Przetwarzanie współbieżne i równoległego

PRZYKŁADOWY KOD: https://gitlab.com/examples4java/kurs2

a) przetwarzanie współbieżne
Wszystkie programy komputerowe wykonywane w konwencjonalny sposób działają sekwencyjnie. Oznacza to, że kod wykonywany jest linia po linii. W tym wypadku komputer (ściśle mówiąc mikroprocesor) zachowuje się jak człowiek czytający książkę – słowo za słowem, linia za linią, strona za stroną.

INFORMACJA: Tutaj, dla ścisłości, należy wspomnieć, że programów czysto sekwencyjnych de facto nie ma. Programy wykonują się sekwencyjnie przeważnie do chwili napotkania:
- jakiegokolwiek przerwania (źródło dowolne)
- skoki warunkowe i bezwarunkowe (wywoływane w różny sposób w zależności od sprzętu i/lub języka programowania)
W wypadku wystąpienia któregokolwiek z czynników następuje przerwanie wykonywania sekwencji, przeskok do podanej funkcji, wykonanie jej, po czym powrót do głównego kodu (bądź innej sekwencji wykonującej). Zachowanie to można przyrównać do człowieka przeglądającego dowolne kompendium wiedzy i/lub instrukcję obsługi, gdzie w tekście otrzymuje plecenie sprawdzenie danego zagadnienia (opisanego na innej strony) bądź chcącego skorzystać z przypisu dolnego/indeksu.

Tego rodzaju wykonywanie jest dobre jeżeli aplikacja działa sama i wykonuje nieskomplikowane operacje, np. operacje arytmetyczne na podawanych przez użytkownika dwóch liczbach. Problem pojawi się, gdy użytkownik podczas liczenia będzie chciał posłuchać muzyki bądź prowadzić rozmowę z innym użytkownikiem poprzez sieć. Mikroprocesor jest jednozadaniowy – może wykonać jedynie 1 zadanie na takt (często nazywany tick). Szczęściem jest duża liczba taktów (1 Hz to jeden rozkaz na sekundę; przy obecnych procesorach 3,5-4 GHz otrzymujemy tychże taktów 3.500.000.000-4.000.000.000 możliwych instrukcji na jedną sekundę). Ponieważ jeden program nie wymaga tak dużej mocy obliczeniowej (może wymagać, jednak często nie wymaga) inżynierowe oprogramowania stworzyli rozwiązania podziału przydziału zasobów czasu procesora dla poszczególnych zadań.

Najprostszym jest podział czasu – podzielenie taktów procesora po równo na każde zadanie (mało praktyczny, acz obrazujący przykład – każde zadanie dostaje po 3 takty w każdym cyklu pracy mikroprocesora).

Pisząc programy komputerowe na jeden z systemów operacyjnych nie musimy się przejmować tego typu rozwiązaniami – to system operacyjny odpowiednio przydziela i zwalnia przydziały dla programów w nim uruchomionych (głównym programem komputera jest rdzeń systemu operacyjnego, podprogramami jego składniki, natomiast „programy” w rozumieniu standardowego użytkownika to podprogramy, powszechnie nazywane zadaniami). Wykonywanie wielu zadań w ramach jednego mikroprocesora (z jednym rdzeniem) zwykło nazywać się przetwarzaniem współbieżnym (współistniejącym).

Nowsze układy posiadają więcej niż jeden rdzeń wykonawczy (jeden rdzeń – jeden procesor logiczny). Ponadto istnieją sprzętowe rozwiązania współbieżności – potokowość, hiperpotokowość. Wszystko to sprawia, że czysto teoretycznie możemy w pełni niezależnie słuchać muzyki, puszczać film na projektorze w drugim pokoju i np. projektować grafikę 3D w odpowiednim oprogramowaniu. Praktycznie ograniczają nas potrzeby uruchamianej aplikacji (może wymagać więcej niż jednego procesora logicznego), przepustowość magistral, dostępu do pamięci, dostępu do peryferiów. Ponadto problemem może być sam system operacyjny (w obecnych czasach też

potrzebuje odpowiedniej rezerwacji zasobów na swoje tzw. rutynowe operacje). Wszystko to sprawia, że środowiska uruchomieniowe, chociaż posiadają niezależne obliczeniowe jednostki logiczne, nadal wykorzystują mechanizmy działania współbieżnego (opisane wcześniej).

b) przetwarzanie równoległe

W ramach jednego programu, np. do modelowania 3D, w chwili renderowania obrazu i/lub sceny wymagane są ogromne zasoby sprzętowe (przede wszystkim liczy się moc obliczeniowa mikroprocesora oraz pojemność pamięci operacyjnej; w dalszej kolejności ważna jest dobry układ graficzny). O ile domowy amator może sobie poczekać na wygenerowanie pożądanej treści (chociaż i on często cierpliwy nie jest), o tyle producenci filmowi każdą minutę przeliczają na tysiące dolarów. Trzeba pamiętać, że także szybkie przeprowadzanie symulacji ma kluczowe znaczenie w przypadku podjęcia akcji misjami kosmicznymi, kierowaniu akcjami ratunkowymi, wydobywania kopalin i innych (nie wspominając o płynności i szybkości działania elektronicznej rozrywki – gry i multimedia).

Do zadań profesjonalnych od dawna wykorzystuje się tzw. superkomputery, rozwiązujące problemy, których nie są w stanie rozwiązać ludzie i lub standardowe urządzenia. Tego typu urządzenia mają połączania fizyczne (większa liczba gniazd procesorowych na płycie głównej, większa ilość obsługiwanej pamięci RAM) lub sieciowe (przekazują dane pomiędzy sobą przy wykorzystaniu protokołów sieciowych). Dzięki odpowiednim bibliotekom i aplikacjom tego typu maszyny mogą pracować nad jednym zadaniem rozbijając je na części (np. obliczanie jednego bloku/fragmentu zdjęcia, gdzie wielkość bloku zależy od ilości możliwych do użycia rdzeni).

Tego typu działanie nazywa się równoległym. Pojedyncze zadanie uruchamia odpowiednią ilość wątków, które wykonują tradycyjne zadanie (wolno) rozkładając je na mniejsze fragmenty (szybsze obliczenia). Niekiedy zadania mogą być niezależne (asynchroniczne), innym razem wymagają spójności pomiędzy obrabianymi przez nie danymi (synchroniczne).

INFORMACJA: Zasadnicza różnica pomiędzy zadaniem a wątkiem jest następująca:
- każde zadanie ma przydzielane przez system swoje niezależne zasoby sprzętowe i podlega systemowi operacyjnemu/zadaniu wywołującemu (tzw. rodzic).
- wątki ściśle podlegają zadaniu, które je wywołuje. Wątki mogą korzystać tylko z tych zasobów, które zostały przydzielone zadaniu je wywołującym. Wątki kończą swoje działanie wraz z zadaniem.

Język Java nastawiony jest na aplikacje wielowątkowe. Posiada proste i wydajne mechanizmy do zrównoleglania zadań (w sensie wykonywania instrukcji – nie niezależnych aplikacji), zarządzania nimi oraz unicestwiania ich (w razie potrzeby).

Więcej informacji:
https://pl.wikipedia.org/wiki/Przetwarzanie_współbieżne
https://pl.wikipedia.org/wiki/Obliczenia_równoległe#Zrównoleglanie_na_poziomie_instrukcji

2. Zastosowanie wątków

Wątki przede wszystkim można wykorzystywać w różnego rodzaju obliczeniach. Szczególnie efektywne są w przypadku dużej ilości liczb, np. mnożenie macierzy 1000000x1000000, transponowanie macierzy, wynajdywaniu ścieżek, ożywiania sztucznej inteligencji (często w grach sztuczna inteligencja dostaje 1 lub dwa wątki celem poprawienia płynności gry). Oczywiście zrównoleglanie działania ma swoje ograniczenia.

Najważniejszym ograniczeniem (zasadą) działania wątków jest ich tzw. bezpieczeństwo. Wątki nie powinny same z siebie modyfikować zmiennych z wątku głównego (programu), a jedynie przez odpowiednie metody, które odpowiednio zadbają o chronologiczny zapis danych. Jeżeli wątki mogłyby np. modyfikować zmienną a w dowolnej chwili, to istnieje wysokie prawdopodobieństwo, że wartość zmiennej a będzie niepoprawna (szczególnie w przypadku, gdy jeden wątek ma do niej dodawać wyliczoną wartość, a drugi ją przemnażać lub dzielić).

Drugim ograniczeniem jest wspomniana chronologiczność (sekwencyjność) działań. Jeżeli wątki działają na swoich zestawach zmiennych to wspomniany problem nie występuje. Jeżeli jednak ich praca musi być co jakiś czas synchronizowana to ich efektywność będzie zależeć w głównej mierze właśnie od częstotliwości dokonywania synchronizacji wyników. Należy tutaj wziąć pod uwagę czas tworzenia i kopiowania danych do wątku, jego wykonanie, czas oczekiwania na pozostałe wątki, uzupełnianie danych wspólnych, ponownie wybudzanie.

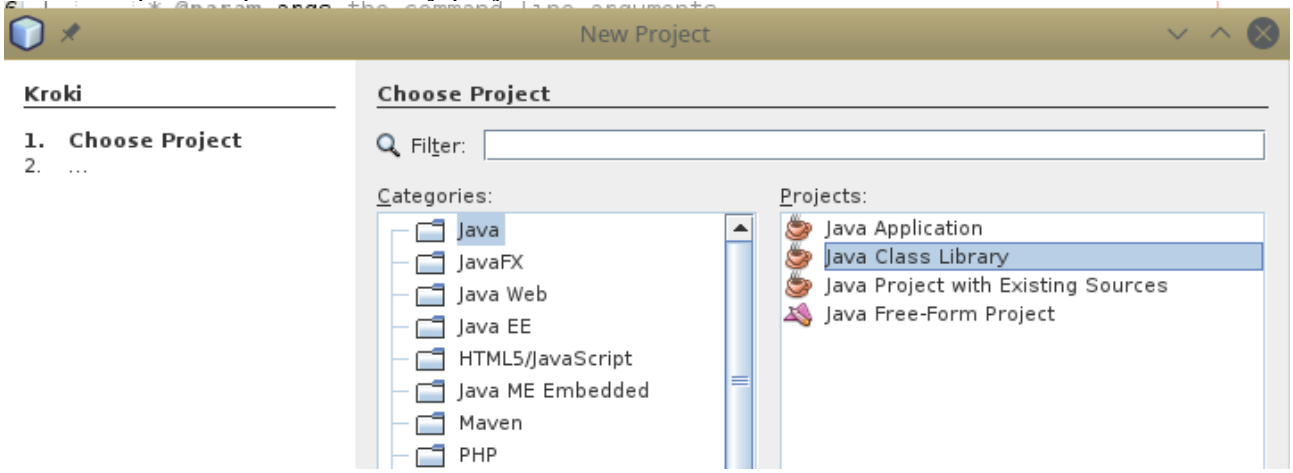
Ostatnim ograniczeniem może być sprzęt jednordzeniowy (obecnie rzadkość). Najlepszym wariantem (optymalnym) jest działanie tylu wątków, ile mamy do dyspozycji rdzeni w mikroprocesorze (jeden wątek na rdzeń) bądź potoków (np. dwu wątkowy rdzeń – specjalność firmy Intel oraz AMD od architektury Ryzen). Jeżeli utworzymy więcej wątków to będą one musiały ze sobą walczyć o czas procesora,a co za tym idzie znacznie je to spowolni (tym samym spadnie efektywność kodu wielowątkowego). Wyjątkiem mogą być sytuacje, gdy wątki mają do spełnienia określone role, większość czasu pozostają w uśpieniu.

3. Przykład prostego programu wykorzystującego wątki.

Program ma kilka zadań:
- pokazać działanie wątków
- pokazać podłączanie bibliotek do swojego programu
- pomiar czasu działania aplikacji
- różnice w wykonywaniu tych samych działań w przypadku typów prostych i obiektowych

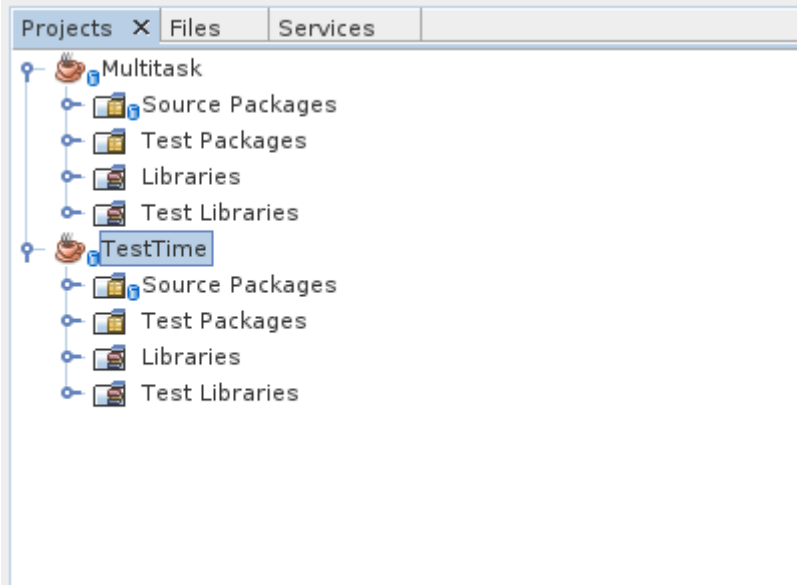
a) Przede wszystkim zacznijmy od tego czym jest biblioteka w Java. Biblioteką jest każdy projekt, który nie posiada klasy wykonawczej (klasy, w której znajduje się metoda main). Biblioteki mają to do siebie, że można je bez problemu łączyć do wielu różnych projektów, nad którymi pracujemy. Jeżeli cokolwiek zmienimy w kodzie biblioteki, program docelowy z niej korzystający automatycznie wykona zmodyfikowany kod (bez potrzeby przekompilowania go).

NetBeans posiada predefiniowany projekt biblioteki:



aczkolwiek nic nie stoi na przeszkodzi by z dowolnego projektu utworzyć bibliotekę (można też wybrać opcję Java Free-Form Project).

W środowiskach programistycznych możemy pracować jednocześnie nad kilkoma projektami, tak więc równolegle możemy mieć otwarty projekt(y) aplikacji oraz bibliotek(i):



W powyższym przypadku projekt TestTime jest biblioteką.

Biblioteka zawiera jedną klasę, która będzie odpowiedzialna za pomiary czasowe. Czas liczony w nanosekundach, odpowiednio konwertowany na milisekundy bądź sekundy.

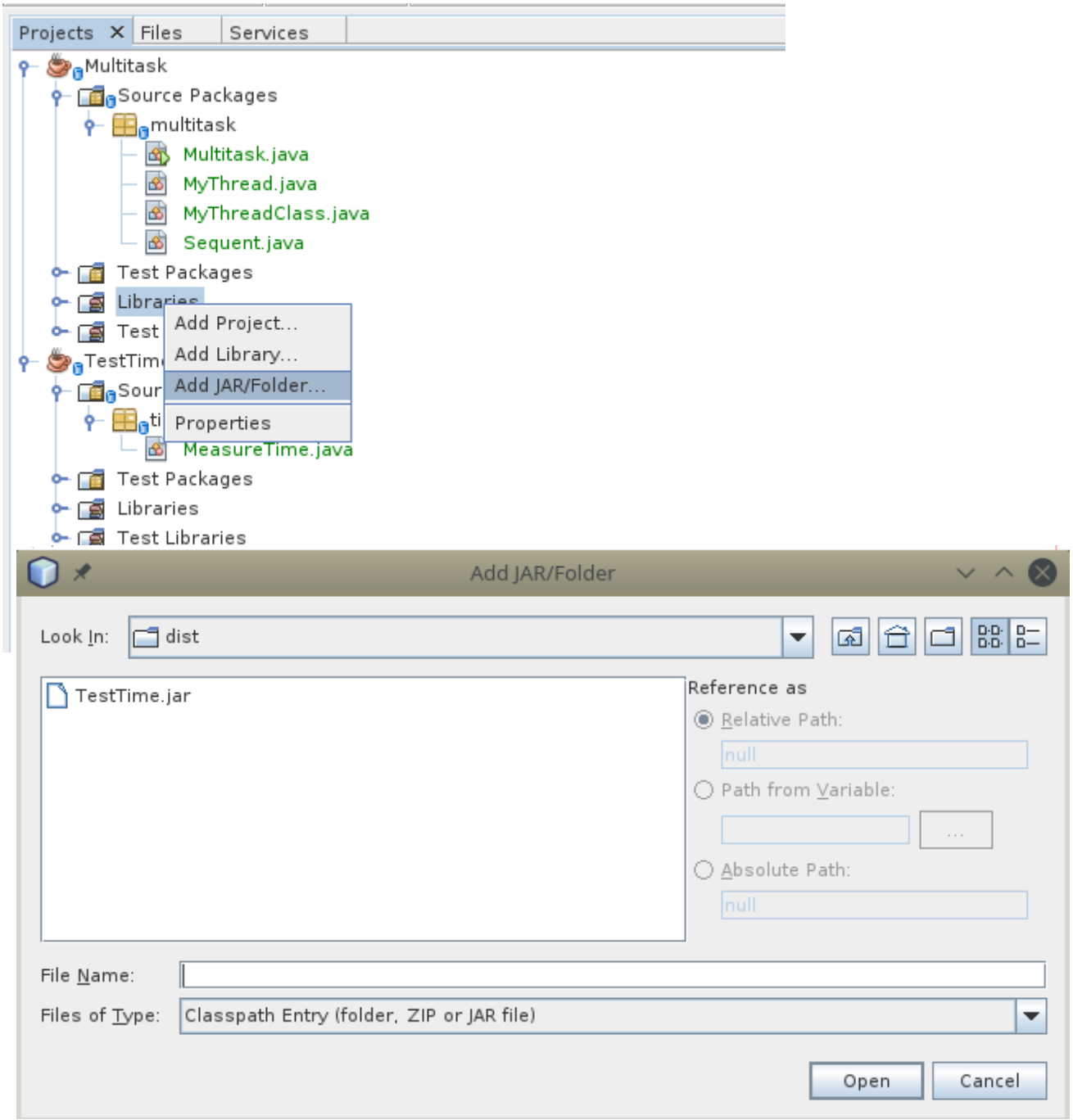
b) Projekt właściwy zawiera cztery klasy:
- MultiTask – będąca główną klasą; jej zadaniem jest jedynie tworzyć obiekty z innych klas i wykonywać ich kod
- MyThread – klasa operująca na wątkach będąca implementacją interfejsu Runnable. Posiada w sobie obiekt typu Thread (klasa wbudowana), który jest właściwym wątkiem.
- MyThreadClass – klasa operująca na wątkach rozszerzona (będąca rozszerzeniem) klasy Threads. Podobnie jak w przypadku interfejsu możemy nadpisywać jej metody.

INFORMACJA: Pomijając mało istotną sprawę ograniczonego dziedziczenia klas po innych klasach (można je pozostawić na inną klasę, bardziej przydatną), interfejs jest wskazywany jako bardziej efektywne rozwiązanie tworzenia wątków. Przykładowo wywołanie jego działania poprzez metodę run() nie spowoduje wycieków pamięci (w klasie dziedziczonej po Thread może do niego dojść). Ponadto konwencja w Java mówi – jeżeli nie chcesz zmieniać zachowania klasy, a jedynie je posiąść lepiej zrobić to poprzez interfejs.

- Sequent – klasa wykonująca to samo zadanie co klasy wątkowe, jednak w postaci jednego wątku, wykonywanego w głównym wątku programu.

Sam program nie robi nic innego jak dodaje liczby typu long z odpowiedniego przedziału. W programie dostępne są 3 pętle – pierwsza operująca na typie long, druga operująca w pełni na typie Long (także w parametrach for) oraz trzecia operująca na typie Long jednak nie w parametrach pętli (tutaj występują standardowe typy podstawowe)

INFORMACJA: Dodanie nowej biblioteki, niezbędnej do tego zadania, jest bardzo proste. Wystarczy wybrać prawym przyciskiem myszy katalog Library i dołączyć plik jar z dysku/innego projektu.



Po tej operacji mamy dostęp do wszystkich klas oraz wszystkich paczek dostępnych w ramach biblioteki (możemy np. używać interfejsów, klas, klas abstrakcyjnych definiowanych w takiej klasie).

3. Wywoływanie wątków w ramach interfejsu Concurrent Future.

Od siódmej wersji Java dysponuje kompleksowym rozwiązaniem obsługi przetwarzania wielowątkowego. Jest to interfejs Future, którego nazwa pochodzi od otrzymywania wyników przyszłości. Oznacza to tyle, że zadania wykonują się asynchronicznie – po rozpoczęciu kodu obsługę nad zadaniem przejmuje tzw. ExecutorService, a nasz kod może dalej wykonywać się sekwencyjnie (bądź równolegle na innych szczeblu). W chwili gdy chcemy otrzymać wyniki możemy użyć specjalnie przygotowanych metod tzw. blokujących (ich blokowanie polega na oczekiwaniu wyniku z wątków i do czasu jego braku zatrzymania wykonywania całego programu).

Chociaż działanie mechanizmu niczym nie różni się od działania własnoręcznie tworzonych wątków, jest ono znacznie wygodniejsze w użyciu:
- nie musimy dbać o obsługę błędów wątków
- nie musimy przejmować się blokowaniem zasobów dla wątków
- nie musimy martwić się posprzątaniem wątków przed zakończeniem programów (gotowe metody shutdown() oraz shutdownNow())

Ponieważ w sieci krąży sporo materiałów na temat tego rozwiązania, jednak wiele z nich mało tłumaczy działanie (często przykłady są dosyć zawile napisane, często w postaci metod anonimowych, których konstrukcja znacznie zaciemnia tłumaczenie) w niniejszym materiale zostanie wytłumaczone działanie mechanizmu w oparciu o dotychczasowy przykład kodu (dostępny pod adresem repozytorium https://gitlab.com/examples4java/kurs2). Będzie zbudowany w oparciu o dodatkową klasę – FutureExample.

Klasa sama w sobie jest standardowa. Posiada prywatne zmienne oraz konstruktor. Generalnie może zawierać dowolne metody, algorytmy i sposoby przekazywania danych. Może nawet wyświetlać komunikaty. Jedynym niepolecanym elementem w tego typu klasie (dotyczy wszystkich klas wątkowych) jest wykorzystywanie wejścia (np. oczekiwanie na parametry od użytkownika). Tego typu operacje powinny (muszą) znajdować się w głównym wątku programu (tak samo zresztą jak elementy operowania na obrazie, klawiaturze czy innych urządzeniach wejścia/wyjścia).

Jedyną różnicą jest użycie interfejsu Callable. Pozwala on na wywołanie naszej klasy poprzez obiekt ExecutorService. Musimy nadpisać metodę call(), której ciało może być identyczne do zawartości metody run() w poprzednich wątkach.

```
public class FutureExample implements Callable<Object>{
   private long _start, _stop;
   private String name;
   public FutureExample() {
      this(0,1000000000,String.valueOf(MeasureTime.getTimeStamp()));
   }
   public FutureExample(long _start, long _stop) {
      this(_start,_stop, String.valueOf(MeasureTime.getTimeStamp()));
   }
   public FutureExample(long _start, long _stop, String name) {
      this._start = _start;
      this._stop = _stop;
      this.name = name;
   }
   @Override
   public Object call() throws Exception {
      MeasureTime t = new MeasureTime();
      MeasureTime t1 = new MeasureTime(true);
      long l = 0;
      Long l1 = l;
      t.startMeasure();
      for (long i = _start; i < _stop; i++)
         l+=i;
      System.out.println("Wątek " + name + ": " +t.getMeasure()+", wynik: " + l);
      t.startMeasure();
      for (Long i = _start; i < _stop; i++)
         l1+=i;
```

```
        System.out.println("Wątek " + name + ": " +t.getMeasure()+", wynik: " + l1);
        l1 = new Long(0);
        t.startMeasure();
        for (long i = _start; i < _stop; i++)
            l1+=i;
        System.out.println("Wątek " + name + ": " +t.getMeasure()+", wynik: " + l1);
        System.out.println("Wątek " + name + " kończy pracę. Całkowity czas pracy: "
+t1.getMeasure());
        return l1;
    }
}
```

Jak można zauważyć, w powyższym kodzie interfejs Callable posiada szablon przyjmowanych danych (w tym wypadku Object). Ponadto metoda call zwraca zmienną typu podanego w diamencie. Chociaż w przykładzie podany został Object (klasa ogólna – podbudowa wszystkich klas) nic nie stoi na przeszkodzie by zwracany był inny typ (zarówno obiektowy typów podstawowych jak i nasz własny obiekt – inna klasa bądź interfejs).

Teraz klasę należy odpowiednio utworzyć i uruchomić. Uruchomienie jej zapewnia wspominany kilkukrotnie ExecutorService. Jest to klasa obiektu, który ma za zadanie uruchamiać, zarządzać oraz zakańczać wywoływane wątki interfejsu Future. Posiada kilka przydatnych metod, z czego najważniejsze to:
- newSingleThreadExecutor() – uruchamia wszystkie podane przez nas wątki SEKWENCYJNIE. Oznacza to, że kolejny wątek uruchamia się po zakończeniu poprzedniego. Zyskiem jest wykonywanie tegoż wątku np. w czasie, gdy użytkownik dokonuje rejestracji w programie (a wynik z takiego wątku otrzyma po wykonaniu zadania)
- newFixedThreadPool(int i) - użyte do przykładu; ustala na sztywno liczbę wątków (wartość i)
- newCachedThreadPool() - podobne do poprzedniego jednak w tym wypadku wykonawca sam decyduje o tworzeniu i/lub usuwaniu wątków (ich liczba jest dynamiczna – poprzednio statyczna)
- newSingleThreadScheduledExecutor() - zwraca nam wykonawcę (executor), który może uruchomić wątek z zadanym opóźnieniem (np. o 10 sekund)
- newScheduledThreadPool(int i) – analogicznie do poprzedniego, jednak ręcznie ustalamy ilość wątków do późniejszego wykonania

Po uruchomieniu utworzone wątki działają „wiecznie”; gdy kończą swoje zadania są niejako „usypiane” na wypadek, gdyby ponownie były potrzebne. To powoduje, że napisany program nie wyłącza się. Aby temu zapobiec trzeba dodatkowo wykonać jedną z dwóch poniższych metod:
- shutdown() - wydaje polecenie zamknięcia/wyłączenia działających wątków. Nie zakończą się one natychmiast lecz dopiero w chwili zakończenie instrukcji w metodzie call (bądź błędu przez nią wywołanego)
- shutdownNow() - kończy bezwzględnie wszystkie rozpoczęte wątki.

Całość kodu wywołującego obiekty na podstawie poprzedniej klasy:

```
        private static final int THREADS_COUNT = 2;
        ExecutorService es = Executors.newFixedThreadPool(THREADS_COUNT);
        long amount = 1000000000;
        long chunk = amount/THREADS_COUNT;
        for(int i=0;i<THREADS_COUNT;i++) {
            FutureExample fe = new FutureExample(chunk*i, (i==THREADS_COUNT-1) ? amount :
chunk*i+chunk,"Praca"+(i+1));
            Future<Object> f = es.submit(fe);
        }
```

```
es.shutdown();
if (es.isTerminated())
   System.out.println("Koniec pracy egzekutora! ");
```

W pierwszej linijce tworzymy nowego wykonawcę kodu z określoną wielkością.

Następnie tworzymy dwie zmienne – całkowitą ilość liczb do przeliczenia oraz pojedynczego kawałka z tych liczb, który będzie liczony na pojedynczym wątku.

W pętli for uruchamiamy odpowiednią ilość wątków. Tworzymy obiekt z napisanej wcześniej klasy i tworzymy nowy obiekt Future, do którego zwracamy nasz przyszły wynik z działania wątku.

UWAGA! Wynik z wątku można pobrać na dwa sposoby:
- blokującą metodą get(). Po zakończeniu wątku zwróci nam ona wynik do odpowiedniej zmiennej. Wymusi ona jednak zatrzymanie się programu na czas wykonywania wątku (co, patrząc na asynchroniczność Fuiture nie jest najlepszym pomysłem)
- można odpytywać wątek czy już zakończył działanie (metoda isDone()). Jeżeli zwrócona przez nią wartość będzie typu true to znaczy, że możemy już pobrać wartość naszego wyniku. W przeciwnym razie możemy wykonywać inny kod aplikacji (np. innego wątku bądź odpytywać naszego użytkownika o różnego rodzaju informacje). W tym wypadku zachowamy asynchroniczne działanie aplikacji.

JEDNAK BY MÓC ODWOŁAĆ SIĘ DO TYCH METO MUSIELIBYŚMY TWORZYĆ ODPOWIEDNIĄ ZMIENNĄ NIELOKALNĄ! (np. tablicę elementów Future)

Na koniec, po pętli wywołujemy metodę shutdown(). Dzięki niej, gdy wątki zakończą działalność, zakończy się również nasz program. Bez tego wywołania program działałby w nieskończoność.

Ostatni if sprawdza, czy wątki zostały zakończone. Jeżeli tak – wyświetla stosowny komunikat. Należy zauważyć, że instrukcja nie jest pętlą jednak wykona się dopiero po jakimś czasie (gdy wątki zakończą działanie!)

Więcej o wątkach:
https://dzone.com/articles/basics-of-using-java-future-and-executor-service
http://czub.info/2017/java-8-wielowatkowosc-cz3-egzekutory-pule-watkow-future-i-callable/
http://www.baeldung.com/java-future
http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/
https://stackoverflow.com/questions/541487/implements-runnable-vs-extends-thread
https://dzone.com/articles/java-callable-future-understanding
https://www.tutorialspoint.com/java/java_multithreading.htm
Pomiar czasu wykonywania programu:
https://stackoverflow.com/questions/1770010/how-do-i-measure-time-elapsed-in-java
https://stackoverflow.com/questions/20689055/java-8-instant-now-with-nanosecond-resolution

Zadania do samodzielnego opracowania:
- w jaki sposób można zarządzać tradycyjnymi wątkami w Java (np. ThreadPool)?
- w jaki sposób można współdzielić wspólne dane pomiędzy wątkami?

Zadanie do napisania:
1. Stworzyć tekstowy symulator sklepu. Należy rozważyć następujące klasy (to jedynie przykład – nie pełne rozwiązanie):
- sprzedawca – pracownik sklepu mogący służyć nam pomocą
- kasjer – sprawdza co mamy w koszyku i nalicza nam należość
- koszyk – zapełniany odpowiednimi artykułami
- gracz – ogólna klasa opisująca gracza, jego stan majątkowy, jego szybkość oraz największe potrzeby
- półka – element, który pozwala nam zdobyć upragniony towar
- towar – ogólna klasa opisująca nazwę produktu, jego cenę oraz kategorię
- klienci – klasa podobna do gracza jednak są nimi osoby sterowane przez komputer

Naszym celem jest zdobycie potrzebnych nam towarów w jak najkrótszym czasie. Na początku programu powinny być rozlosowane ilości półek, towarów na nich, ilość kupujących, ich statystyki oraz statystyki gracza. Następnie ilość półek powinna być tożsama ze sprzedawcami.

Gracz, mając do dyspozycji swoją listę zakupów (preferencje zakupowe) może wybrać półkę, od której chce zacząć kupować. Dostęp do półki zajmuje określony czas (odległość do przebycia). Po przybyciu sprawdzane jest ile osób czeka do sprzedawcy (każdy klient w kolejce + czas wykonywania akcji).

Po zakupach trafiamy do kasy, gdzie kasjer ściąga należność. Najszybszy wygrywa.