# Wielowątkowość

## Wstęp

Proces to każdy działający program w pamięci. Proces ma kawałek pamięci dostępny tylko dla siebie. W każdym procesie istnieje co najmniej jeden wątek zwany wątkiem głównym. Z tego wątku możemy tworzyć wątki poboczne. Wszystkie wątki w obrębie jednego procesu współdzielą ze sobą pamięć. Dzięki temu komunikacja między wątkami jest bardzo szybka a to współdzielenie pamięci jest źródłem wszystkich problemów związanym z programowaniem wielowątkowym.

## Rozszerzenie klasy Thread

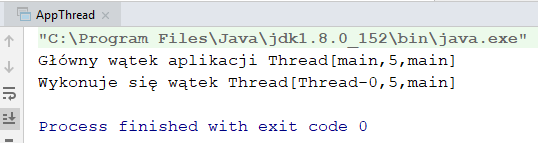
//MyThread

Własna klasa wątku może wyglądać tak

public class MyThread extends Thread {  
 @Override  
 public void run(){  
 IntStream.*rangeClosed*(0,100)  
 .forEach(i->System.*out*.println(i+" running "+MyThread.*currentThread*().getName()));  
 }  
  
 public MyThread(String name) {  
 super(name);  
 }  
}

a tak uruchamia się wątek

Thread mythread=new MyThread("My Thread");  
mythread.start();



Nazwy main i Thread-0 otrzymamy jeśli wyświetlanie nazwy wątku zapiszemy jako

MyThread.*currentThread*().getName());

### A jak ustalić własną nazwę dla wątku?

Należy napisać konstruktor i odpowiednio utworzyć obiekt naszego wątku

**public** MyThread(String name) {  
 **super**(name);  
}

Thread thread=**new** MyThread(**"mój wątek"**);

### Metoda start i run

Metoda **start** uruchamia nowy wątek i uruchamia metodę **run**. Gdybyśmy uruchomili metodę run dla wątku, to nasz kod **nie wykona się w osobnym wątku, ale w wątku głównym aplikacji**.

## Implementacja interfejsu Runnable

// MyRunnable

public class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 IntStream.*rangeClosed*(0,100)  
 .forEach(i->System.*out*.println(i+" Wykonuje się "+MyThread.*currentThread*().getName()));  
 }  
}

Runnable runnable1=new MyRunnable();  
Thread thread3=new Thread(runnable1,"Thread number 3");

thread3.start();

### Czy trzeba tworzyć osobną klasę MyRunnable?

// MyThreadMain

Nie, wykorzystamy anonimową klasę.

Runnable runnable2=new Runnable() {  
 @Override  
 public void run() {  
 IntStream.*rangeClosed*(0,100)  
 .forEach(i->System.*out*.println(i+" running "+MyThread.*currentThread*().getName()));  
 }  
};

I ten obiekt (runnable2) można teraz przekazać jako parametr do inicjalizacji Threada.

Thread thread4=new Thread(runnable2,"Thread number 4");

Ten sposób wstrzykiwania runnable2 do konstruktora nazywa się **wzorcem strategii**.

### Można jeszcze wszystko uprościć stosując wyrażenie lambda.

Thread thread5=new Thread(()->IntStream.*rangeClosed*(0,100)  
 .forEach(i->System.*out*.println(i+" running "+MyThread.*currentThread*().getName())),"Thread number 5");

### extends Thread czy implements Runnable?

Lepiej jest stosować implementację interfejsu Runnable, bo

1. Nie zamykamy sobie drogi do dziedziczenia
2. Można stosować lambdę (mało kodu i przejrzystość)

## Thread.sleep(czas)

Usypia wątek na tyle milisekund ile podamy w nawiasie. Jednak nie ma pewności, że po tym czasie wątek wróci do działania. Ten czas może być większy, gdy procesor jest czymś zajęty.

Zamiast sleep() można użyć

TimeUnit.***SECONDS***.sleep(1);

(Wprowadzono to od Javy 1.8)

## Join()

Aktualny wątek czeka na wątek na którym uruchomimy tę metodę.

thread4.start();  
try {  
 thread4.join();  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}  
thread5.start();

**thread5 poczeka zatem na zakończenie thread4**

W join można podać maksymalną liczbę ms przez którą będzie czekał następny wątek na zakończenie pierwszego wątku.

## Executor Service

Jest to Framework zarządzający wielką liczbą wątków a nie kilkoma ;)

// ExecutorMain

public static void main(String[] args) {  
 //statyczna metoda fabrykująca  
 ExecutorService executorService= Executors.*newSingleThreadExecutor*();  
 Runnable runnable=()-> IntStream.*rangeClosed*(0,100)  
 .forEach(i->System.*out*.println(i+" running "+MyThread.*currentThread*().getName()));  
 executorService.submit(runnable);  
 executorService.shutdown();  
}

Jak widzimy po sposobie tworzenia executora pula wątków ma miejsce tylko na jeden wątek. (**SingleThreadExecutor**) Dlatego też, gdybyśmy dodali drugi wątek do puli, to musiałby on poczekać na zakończenie poprzedniego. Ponadto program nie kończy się automatycznie, bo pula czeka na nowe zadania.

**Do zamykania executora** mamy dwie metody:

**shutdown()** (czeka na zamknięcie wszystkich przesłanych wątków do puli)

**shutdownNow()** (natychmiast kończy wszystkie wątki) //może wywołać wyjątek

Zamiast używać  **SingleThreadExecutor** można użyć **newFixedThreadPool**,

// FixedExecutorMain

gdzie ustawiamy wielkość puli np. na 2. Oznacza to, że max dwa wątki mogą działać jednocześnie.

ExecutorService executorService= Executors.*newFixedThreadPool*(2);  
executorService.submit(worker1);  
executorService.submit(worker2);  
executorService.submit(worker3);  
executorService.shutdown();

Jeśli jest więcej wątków niż jest w stanie **Executor** obsłużyć, to czekają one w kolejce, a gdy kolejka jest zbyt duża, to następne wątki nie są już wpuszczane do kolejki. Należy pamiętać, że w puli wątków w executorze są już utworzone wątki i czekają one na zapełnienie.

### ScheduledThreadPool

Pozwala na ustawienia opóźnienia zadania albo na ustawienie zadania wykonywanego cyklicznie.

// ScheduledExecutorMain

#### Uruchomienie z opóźnieniem

ScheduledExecutorService executorService1= Executors.*newScheduledThreadPool*(2);  
executorService1.schedule(worker1,5,TimeUnit.*SECONDS*);  
executorService1.shutdown();

#### Uruchamianie zadanie ustawionego cyklicznie

ScheduledExecutorService executorService2=Executors.*newScheduledThreadPool*(2);  
executorService2.scheduleAtFixedRate(worker2,0,3,TimeUnit.*SECONDS*);

## Callable i Future

**Callable jest podobne do Runnable ale jest lepszy bo zwraca wynik**.

// CollableMain

Callable<Integer> thread = () -> {  
 TimeUnit.*SECONDS*.sleep(5);  
 return 44;  
};  
ExecutorService executor = Executors.*newFixedThreadPool*(2);  
Future<Integer> result = executor.submit(thread);  
executor.shutdown();  
System.*out*.println("Wynik " + result.get());

Klasa Future odbiera wynik ;)

Należy pamiętać, że operacja result.get() blokuje wykonywania głównego wątku do momentu otrzymania wyniku.

**Future nie jest więc idealnym rozwiązaniem bo blokuje główny wątek aplikacji.**

### Jak więc szybciej otrzymać wynik

1. result.isDone() czy jest już wynik
2. result.get(4,TIMEUNITS.SECONDS) – czekamy 4 s na wynik, jeśli go nie będzie to będzie wyjątek

### invokeAll(), invokeAny()

**invokeAll()**

// CollableInvokeMain

działa na kolekcji Callable. Listę obiektów typu Collable przekazujemy jako parametr meotdy. Wynikiem jest Lista obiektów klasy Future.

List<Callable<Integer>> list= Arrays.*asList*(thread1,thread2,thread3);  
ExecutorService executor= Executors.*newFixedThreadPool*(2);  
List<Future<Integer>> futures = executor.invokeAll(list);  
for (Future<Integer> element:futures  
) {  
 System.*out*.println(element.get());  
}

Wyniki zostaną wyświetlone dopiero wtedy gdy wszystkie wątki się zakończą

**invokeAny()**

Integer integer=executor.invokeAny(list);

czyli zwrócona zostanie wartość z najszybciej wykonanego Collable.

## CompletableFuture

To następca Collable i Future

// CompletableFutureMain

**Klasa Future** wady:

1. get() blokuje wykonanie wątku z którego został wywołany.
2. Brakuje możliwości wywołań Future w łańcuch i obsługi błędów w tym łańcuchu

### runAsync()

runAsync() nie zwraca wyniku

CompletableFuture.*runAsync*(  
 ()-> System.*out*.println(Thread.*currentThread*().getName()),executor  
);

Wyniki: Wątek **ForkJoinPool.**commonPool-worker-1

### supplyAsync()

Podobne do runAsync(), ale w tym przypadku możemy pobrać wartość

CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 44;

},executor)

A jak się dobrać do wyniku? Podstawić poprzednie wyrażenia do zmiennej.

CompletableFuture<Integer> integer=CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 44;  
});  
//blocker  
System.*out*.println(integer.get());

**Niestety get() jest znowu blokujące.**

Oba sposoby asynchroniczne można uruchomić w znanym nam executorze

CompletableFuture<Integer> integer2=CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 44;  
},executor);

### Jak zrobić wywołanie łańcuchowe, nieblokujące CompletableFuture?

Aby nie blokować głównego wątku main pobraną wartością, należy przenieść CompetableFuture do osobnego wątku

**thenApply()** – metoda pozwala na modyfikację wyniku (zwraca obiekt CompetableFuture)

**thenAccept()** – konsumuje wynik

CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 44;  
}, executor).exceptionally(exception -> {  
 System.*out*.println("Error");  
 return 2;  
})  
 .thenApply(r -> {  
 System.*out*.println(Thread.*currentThread*().getName());  
 return r \* 3;  
 })  
 .thenAccept(r -> {  
 System.*out*.println(Thread.*currentThread*().getName());  
 System.*out*.println(r);  
 });

### Łączenie ze sobą CompletableFuture

// CompletableFutureMergingCompose

**ZALEŻNE thenCompose()**

public static void main(String[] args) throws ExecutionException, InterruptedException {  
 CompletableFuture<Long> idFuture = CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return *getUserId*();  
 });  
  
 CompletableFuture<Void> future = idFuture  
 .thenCompose(id -> CompletableFuture.*supplyAsync*((() -> *getDiscount*(id)))  
 .thenAccept(System.*out*::println));  
 future.get();  
   
}  
  
public static Long getUserId() {  
 return 144L;  
}  
  
public static Double getDiscount(Long id) {  
 return 1.4;  
}

**NIEZALEŻNE thenCombine()**

// CompletableFutureMergingCombine

Przykład na przemnożenie dwóch liczb zwracanych przez każdy **CompletableFuture**

CompletableFuture<Long> future1 = CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 100L;  
});  
  
CompletableFuture<Long> future2 = CompletableFuture.*supplyAsync*(() -> {  
 try {  
 TimeUnit.*SECONDS*.sleep(5);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return 200L;  
});  
  
CompletableFuture<Long> future = future1.thenCombine(future2, (aLong, aLong2) ->  
 aLong \* aLong2);  
Long result = future.get();  
System.*out*.println(result);

### Obsługa błędów

},executor**).exceptionally(exception->{  
 System.*out*.println("Błąd");  
 return 2;  
})** .thenApply(r->{

## Race Condition

Wyścig tzn wynik całego programu zależy od kolejności wykonywania wątków w celu dotarcia do miejsca w pamięci.

executor.shutdown();

executor.awaitTermination(1, TimeUnit.***MINUTES***);

poczekaj minutę a potem zamknij wątki

## Synchronized

Ograniczenie dostępu do sekcji krytycznej

**synchronized public void** increase(){  
 **count**=count+1;  
}

Wady: długi czas działania, brak asynchroniczności

Lepiej zrobić tak

public void increase(){  
 synchronized (this){  
 count=count+1;  
 }  
}

bo przecież nie cały czas kod metody musi być chroniony

//Counter

// CounterMain

ExecutorService executor= Executors.*newFixedThreadPool*(10);  
Counter counter=new Counter();  
for (int i = 0; i < 100; i++) {  
 executor.submit(counter::increase);  
}  
executor.shutdown();  
executor.awaitTermination(1, TimeUnit.*MINUTES*);  
System.*out*.println(counter.getCount());

## Atomowe zmienne

// AtomicCounter

// AtomicCounterMain

Metody atomowe są zamknięte w synchronicznym bloku ale działa o wiele szybciej niż w bloku sychronized.

**Wada:** operujemy na obiektach a nie prymitywach.

public class AtomicCounter {  
 private AtomicInteger count=new AtomicInteger(0);  
 public void increase(){  
 count.getAndIncrement();  
 }  
 public int getCount(){  
 return count.get();  
 }  
}