PWSZ – Nowy Sącz Programowanie Współbieżne i Rozproszone (Projekt) - 2020/2021	
PWiR_07	
Imię i nazwisko: Filip Rzepiela	**Grupa:** L3
Data: 21.04.2021	**Ocena:**

Java.util.concurrency

Example1

Pokazuje w jaki sposób utworzyć kilka wątków. Można zaobserwować że kolejność wykonywania jest niedeterministyczna.

Example2

Przykład demonstruje użycie bariery.

Example3

Pokazuje w jaki sposób kontrolować kolejność uruchamiania się wątków, zademonstrowanie oczekiwania na śmierć wątków.

Example4

Przykład pokazujący jak zabić wątki w czasie ich trwania z wykorzystaniem przerwania.

Zadanie:

1. Wykonaj sprawozdanie, opisz wymienione poniżej funkcje/klasy/interfejsy paczki java.util.concurrency odpowiadające za zrównoleglenie, użyte w przykładach.

   - interfejs Runnable

   - interfejs Callable<T>

   - klasa Executor (w tym metoda newFixedThreadPool())

   - klasa ExecutorService (szczególnie metody shutdown(), shutdownNow() , awaitTermination(), isTerminated())

   - klasa FutureTask<T> i jej metoda T

   - metody Thread.sleep(), Thread.yield(), <<Thread Object>>.join()

   - funkcje System.currentTimeMillis()

   - czym różni się Catch(Exception e) od Catch(InterruptException e)

Wyniki oraz program prześlij do swojego repozytorium. Umieść je w folderze o tej samie nazwie co ten PDF.

interfejs Runnable

```
private int id;



public MyRun(int id)
        { this.id = id;

}



@Override public
void run() {

        while(true) {
                System.out.println("Watek "+id);
                try {

                        //usypiamy wątek na 100 milisekund

                        Thread.sleep(100);

                } catch (InterruptedException e) {

                        e.printStackTrace();}}}}
```

Jak widać interfejs Runnable posiada jedną metodę, którą musimy zaimplementować- run(). Wszystko co się w niej znajduje zostanie wykonane po uruchomieniu wątku, do którego przekażemy obiekt klasy MyRun. My utworzyliśmy wewnątrz niej nieskończoną pętlę, której zadaniem jest wyświetlanie ID wątku, a następnie wstrzymanie działania na 100 milisekund za pomocą statycznej metody sleep().

- **Interfejs Callable**- Jest to kolejny sposób utworzenia wątku w języku Java. Bardzo podobny do Runnable, z tą różnicą że Callable może zwrócić w wyniku jakąś określoną wartość czego nie może zrobić Runnable. W stosunku do interfejsu runnable są to dwa ważne udogodnienia: zwrot wyniku oraz możliwość zgłoszenia wyjątku kontrolowanego.

- **Klasa Executor (w tym metoda newFixedThreadPool()) -** Aby ułatwić tworzenie wątków, od Javy 1.5 pojawiła się klasa Executors, która posiada metody statyczne do tworzeni puli wątków. Executory pomagają zarządzać wątkami w wygodny sposób. Klasa Executor dostarcza statycznych metod fabrycznych za pomocą których możemy utworzyć żądany executor.

  newFixedThreadPool
  public static ExecutorService newFixedThreadPool (int nThreads,
  ThreadFactory threadFactory)

Tworzy pulę wątków, która ponownie wykorzystuje stałą liczbę wątków działających poza udostępnioną nieograniczoną kolejką, używając dostarczonej ThreadFactory do tworzenia nowych wątków w razie potrzeby. W dowolnym momencie co

najwyżej nThreads będzie aktywnymi zadaniami przetwarzania. Jeśli dodatkowe zadania zostaną przesłane, gdy wszystkie wątki będą aktywne, będą czekać w kolejce, aż wątek będzie dostępny. Jeśli jakikolwiek wątek zostanie zakończony z powodu awarii podczas wykonywania przed zamknięciem, nowy wątek zajmie jego miejsce, jeśli będzie to konieczne do wykonania kolejnych zadań. Wątki w puli będą istnieć, dopóki nie zostaną jawnie podane shutdown.

- **Klasa ExecutorService (szczególnie metody shutdown(), shutdownNow() , awaitTermination(), isTerminated())**

  Klasa ExecutorService to swego rodzaju rozszerzenie klasy Executor o dodatkowe metody pozwalające na np. kończenie pracy wykonawców. Jego metodami są:
  - **shutdown()** – metoda dezaktywująca wykonawcę
  - **shutdownNow()** – metoda zatrzymująca wszystkie wykonywane zadania i anuluje oczekujące
  - **awaitTermination(long timeout, TimeUnit unit)** – dezaktywuje wykonawcę i oczekuje na zakończenie wykonania zleconych zadań w określonym czasie
  - **isTerminated()** – jeżeli shutdown() zadziałało to zwraca wartość true

- **Klasa FutureTask i jej metoda T** - Jest to implementacja interfejsu

  Future, umożliwia ona uruchomienie pojedynczego zadania w osobnym wątku, pozwala też podanie obiektu Callable do Execute

- **Metody Thread.sleep(), Thread.yield(), <>.join()**
  - **Thread.sleep()** – metoda statyczna sprawiająca, że bieżący wątek zasypia na określony czas
  - **Thread.yield()** - zapewnia mechanizm informujący „planującego", że bieżący wątek jest skłonny zrezygnować z obecnego wykorzystania procesora, ale chciałby, aby został zaplanowany z powrotem tak szybko, jak to możliwe.
  - **join()** – metoda oczekiwania na zamknięcie wątku

- **Funkcje System.currentTimeMillis()**

  Zwraca bieżący czas w milisekundach. Jednostką czasu zwracanej wartości jest milisekunda, a stopień szczegółowości wartości zależy od systemu operacyjnego i może być większy.

  Na przykład wiele systemów operacyjnych mierzy czas w dziesiątkach milisekund.
  Deklaracja: Public static long currentTimeMillis()

- **Czym różni się Catch(Exception e) od Catch(InterruptException**

  Zestawienie pozwala zdefiniować blok kodu do wykonania, w przypadku wystąpienia błędu w bloku try.