanie programu nie skończy się nieoczekiwanymi wynikami. Spójrz na poniższy przykład:

```
public class OperatorExample {
    private int count = 0;

    private void incrementAndPrint() {
        System.out.print((count = count + 1) + " ");
    }

    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newFixedThreadPool(50);
            OperatorExample example = new OperatorExample();
            for (int i = 0; i < 10; i++) {
                service.submit(example::incrementAndPrint);
            }
        } finally {
            if (service != null) {
                service.shutdown();
            }
        }
    }
}
```

Wartość licznika jest aktualizowana przy wykorzystaniu operatora +. Jeżeli dwa wątki wykonają jednocześnie operację dodawania, odczytując starą wartość, zanim zostanie zapisana ta nowa, oba wątki zwiększą wtedy tę starą wartość o 1. Inaczej mówiąc, jeden wątek nadpisze wtedy pracę drugiego. Spójrz na poniższą grafikę, wtedy łatwiej będzie to zrozumieć.



Obraz 8. Brak synchronizacji

Jak widać na powyższej grafice, drugi wątek odczytuje tę samą wartość, co odczytał pierwszy, zanim ten pierwszy zdążył ją zaktualizować. W tym sensie aktualizacja wartości wykonana przez pierwszy wątek została nadpisana przez aktualizację wartości dokonaną przez drugi wątek. Na tej podstawie można wysnuć wniosek, że operacja dodawania nie jest thread safe. Co więcej, opisana sytuacja jest dobrym przykładem *race condition*.

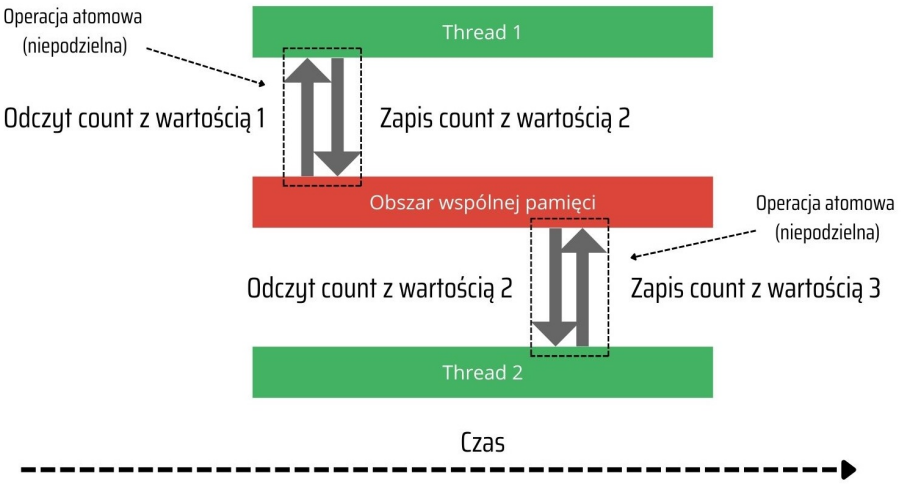
Jeżeli uruchomisz powyższy kod, to na ekranie możesz zobaczyć, np. takie wartości:

```
Uruchomienie pierwsze: 9 8 2 2 3 1 7 4 5 6
Uruchomienie drugie: 10 4 8 1 3 9 5 2 7 6
Uruchomienie trzecie: 10 6 3 1 5 7 8 9 4 2
```

Czyli w jednym z trzech przypadków, dwa wątki odczytały jednocześnie tę samą wartość i ten, który skończył jako drugi, nadpisał wartość zaktualizowaną przez ten pierwszy. Przez to w wydruku zabrakło wartości *10*.

W Javie występuje taka paczka jak `java.util.concurrent.atomic`. Służy ona do ułatwienia dostępu do wartości prymitywnych oraz referencji w środowisku wielowątkowym. W przypadku prymitywa, który został przedstawiony wyżej, operacja dodawania nie jest atomowa. Można ją rozdzielić na dwa zadania, odczytu oraz zapisu, które mogą być wzajemnie przeplatane pomiędzy wątkami.

Atomowość (niepodzielność) w tym przypadku oznaczałaby, że operacja miałaby zostać przeprowadzona jako pojedyncza porcja pracy, bez możliwości ingerencji innego wątku. Thread safe operacja zwiększenia wartości licznika oznaczałaby takie wykonanie operacji, że odczyt i zapis byłyby traktowane jako pojedyncze działanie i żaden inny wątek nie miałby dostępu do tej zmiennej podczas wykonywania tej operacji. Spójrz na poniższą grafikę:



Obraz 9. Operacja atomowa

Każdy wątek próbujący uzyskać dostęp do zmiennej `count` będzie musiał poczekać, gdyż operacja atomowa jest w trakcie wykonywania, aż zostanie ona ukończona. Oczywiście, ten wyłączny dostęp dotyczy tylko wątków próbujących uzyskać dostęp do zmiennej `count`. Zastosowanie zmiennej *atomowej* nie będzie miało wpływu na inne miejsca w pamięci.

Z racji, że przedstawiony przypadek jest w Java popularny, Concurrency API wprowadza kilka klas, które mają pomóc w takich sytuacjach i wspierać atomowe operacje.

Tabela 1. Przykładowe klasy Atomic w Java

Nazwa Klasy	Opis
AtomicBoolean	Wartość `boolean`, która może być aktualizowana atomowo
AtomicInteger	Wartość `int`, która może być aktualizowana atomowo

Nazwa Klasy	Opis
AtomicIntegerArray	Tablica `int`, w której elementy mogą być aktualizowane atomowo
AtomicLong	Wartość `long`, która może być aktualizowana atomowo
AtomicLongArray	Tablica `long`, w której elementy mogą być aktualizowane atomowo
AtomicReference	Generyczna referencja do obiektu, która może być aktualizowana atomowo
AtomicReferenceArray	Tablica generycznych referencji do obiektów, w której elementy mogą być aktualizowane atomowo

Każda z wymienionych klas zawiera metody, które są równoważne do wielu wbudowanych operatorów, takich jak np. operator przypisania czy operator inkrementacji. Spójrz na poniższą tabelę.

Tabela 2. Przykładowe operacja w klasach Atomic w Java

Nazwa metody	Opis
get()	Otrzymaj obecną wartość
set()	Przypisz wartość, zamiennik operatora przypisania
getAndSet()	Atomowo przypisz nową wartość i zwróć starą wartość
incrementAndGet()	Dla klas numerycznych, atomowy pre-increment, operacja będąca odpowiednikiem ++value
getAndIncrement()	Dla klas numerycznych, atomowy post-increment, operacja będąca odpowiednikiem value++
decrementAndGet()	Dla klas numerycznych, atomowy pre-decrement, operacja będąca odpowiednikiem --value
getAndDecrement()	Dla klas numerycznych, atomowy post-decrement, operacja będąca odpowiednikiem value--

Napisz teraz powyższy przykład w ten sposób:

```
public class OperatorExample {
    private AtomicInteger count = new AtomicInteger(0);

    private void incrementAndPrint() {
        System.out.print(count.incrementAndGet() + " ");
    }

    // .. reszta bez zmian
}
```

Po dokonaniu zmiany, liczby od 1 do 10 będą wydrukowane na ekranie za każdym razem. Użycie klasy `AtomicInteger` zapewnia spójność danych i żadna wartość nie zostanie utracona z powodu jednoczesnego dostępu do niej dwóch wątków.

JDK 21

Java 21 wprowadza kilka istotnych zmian do tematu wielowątkowości. Niektóre z tych zmian zostały zapoczątkowane już we wcześniejszych wersjach Java, natomiast przyjmiemy JDK 21 jako taki "game changer".

Chcemy zgromadzić tematy dotyczące wielowątkowości w jednym miejscu, dlatego właśnie omawiamy zmiany dotyczące wielowątkowości w Javie wprowadzone w JDK 21 w tym warsztacie, zamiast w warsztacie o aktualizacji wersji Java.

Virtual Threads

Virtual Threads (*Wątki wirtualne*) zostały wprowadzone w JEP-444. Założeniem tej zmiany jest wprowadzenie do Javy lekkich wątków, które zmniejszą wysiłek potrzebny do napisania i utrzymania aplikacji wielowątkowych.

Pewnie udało Ci się na tym etapie zauważyć, że wielowątkowość do prostych nie należy. Wraz z tą zmianą ma być łatwiej i prościej, zobaczymy, co z tego wyjdzie.

Platform Threads

Na tym etapie wiesz już, że w Javie można pisać aplikacje wielowątkowe wykorzystując np. klasę `java.lang.Thread`. Wspomniana klasa istniała w Java przed JDK 21 i istnieje nadal, natomiast do wątków tworzonych w ten sposób będziemy odnosić się nazywając je wątkami platformowymi (*Platform Thread*).

Nazwa *Platform Thread* wynika z tego, że stworzone w ten sposób wątki są zazwyczaj mapowane 1:1 na wątki systemu operacyjnego. Należy tutaj zaznaczyć, że wątki systemu operacyjnego są uważane za "ciężkie". W zależności od systemu operacyjnego, wątek taki może zużywać od 2MB do 10MB. Czyli wychodzi na to, że jeżeli chcesz stworzyć aplikację, która będzie wykorzystywała 100 wątków, będzie Ci potrzebne od 200MB do 1000MB. Dla porównania, jeżeli Twoja aplikacja będzie korzystała z 1000 wątków, będzie Ci potrzebne od 2GB do 10GB wolnej pamięci i tak dalej...

Jak widzisz, jest to pewnego rodzaju ograniczenie, które w konsekwencji wpływa na skalowalność aplikacji, albo inaczej mówiąc, wpływa to np. na to, ilu użytkowników dana aplikacja jest w stanie jednocześnie obsłużyć.

Dygresja, takie np. systemy bankowe są tworzone w taki sposób, żeby jednocześnie mogło z takiego systemu korzystać dużo osób, a nie np. 10. Wyobrażasz sobie, że wchodzisz na stronę bankowości internetowej Twojego banku i wyświetla się napis: "Musisz poczekać, jesteś 641 w kolejce!". No ja też nie ... ☺

Stąd właśnie aplikacje muszą być skalowalne, inaczej mówiąc, napisane w taki sposób, żeby były przygotowane na zwiększanie jednoczesnej ilości użytkowników, którzy z takiej aplikacji mogą korzystać.

Powyższy przykład jasno pokazuje, że podczas tworzenia aplikacji, zawsze pojawiają się jakieś ograniczenia - w tym przypadku, zasoby dostępne na komputerze, na którym aplikacja jest

uruchamiana, stanowią takie ograniczenie. Nie stworzysz więcej *platform threads* niż pozwoli Ci dostępność pamięci komputera.

To, co zostanie napisane poniżej będzie omówione bardzo pobieżnie. Temat będzie rozkładany na czynniki pierwsze, gdy przejdziemy do własnoręcznego pisania aplikacji uruchamianej na serwerze, ale muszę to tutaj wpleść, żeby dać szerszy kontekst omawianego problemu.

Thread per request

Zaczniemy od maksymalnie uproszczonego wprowadzenia do aplikacji WEB. Będzie nam to potrzebne do zrozumienia omawianego problemu. Sama tematyka aplikacji WEB i sposób ich tworzenia będą omawiane w kolejnych warsztatach.

Maksymalnie uproszczone wprowadzenie do aplikacji WEB

Realizując ten kurs, korzystasz z platformy *app.zajavka.pl*, która musi być uruchomiona na jakimś serwerze. Serwer to komputer, który fizycznie gdzieś jest, do którego możesz połączyć się przez sieć i który dostarcza treść wyświetlaną w Twojej przeglądarce.

Serwery działają na zasadzie *request - response*, czyli, Twoja przeglądarka wysyła żądanie do serwera, żeby pobrać zawartość strony internetowej do wyświetlenia, a serwer odpowiada, gdzie odpowiedzią jest treść strony, którą wyświetla przeglądarka.

Pamiętaj, że omawiamy to w maksymalnym uproszczeniu, wrócimy jeszcze do tego.

Wielowątkowość do aplikacji serwerowych ma się w taki sposób, że (w ogromnym uproszczeniu), gdy wysyłamy żadanie do serwera, tworzony jest tam nowy wątek, żeby to żądanie obsłużyć. Serwer wysyła *response* (czyli odpowiedź na *request*) i wspomniany wątek kończy swoje działanie.

Załóżmy, że platforma *app.zajavka.pl* jest uruchomiona w oparciu o Tomcat (serwer, który będziemy poznawać później). Załóżmy również, że w normalny dzień ruch na platformie wynosi około 500 żądań na sekundę. Domyślnie maksymalna ilość wątków, jaką tworzy Tomcat wynosi 200. Jeżeli do tego dołożymy, że średni czas odpowiedzi aplikacji na żądanie wynosi 250ms, to możemy oszacować, że aplikacja jest w stanie obsłużyć np. około 800 żądań na sekundę, czyli w ciągu normalnego dnia to wystarczy.

Co natomiast się stanie, gdy użytkownicy nagle dojdą do wniosku, że to jest idealny moment, żeby się pouczyć i nagle z aplikacji zacznie korzystać 1000 osób w ciągu sekundy? Aplikacja (serwer) zostanie przytłoczona ilością żądań.

Wynika to z tego, że w tym podejściu mamy określoną maksymalną liczbę wątków dedykowanych do obsługi żądań, gdzie każdy wątek jest przypisywany do każdego przychodzącego żądania i taki wątek zarządza cyklem życia takiego żądania. Oznacza to, że aplikacja jest w stanie obsłużyć określoną liczbę jednoczesnych żądań, a liczba ta wynika z maksymalnej ilości możliwych do utworzenia wątków.

Oczywiście jednym ze sposobów na rozwiązanie tego problemu jest zwiększenie maksymalnej ilości dostępnych wątków, ale tutaj trzeba pamiętać, że każdy wątek zajmuje pamięć. Oznacza to, że w pewnym momencie skończy się pamięć na serwerze, na którym jest aplikacja i może się to skończyć errorem *OutOfMemoryError*.

W powyższym podejściu, za każdym razem, gdy jest tworzony nowy wątek, system operacyjny alokuje odpowiednią ilość pamięci, bo przecież każdy wątek ma swój dedykowany stack. Zatem, jeżeli aplikacja zacznie tworzyć coraz więcej wątków, to doprowadzi to do wyczerpania dostępnej pamięci. A dostępna pamięć też jest przecież skończonym zasobem.

Jak to się ma do omawianego problemu?

Skoro w typowej aplikacji serwerowej stosowane jest podejście *One thread per request* (są też inne, ale to jest najprostsze), oznacza to, że w pewnym momencie pojawia się granica sprzętowa. Nie stworzymy więcej wątków, niż pozwala nam na to fizyczna pamięć serwera.

Gdy dołożymy do tego informację o tym, że *platform threads* są mapowane 1:1 na wątki systemu operacyjnego, bardzo łatwo możemy zauważyć, że możliwa jest sytuacja, w której ilość możliwych do stworzenia wątków się zwyczajnie wyczerpie.

Podejście to (*One thread per request*) ma wiele zalet, jak np. łatwiejsze zarządzanie aplikacją i sprzątanie nieużywanych zasobów, natomiast wprowadza granicę, jaką jest dostępność pamięci.

Lekkie wątki

Mam nadzieję, że rozumiesz już na tym etapie, że nie da się zwiększyć ilości możliwych do stworzenia wątków w systemie operacyjnym, bez zwiększania możliwości sprzętowych (hardware). Rozwiązaniem w tym przypadku może być kolejna warstwa abstrakcji, która pomoże w zarządzaniu tymi wątkami i np. w reużywaniu ich.

Przypomnę, że jest takie fajne powiedzenie:

> Każdy problem w informatyce da się rozwiązać wprowadzając kolejny poziom abstrakcji, oprócz problemu zbyt dużej ilości poziomów abstrakcji.

Taka warstwa abstrakcji powodowałaby, że wątek, który nazwiemy *lekkim*, nie będzie powiązany 1:1 z konkretnym wątkiem systemu operacyjnego. Wątki byłyby zarządzane przez warstwę abstrakcji, a nie przez system operacyjny. Dzięki temu moglibyśmy stworzyć większą liczbę wątków i zarządzałaby nimi warstwa abstrakcji.

Ogólnie, ta koncepcja nie jest niczym nowym, wiele języków ma jakąś formę lekkich wątków:

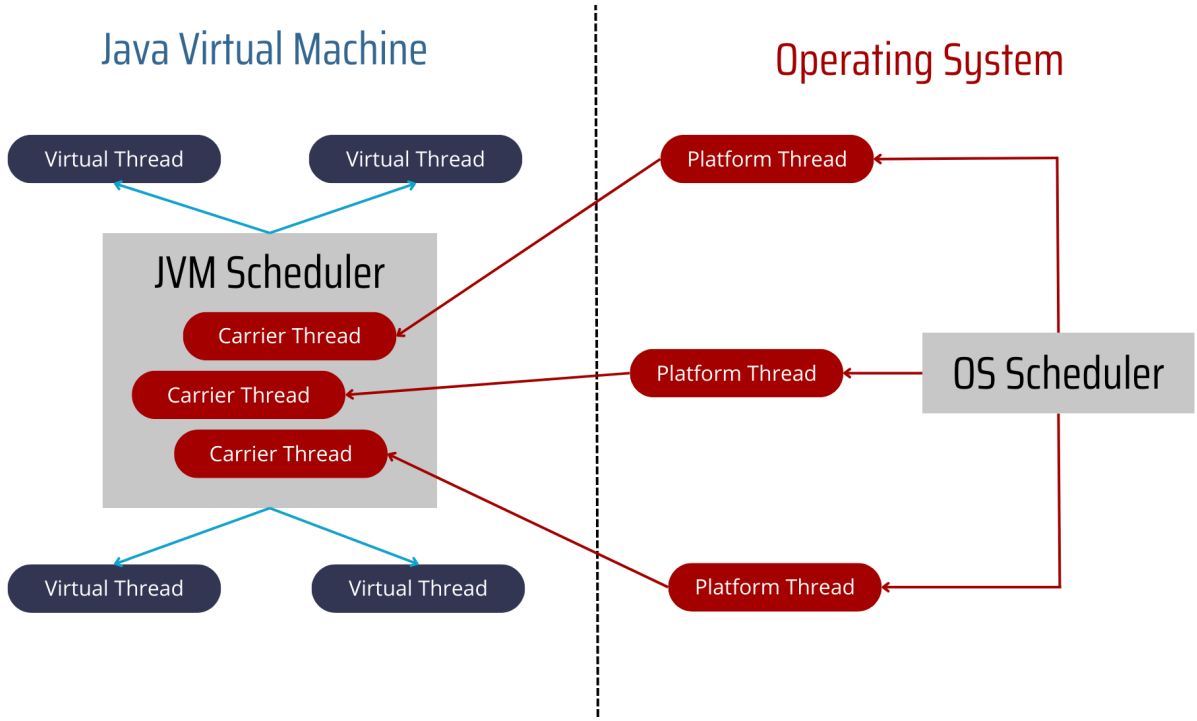
- Wątki w Haskell,
- Goroutines w Go,
- Procesy w Erlang.

Natomiast wraz z wersją JDK21, Java wreszcie wprowadziła własną implementację lekkich wątków i nosi to nazwę *Virtual Threads*.

Jak to działa

Wątki wirtualne są nowym podejściem do wielowątkowości w Javie, które skupia się wokół tego, że wątek wirtualny nie jest zarządzany przez system operacyjny. Zamiast tego to JVM przejmuje to

odpowiedzialne zadanie. Koniec końców, faktyczna praca, jaką wykonuje wątek musi być wykonana przez system operacyjny, natomiast JVM wykorzystuje tzw. *carrier threads*, które służą do przenoszenia wątków wirtualnych na wątki systemu operacyjnego, gdy przyjdzie moment wykonania takiego wątku. Najlepiej będzie zobaczyć to na grafice ☺:



Obraz 10. JVM i Virtual Threads

Wróćmy do naszego uproszczonego modelu działania aplikacji WEB. Jeżeli do tej koncepcji dołożymy wątki, które są lekkie i tanie, możemy spokojnie korzystać z modelu *One Thread per request*. JVM załatwia za Ciebie wykorzystanie zasobów na optymalnym poziomie, a co za tym idzie - możesz prościej pisać aplikacje wielowątkowe, które będą przygotowane na obsługę dużej ilości użytkowników.

Ponieważ wątki wirtualne są tanie, nie występuje tutaj potrzeba reużywania ich, bądź tworzenia *thread pooli*. Każde zadanie do wykonania może być wykonywane w nowym wątku wirtualnym.

JVM Scheduler jest odpowiedzialny za zarządzanie *carrier threads*, wymaga to pewnych założeń, żeby mieć pewność, że wątki wirtualne będą działały jak należy. Zostało to rozwiązane w taki sposób, żeby *carrier thread* był odseparowany od *virtual thread*, który będzie nosił.

Niektóre polskie stwierdzenia są niestety niefortunne. To "noszenie" wzięło się stąd: "separation between carrier thread and any virtual thread it would be carrying" - można to przetłumaczyć w taki sposób, że *carrier thread* będzie odseparowany od *virtual thread*, którym będzie zarządzał.

A co to za założenia?

- *Virtual Thread* nie ma dostępu do *Carrier Thread*,
- Wywołanie `Thread.currentThread()` zwraca *Virtual Thread*,
- Stack trace'y są od siebie odseparowane, a wyjątek wyrzucony w *Virtual Thread* będzie zawierał tylko swoje własne *stack frames*,
- Zmienne *Thread Local* wątku wirtualnego są niedostępne dla *carrier thread* i wzajemnie,

- Patrząc od strony kodu, zależność między *platform thread*, *carrier thread* i *virtual thread* jest niewidoczna. Trochę jak GC, nie pracujesz w kodzie bezpośrednio z GC.

Kod!

Przypomnijmy sobie na początku, w jaki sposób mogliśmy (w klasyczny sposób - *platform threads*) poinstruować Javę, żeby jakaś praca została wykonana w nowym wątku:

```
Runnable someWork = () -> {
    // tutaj coś robimy
};

Thread thread = new Thread(someWork).start();
```

Zanim przejdziemy dalej, warto tutaj zaznaczyć, że razem z *Virtual Threads* w JDK21 zostało wprowadzone nowe API, które pozwala na tworzenia wątków na zasadzie wzorca *Builder*:

```
Runnable someWork = () -> {
    // tutaj coś robimy
};

Thread thread = Thread
    .ofPlatform()
    .name("my-thread-name")
    .start(someWork);
```

Zachęcam do zapoznania się z nowymi metodami i nowymi możliwościami: JDK21 - Thread Class

Przejdźmy teraz do tego, jak można stworzyć wątek wirtualny. Skorzystamy z tego samego nowego API.

```
Runnable someWork = () -> {
    // tutaj coś robimy
};

Thread thread = Thread
    .ofVirtual(someWork)
    .start();
```

Kolejna opcja wygląda tak:

```
Thread thread = Thread.startVirtualThread(() -> {
    // tutaj coś robimy
});
```

Przedstawiona lambda jest implementacją interfejsu `Runnable`.

Do pracy z *virtual threads* możemy również skorzystać z klasy `Executors`:

```
var executorService = Executors.newVirtualThreadPerTaskExecutor();

executorService.submit(() -> {
    // tutaj coś robimy
});
```

Virtual Threads - podsumowanie

Na co warto zwrócić uwagę - pomimo tego, że wprowadzanie tej zmiany daje dużo możliwości, nie musisz przyswajać całego nowego paradygmatu programowania, czy nowego wielkiego API. Dlaczego o tym mówię? Gdy wprowadzono Java 8, doszedł cały nowy paradygmat i ogromne nowe API do nauczenia.

Gdy ogarniasz już, jak pisać aplikacje wielowątkowe, przeskoczenie na *virtual threads* jest relatywnie proste.

W przyszłości zauważysz, że dużo problemów dotyczących wielowątkowości jest rozwiązywanych przez frameworki, z których będziesz korzystać. Zobaczysz, co mam na myśli, gdy zapoznasz się z tematem wielowątkowości w warsztacie o tworzeniu aplikacji WEB (#19).

Polecam wrócić do tego tematu, gdy ogarniesz już warsztat o tworzeniu aplikacji WEB (#19).

Podsumowanie

Na koniec naszych rozważań o wielowątkowości w Javie została jeszcze jedna kwestia. Chociaż może wydawać się to przesadzone, to pamiętaj, że w tym artykule jedynie liznęliśmy tematu. Są to tak naprawdę podstawy... Co prawda jesteś bardziej niż gotowy/-a odpowiadać na typowe pytania rekrutacyjne w tym zakresie, jak i tworzyć wielowątkowe programy, to pamiętaj jednak, że multithreading w Javie to obszar, na temat którego powstają naprawdę pokaźne tomy. Jeśli chcesz wejść głębiej w wielowątkowość, dobrą decyzją będzie jak zawsze: przejrzenie dokumentacji Javy dla klas, o których wspomniałem w tym artykule. Można naprawdę dużo się z nich dowiedzieć.

Z wielowątkowością będziesz mieć tak naprawdę do czynienia podczas tworzenia aplikacji WEB, jednakże wiele rzeczy będą za Ciebie robiły frameworki. Powyższe zagadnienia są jednak istotne i dlatego właśnie zostały tutaj przedstawione. Jak przejdziemy do omówienia aplikacji WEB, to wrócimy do tego tematu.

Jeśli temat Concurrency Cię bardziej zainteresuje, to w internecie i bibliotece znajdziesz niezliczone artykuły i książki na ten temat. Powodzenia!