Wątki

Siłą Javy jest możliwość programowania współbieżnego. Można pisać programy tak, by wykonywały wiele rzeczy jednocześnie. Pojęcie “jednocześnie” może czasem być tutaj mylące gdyż dla maszyny o 1 procesorze wielowątkowość realizowana jest poprzez naprzemienne przełączanie czasu procesora między kilka wątków. Nie można tutaj więc mówić o pracy równoległej. Dla procesorów nowszych faktycznie dzieje się tak, że kilka wątków pracuje równocześnie.
W Javie możemy stworzyć nowe wątki na dwa sposoby:

1. Tworząc klasę dziedziczącą po **Thread**
2. Tworząc klasę implementującą **Runnable**

Druga opcja stworzona jest na wypadek gdy dana klasa już dziedziczy po innej klasie (a jako, że w Javie nie można dziedziczyć po kilku klasach) i zostaje nam opcja z implementacją interfejsu. Ostatecznie wszystko i tak sprowadza się do tego samego. Pracą programisty jest implementacja metody **run(),** która to metoda zostanie odpalona w oddzielnym wątku. Najważniejsze jest to, że implementujemy metodę **run**, ale wątek startujemy metodą **start**. Użycie metody **run** wprost - nie spowoduje startu wątku. Może przykłady:

```
// dziedziczenie po Thread
class Counter1 extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(" Counter : " + i);
        }
        System.out.println(" Counter finished .");
    }

    public static void main(String[] args) {
        Counter1 c = new Counter1();
        System.out.println(" Counter created .");

        c.start();
        System.out.println(" Counter started .");

        System.out.println(" Main finished .");
    }
}

// lub implementacja Runnable
class Counter2 implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(" Counter : " + i);
        }
        System.out.println(" Counter finished .");
    }

    public static void main(String[] args) {
        Counter2 c = new Counter2();
        Thread t = new Thread(c);
        System.out.println(" Counter created .");
```

```
        t.start();
        System.out.println(" Counter started .");

        System.out.println(" Main finished .");
    }
}
```

Oba podejścia są równoważne. To co należy wiedzieć o wątkach, że gdy wątek rozpocznie swoją pracę a następnie metoda **run** się zakończy - wątek umrze. Nie można takiego wątku ponownie wystartować. Należy powołać nowy obiekt którego wystartujemy.

Synchronizacja wątków

Przy pracy równoległej bardzo powszechnym problemem jest współdzielenie zasobów. Gdy wiele wątków zaczyna pisać/czytać do jakiejś zmiennej może pojawić się wiele nieprzewidzianych problemów, które mogą zaskoczyć. Wyobraźmy sobie prosty przykład gdzie wątek działa tak, że odczytuje jakąś zmienną liczbową a następnie wpisuje do niej liczbę o 1 większą. Co jeśli dwa wątki odczytają wartość danej zmiennej jednocześnie a następnie jeden wątek pierwszy zwiększy liczbę (dodając 1 do tej odczytanej) a następnie drugi zamiast zwiększyć liczbę ponownie o 1 - ponownie wpisze tą samą liczbę? Inny poważniejszy przykład: wątek który przed obciążeniem kąta klienta bankowego sprawdza stan konta - a następnie wykonujący przelew gdy saldo jest większe od wartości przelewu. Co się stanie gdy oba wątki odczytają stan konta w podobnym czasie a następnie wykonają przelewy? Klient może skończyć z ujemnym saldem.

Najłatwiejszym rozwiązaniem jest stworzenie w newralgicznych miejscach kodu metody **synchronized**. Jak działa metoda oznaczona jako **synchronized**?

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Oznacza to nic innego, że gdy jakiś wątek wejdzie do takiej metody to zakładana jest blokada na dany obiekt w której ta metoda występuje. Gdy więc jakiś inny wątek będzie chciał wykonać jakąś akcję na obiekcie - to będzie musiał zaczekać. Może i nieco to zwolni wykonywanie całego programu - ale pozbawi program wielu błędów.

Co jeśli jednak mamy taką sytuację gdy jakaś grupa wątków produkuje dane dla innej grupy wątków? Stosowanie prostej zasady **synchronized** może spowodować, że wątki z drugiej grupy próbując pobrać dla siebie zadania - nie zastając ich (pusta lista zadań) będą czekać w metodzie **synchronized** na swoje przydziały - blokują obiekt na tyle skutecznie - że uniemożliwią wrzucenie zadań przez wątki

produkujące zadania. Nastanie klasyczny pat. Na tego typu okazje zostały w klasie **Object** stworzone metody **wait()** oraz **notify()**. Pierwszy z rozkazów działa tak, że wątek zwalnia blokadę i przechodzi do trybu oczekiwania i wznowi swoje działanie gdy jakiś inny wątek wywoła metodę **notify()** informując, że inne wątki mogą spróbować wybudzić się i zająć blokadę.

Brzmi to skomplikowanie? Wielowątkowość to nie jest prosta sprawa. Spójrzmy na klasyczny przykład współpracy kilku klas na przykładzie Producentów i Konsumentów współdzielący jeden zasób (**Resource**). Spróbuj chociaż zrozumieć co dzieje się w kodzie i jak to działa:

```
public class Resource {

    int number; // przechowywana liczba
    boolean producerDone = false; // true gdy producent zakończył pracę
    boolean resourceEmpty = true; // true gdy zasób pusty

    void setDone() {
        producerDone = true;
    }

    // umieszczanie liczby w zasobie
    synchronized void put(int number) {
        if (!resourceEmpty) {        // zasób nie jest pusty
            try {
                System.out.println("Producent czeka...");
                wait();        // czekaj aż zasób będzie pusty
            } catch (InterruptedException e) { }
        }
        // umieść liczbę w zasobie
        this.number = number;
        System.out.println("Liczba " + number + " została wyprodukowana");
        resourceEmpty = false;
        notify();
    }

    // pobranie liczby z zasobu
    synchronized int get() {
        if (resourceEmpty) {      // zasób jest pusty
            try {
                System.out.println("Konsument czeka...");
                wait();            // czekaj aż pojawi się liczba w zasobie
            } catch (InterruptedException e) {
            }
        }

        int number = this.number; // pobierz liczbę z zasobu
        System.out.println("Liczba " + number + " została skonsumowana");
        resourceEmpty = true;
        notify();
        return number;
    }
}

class Consumer implements Runnable {

    Resource resource; // zasób

    Consumer( Resource resource ) {
```

```
        this.resource = resource;
    }

    public void run() {
        int number;
        while(!resource.producerDone) {
            try {
                Thread.sleep( 50 );             // uśpij konsumenta na 50
milisekund
            }
            catch( InterruptedException e )
            {}

            number = resource.get(); // pobierz liczbę z zasobu
        }
    }
}

class Producer implements Runnable
{
    Resource resource; /** zasób */

    Producer( Resource resource ) {
        this.resource = resource;
    }

    public void run() {
        for( int i=1; i<=5; i++ ) // umieszczanie liczb w zasobie
            resource.put( i );

        // koniec pracy producenta
        resource.setDone();
        System.out.println( "Producent zakończył pracę.");
    }
}

public class PCProblem
{
    // utwórz zasób
    static Resource resource = new Resource();

    /** główna metoda aplikacji */
    public static void main(String args[]) {
        // utwórz producenta i konsumenta
        new Thread(new Producer( resource )).start();
        new Thread(new Consumer( resource )).start();
    }
}
```

Podstawy wątków

Cechy o których musimy mieć pojęcie:

- Specyfikacja Javy definiuje mechanizm który umożliwia współbieżne wykonywanie operacji na jednym procesorze fizycznym

- Jako, że program Javy nie wykonuje się bezpośrednio na systemie a na Wirtualnej Maszynie Javy stąd system operacyjny i JVM odpowiednio planują wykonanie operacji, aby zasymulować równoległość wykonania
- Sam sposób wykonania powyższego - nie jest zawarty w specyfikacji i jest zależne od aktualnej implementacji JVM
- Każdy wątek zawiera swój własny stos wywołań (niezależnie od innych wątków)
- Wątki mogą współdzielić zmienne między sobą jak również posiadać własne zmienne tylko dla siebie
- Dostęp do współdzielonych zmiennych bardzo często musi być synchronizowany
- Każdy wątek z Javie jest obiektem klasy **Thread**, operacje wykonywane w wątku muszą być zawarte w metodzie **void run()**
- Utworzenie nowego wątku może odbyć się na dwa sposoby: dziedziczenie po **Thread,** implementację interfejsu **Runnable**

  Przykłady:

```
public class Example {

    // dziedziczenie po Thread
    static class MyThread extends Thread {
        public void run() {
            System.out.println("No cześć. Jestem wątek " +
Thread.currentThread().getName());
        }
    }

    // implementacja Runnable
    static class MyThread2 implements Runnable {
        public void run() {
            System.out.println("No cześć. Jestem wątek " +
Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();

        Thread myThread2 = new Thread(new MyThread2());
        myThread2.start();
    }
}
```

  Cykl życia wątku jest następujący:

- **NEW** -- gdy tworzymy obiekt typu **Thread** (wątek nieaktywny),
- **RUNNABLE** -- po wywołaniu metody **start()** wątek jest aktywny i oczekuje na przydzielenie czasu procesora (wątek aktywny),
- **RUNNING** -- procesor przydzielił czas, trwa wykonywanie wątku (wątek aktywny),
- **SLEEPING/WAITING/BLOCKED** -- wątek jest aktywny -- ale w tej chwili nie jest uruchomiony (wątek aktywny),
- **DEAD –** metoda **run()** dobiegła końca, bądź wyjątek zakończył działanie wątku (wątek nieaktywny).

  To czy wątek jest aktywny bądź nie - można sprawdzić za pomocą metody **isAlive()** klasy **Thread**. Należy zauważyć, że pomimo, że wątek może się zakończyć to nadal pozostaje prawidłowym obiektem

z punktu widzenia Javy. Nie można jednak danego wątku znów wystartować (np. przy pomocy metody start(), która wcześniej ten wątek wystartowała).

Jak to się dzieje, że dany wątek się wykonuje bądź nie? Od tego jest stworzony odpowiedni mechanizm zwany **Schedulerem**, który decyduje, któremu wątkowi przydzielić czas procesora, na jak długo i czy w ogóle. Implementacja Schedulera jest zmieniana stąd nie mamy nigdy gwarancji, że wątki będą się zachowywać dokładnie tak jak byśmy tego chcieli (kolejność uruchamiania wątków, czas przydziału procesora itp.). Możemy jednak mieć wpływ na działanie wątków poprzez kilka mechanizmów:

- Przydzielanie Priorytetów wątkom (metoda **setPriority(int priority)** w klasie **Thread**). Wtedy można założyć, że **Scheduler** wybierający jakiś wątek do uruchomienia wybierze wątek z priorytetem nie mniejszym niż jakikolwiek inny oczekujący w puli).
- Poprzez metodę **static sleep(long ms)** -- wstrzymującą działanie aktualnego wątku na wskazany czas
- Poprzez metodę**static yield()** -– umożliwienie innemu wątkowi o tym samym/wyższym priorytecie zająć czas procesora.
- Poprzez metodę **join()** -- powodującą zatrzymanie aktualnego wątku do czasu zakończenia działania wątku na rzecz którego wywołaliśmy metodę **join**.

Przykład:

```
public class Processor extends Thread {

    public void run() {
        // wykonuję ciężkie obliczenia
    }


    public static void main(String[] args) {
        Processor t1 = new Processor();
        t1.start();

        Processor t2 = new Processor();
        t2.start();

        Processor t3 = new Processor();
        t3.start();

        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // prezentuj wyniki...
    }
}
```

Podczas działania na wątkach może się zdarzyć, że wątek rzuci wyjątkiem: **InterruptedException** jest on rzucany i wymaga obsłużenia gdy np. wątek zostanie zatrzymany gdy jest w stanie oczekiwania na jakieś zdarzenie (np. jest w fazie **sleep**). Wymaga to obsłużenia gdy wątek wykonuje metodę **sleep()** oraz **join()** klasy **Thread** oraz **wait()** klasy **Object**.

Synchronizacja

Często jest tak, że wątki pracują na tych samych danych. Może to prowadzić do bardzo niebezpiecznych zachowań, których nie będziemy w stanie często przewidzieć. Co się stanie gdy małżonkowie zalogują się na wspólne konto bankowe i w tym samym momencie będą chcieli puścić przelew na znaczną sumę? Każda akcja wywoła uruchomienie wątku próbującego wykonać przelew. Jeden wątek sprawdzi saldo konta i sprawdzi czy przelew może zostać wykonany. Jeśli suma jest wystarczająca - wykona przelew. Co jeśli jednak po tym jak jeden wątek sprawdził saldo ale jeszcze przed wypuszczeniem przelewu - inny wątek sprawdzi saldo by sprawdzić możliwość wykonania przelewu? W skrajnej sytuacji obojgu małżonkom udałoby się wykonać przelew i przez przypadek wejść na pokaźny debet.

Aby przeciwdziałać tego typu sytuacjom - wprowadzono mechanizm synchronizacji, który blokuje wykonanie danej metody do czasu aż inny wątek nie zwolni blokady. Dzięki temu - nie jest możliwe aby zdarzyła się sytuacja jak wyżej. Należy starać się, aby metody, które odwołują się do współdzielonych zasobów - były synchronizowane. Wejdźmy jednak w szczegóły:

```
public class BankAccount {
    private int balance = 1000;

    public synchronized void deposit(int amount) {
        balance += amount;
    }

    public synchronized void windraw(int amount) {
        if(balance + amount > 0){
            balance -= amount;
        }
    }

    public synchronized int value() {
        return balance;
    }
}
```

Jeżeli jakikolwiek wątek wejdzie do metody synchronizowanej to zakładana jest blokada na danym obiekcie na którym wykonano daną metodę. Dokładniej - mówi się wtedy, że dany wątek staje się właścicielem monitora (obiektu, który blokuje wtedy inne wątki przed wywołaniem dowolnej innej metody synchronizowanej monitora). Każdy inny wątek musi poczekać wtedy na zwolnienie monitora aby móc wejść do metody.

Kilka rad:

- Metody odwołujące się do danych statycznych powinny być zadeklarowane jako synchronizowane metody statyczne
- Metody odwołujące się do niestatycznych składników klasy powinny być zadeklarowane jako synchronizowane metody niestatyczne
- Najlepiej unikać odwołań jednocześnie do danych statycznych i niestatycznych w ramach jednej metody, która powinna być synchronizowana

Bloki synchronizowane

Nie zawsze jest tak, że należy blokować wszystkie metody (synchronizowane) danej klasy tylko dlatego, że jakiś wątek wszedł do jednej z metod synchronizowanych. Powoduje to czasem niepotrzebne zwolnienie działania programu. Gdy jakiś wątek wykonuje operację przelewu w systemie bankowym, nie ma potrzeby blokować dostępu innym wątkom do innych operacji systemu bankowego dopóki nie będzie to zagrażało stabilności systemu (np. przelew na całkowicie inne konto raczej powinien być dozwolony itp).

W celu optymalizacji dostępu wątków wprowadzono bloki synchronizowane. Zasada działania takiego bloku jest podobna do metody synchronizowanej z tą różnicą, że można tutaj wyróżnić element na którym zakładana jest blokada.

Przykład:

```
public class BankAccount {

    private static Integer ID = 101;

    private Integer balance = 1000;

    private Object writeLock = new Object();

    public void deposit(int amount) {
        synchronized (balance){
            balance += amount;
        }

    }

    public void windraw(int amount) {
        synchronized (this) { // blokada na całym obiekcie
            if (balance + amount > 0) {
                balance -= amount;
            }
        }
    }

     public static void setID(Integer id){
        synchronized (BankAccount.class){ // blokada na całej klasie (wszystkich
instancjach tej klasy)
            ID = id;
        }
    }

    public void setValue(int value) {
         synchronized (writeLock){ // blokada na jakimś obiekcie
             balance = value;
         }
    }
}
```

Przykład powyżej pokazuje wszystkie możliwe sposoby zakładania blokad na obiektach. Przypadek **synchronized (this) {...}** w zasadzie jest równoważny napisaniu **synchronized** na całej metodzie. Należy zauważyć, że blok synchronizowany nie musi rozpoczynać się na początku metody - może rozpocząć się np. na końcu metody i zajmować raptem kilka linijek. Powinniśmy minimalizować obszar

synchronizacji do minimum aby wątki nie były niepotrzebnie blokowane gdy nie muszą. Należy zauważyć, że obiekt **writeLock** w powyższym przykładzie może być wspólny dla kilku różnych klas (np. przekazany przez konstruktor). Wtedy może on pełnić rolę tej samej blokady w różnych klasach i różnych metodach.

Aby tego było mało na obiekcie, który służy do blokowania można wykonywać odpowiednie metody, które mogą zmieniać dostęp wątków do niego. Chodzi tu o metody:

- **wait()** -- blokuje działanie aktualnego wątku do czasu wysłania sygnału przez inny wątek,
- **notify()** -- uruchamia jeden z zablokowanych na danym obiekcie wątków,
- **notifyAll()** -- uruchamia wszystkie wątki.

Przykład:

```
public class NotifyExample {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try{
                System.out.println("Czekam na zakończenie wątku b...");
                b.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            System.out.println("Suma to: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total;

    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += i;
            }
            notify();
        }
    }
}
```

W przykładzie powyżej wykorzystałem jako element blokujący -- obiekt wątku. Nikt nie zabroni mi tego robić -- jest on takim samym dobrym obiektem jak inny. Mogę równie dobrze wykorzystać jakiś wspólny obiekt (np. **writeLock** z poprzedniego przykładu) - ważne aby **wait()** oraz **notify()** wykonywane było na tym samym obiekcie -- inaczej mechanizm blokad nie będzie działał tak jak powinien.

Przypadki szczególne

Są różne specyficzne przypadki, które mogą wystąpić w wątkach. Często wynikają one z błędów programisty i należy mieć świadomość, że dana sytuacja może wystąpić. Takim przypadkiem jest zakleszczenie wątków, kiedy to wątki będą oczekiwać nawzajem na zwolnienie monitora i w konsekwencji program nigdy się nie skończy.

```
public class BankAccount {

    private Integer id;

    public BankAccount(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return this.id;
    }

    public synchronized void borrow(BankAccount bower) {
        System.out.println(this.getId() + " pożycza " + bower.getId());
        // a następnie oddaję kasę
        bower.returnMoney(this);
    }

    public synchronized void returnMoney(BankAccount bower) {
        System.out.println(bower.getId() + " oddaje " + bower.getId());
    }


    public static void main(String[] args) {
        BankAccount oneAccount = new BankAccount(1);
        BankAccount secondAccount = new BankAccount(2);

        new Thread(new Runnable() {
            public void run() {
                oneAccount.borrow(secondAccount);
            }
        }).start();

        new Thread(new Runnable() {
            public void run() {
                secondAccount.borrow(oneAccount);
            }
        }).start();
    }
}
```

Oczywiście ten przypadek jest jest dość trywialny. Może jednak zajść sytuacja kiedy to wątek A czeka na B, B na C, C na D zaś D na A. Wtedy już programista może mieć problem ze zrozumieniem co jest nie tak.

Innym problemem może być zagłodzenie wątku. Dochodzi do niej w sytuacji gdy jakiś wątek (bądź grupa wątków) nie może pracować bo wiecznie czeka na zwolnienie monitora bo inną wątek zawłaszczył go sobie. Innym przykładem jest sytuacja gdy wątek ma tak mały priorytet, że zawsze Scheduler wybierze z puli wątków inny - tym samym wątek nigdy nie doczeka się na czas procesora.

Przykład:

```
public class ZaglodzenieExample implements Runnable {

    private final Object resource;
    private final String message;

    public static void main(String[] args) {
        final int cpus = Runtime.getRuntime().availableProcessors(); // ilość
dostępnych procesorów
        final int runners = cpus * 2;
        final Object resource = new Object();

        // create sample runners and start them
        for (int i = 1; i <= runners; i++) {
            (new Thread(new ZaglodzenieExample(resource,
String.valueOf(i)))).start();
        }

        // zawieś wątek main
        synchronized (ZaglodzenieExample.class) {
            try {
                ZaglodzenieExample.class.wait();
            } catch (InterruptedException ignored) {
            }
        }
    }

    public ZaglodzenieExample(Object resource, String message) {
        this.resource = resource;
        this.message = message;
    }

    public void run()
    {
        synchronized (this) {
            while (true) {
                synchronized (resource) {
                    print(message);
                    try {
                        this.wait(100); // zagłodzenie
//                        resource.wait(100); // bez zagłodzenia
                    } catch (InterruptedException ignored) {
                    }
                }
            }
        }
    }

    private static void print(String str) {
       System.out.print(str);
    }
}
```

Zadania asynchroniczne

Dość łatwym podejściem do wielowątkowości jest stosowane interfejsu **Callable** oraz ich związek z tzw. **Executorami**. Tworzymy klasę implementującą interfejs **Callable** mając możliwość wyspecyfikowania typu wyniku jaki dane zadanie zwraca (np. **Callable<String>**). Gdy owe zadanie stworzymy -- wrzucamy je do **Exacutora**, który to przydzieli jakiś wątek na to zadanie i rozpocznie obliczenia. **Executor** nie zwróci nam wyniku ale “przyszły wynik” w postaci obiektu **Future<typWynikuCallable>** np. **Future<String>**. Zadanie wykonuje się już w tle a my możemy sprawdzać status wykonania zadania (metoda **isDone()**), możemy zniszczyć zadanie (metoda **cancell()**) lub po prostu poczekać na wynik (metoda **get()**) albo nawet poczekać na wynik dając maksymalny czas na oczekiwanie na wynik (**get(int timeInMS**)) gdzie potem poleci wyjątek.
Prosty przykład:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class CallableExample {

    public static void main(String args[]){

        // Exacutor z liczbą wątków 10
        ExecutorService executor = Executors.newFixedThreadPool(10);
        // Lista przetrzymująca obiekty wyników klas typu Callable
        List<Future<Integer>> taskList = new ArrayList<>();

        for(int i=0; i< 100; i++){

            final int start = i;

            // klasa anonimowa :)
            Callable<Integer> callable = new Callable<Integer>(){

                @Override
                public Integer call() throws Exception {
                    return start * 1000;
                }
            };

            // wrzucamy zadanie do Exacutora a otrzymujemy obietnicę wyniku
            Future<Integer> futureTask = executor.submit(callable);
            // dodajmy przyszły wynik do listy
            taskList.add(futureTask);
        }
        for(Future<Integer> future : taskList){
            try {
                // poczekajmy na wynik:
                System.out.println(future.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
        //shutdown exacutor
        executor.shutdown();
```

```
    }
}
```

Obsługa wątków w ten sposób jest dość prosta i łatwa do zrozumienia.

Chcesz wiedzieć więcej? Zachęcam do zgłębiania wiedzy o wątkach:

Stosowanie lock'ów:

- http://tutorials.jenkov.com/java-concurrency/read-write-locks.html
- https://www.journaldev.com/2377/java-lock-example-reentrantlock