Wielowątkowość

dzień 2

v3.0

Plan

1. BlockingQueue
2. ExecutorService
3. ThreadPoolExecutor
4. ScheduledExecutorService
5. Java Collections

BlockingQueue

BlockingQueue

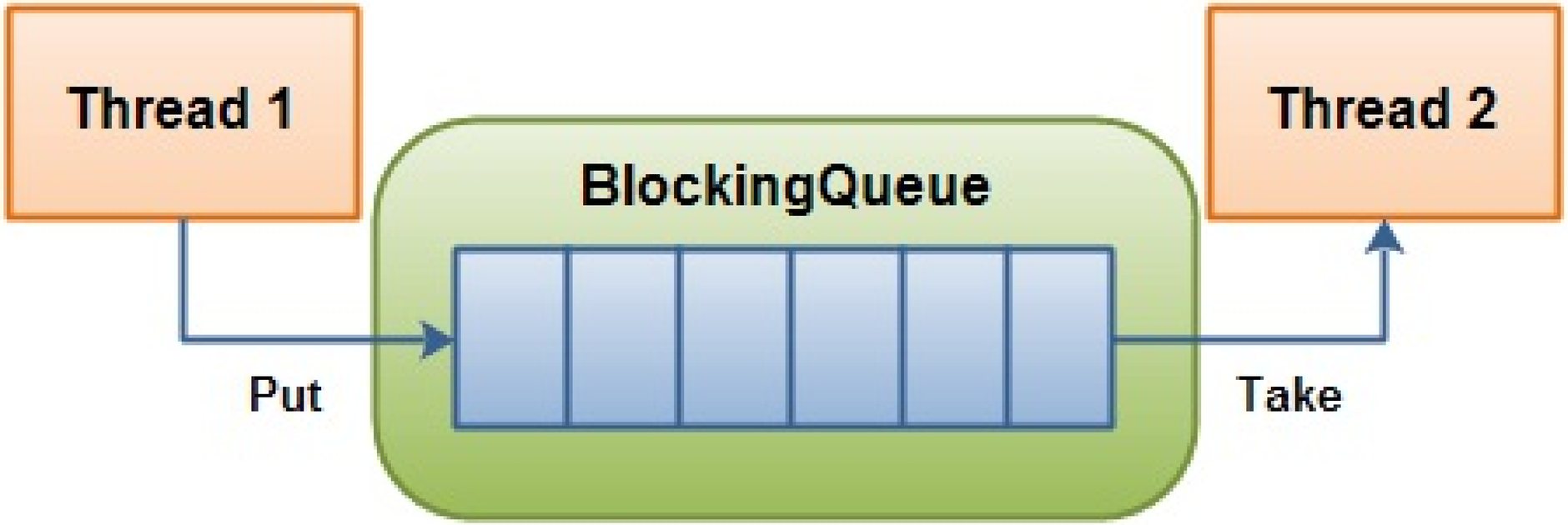
BlockingQueue jest interfejsem z pakietu java.util.concurrent i reprezentuje kolejkę (ang. queue) która jest bezpieczna do użycia podczas pracy z wątkami

BlockingQueue - jest idealnym rozwiązaniem, jeżeli chcemy ominąć złożoność związaną z poznanymi wcześniej metodami wait() oraz notify(), a także idealnym rozwiązaniem przy problemie producent/konsument, który poznamy przy omawianiu wzorców projektowych.

BlockingQueue bez problemu obsługuje dodawanie elementów do kolejki oraz w prosty sposób umożliwia korzystanie z zasobów kolejki.

Java sama w sobie udostępnia metody do kontroli takiego zachowania gdzie jeden wątek tworzy elementy kolejki a drugi z nich korzysta właśnie poprzez interfejs BlockingQueue

BlockingQueue



BlockingQueue

Najpopularniejsze implementacje interfejsu BlockingQueue to:

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

Główne metody interfejsu BlockingQueue to:

- **`put()`** - służy do wprowadzania danych do kolejki
- **`take()`** - służy do wyciągania danych z kolejki

BlockingQueue przykład

Klasa dodająca elementy do kolejki:

```
public class Producer implements Runnable{
  protected BlockingQueue queue = null;
  public Producer(BlockingQueue queue) {
    this.queue = queue;
  }
}
```

BlockingQueue przykład

Klasa dodająca elementy do kolejki:

```
public class Producer implements Runnable{
  protected BlockingQueue queue = null;
  public Producer(BlockingQueue queue) {
    this.queue = queue;
  }
}
```

utworzenie nowej klasy implementującej interfejs Runnable

BlockingQueue przykład

Klasa dodająca elementy do kolejki:

```
public class Producer implements Runnable{
  protected BlockingQueue queue = null;
  public Producer(BlockingQueue queue) {
    this.queue = queue;
  }
}
```

utworzenie pola klasy

BlockingQueue przykład

Klasa dodająca elementy do kolejki:

```
public class Producer implements Runnable{
  protected BlockingQueue queue = null;
  public Producer(BlockingQueue queue) {
    this.queue = queue;
  }
}
```

utworzenie konstruktora

BlockingQueue przykład

Klasa dodająca elementy do kolejki:

```
public class Producer implements Runnable{
  protected BlockingQueue queue = null;
  public Producer(BlockingQueue queue) {
    this.queue = queue;
  }
}
```

konstruktor przyjmuje obiekt typu BlockingQueue

BlockingQueue przykład

Metoda run() klasy dodającej elementy do kolejki:

```
public void run() {
    try {
      queue.put("1");
      Thread.sleep(1000);
      queue.put("2");
      Thread.sleep(2000);
      queue.put("3");
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

BlockingQueue przykład

Metoda run() klasy dodającej elementy do kolejki:

```
public void run() {
    try {
      queue.put("1");
      Thread.sleep(1000);
      queue.put("2");
      Thread.sleep(2000);
      queue.put("3");
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

dodanie elementów do kolejki

BlockingQueue przykład

Metoda run() klasy dodającej elementy do kolejki:

```
public void run() {
    try {
      queue.put("1");
      Thread.sleep(1000);
      queue.put("2");
      Thread.sleep(2000);
      queue.put("3");
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

uśpienie wątku na 1 a potem 2 sekundy

BlockingQueue przykład

Klasa pobierająca elementy z kolejki:

```
public class Consumer implements Runnable{
  protected BlockingQueue queue = null;
  public Consumer(BlockingQueue queue) {
        this.queue = queue;
  }
}
```

BlockingQueue przykład

Metoda run() klasy pobierającej elementy z kolejki:

```
public void run() {
    try {
      System.out.println(queue.take());
      System.out.println(queue.take());
      System.out.println(queue.take());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

BlockingQueue przykład

Metoda run() klasy pobierającej elementy z kolejki:

```
public void run() {
    try {
      System.out.println(queue.take());
      System.out.println(queue.take());
      System.out.println(queue.take());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

pobieranie elementów z kolejki

BlockingQueue przykład

Uruchomienie kodu:

```
public static void main(String[] args) throws Exception {
  BlockingQueue queue = new ArrayBlockingQueue(1024);
  Producer producer = new Producer(queue);
  Consumer consumer = new Consumer(queue);
  new Thread(producer).start();
  new Thread(consumer).start();
}
```

BlockingQueue przykład

Uruchomienie kodu:

```
public static void main(String[] args) throws Exception {
  BlockingQueue queue = new ArrayBlockingQueue(1024);
  Producer producer = new Producer(queue);
  Consumer consumer = new Consumer(queue);
  new Thread(producer).start();
  new Thread(consumer).start();
}
```

utworzenie obiektu BlockingQueue typu ArrayBlockingQueue

BlockingQueue przykład

Uruchomienie kodu:

```
public static void main(String[] args) throws Exception {
  BlockingQueue queue = new ArrayBlockingQueue(1024);
  Producer producer = new Producer(queue);
  Consumer consumer = new Consumer(queue);
  new Thread(producer).start();
  new Thread(consumer).start();
}
```

utworzenie obiektów wcześniej stworzonych klas

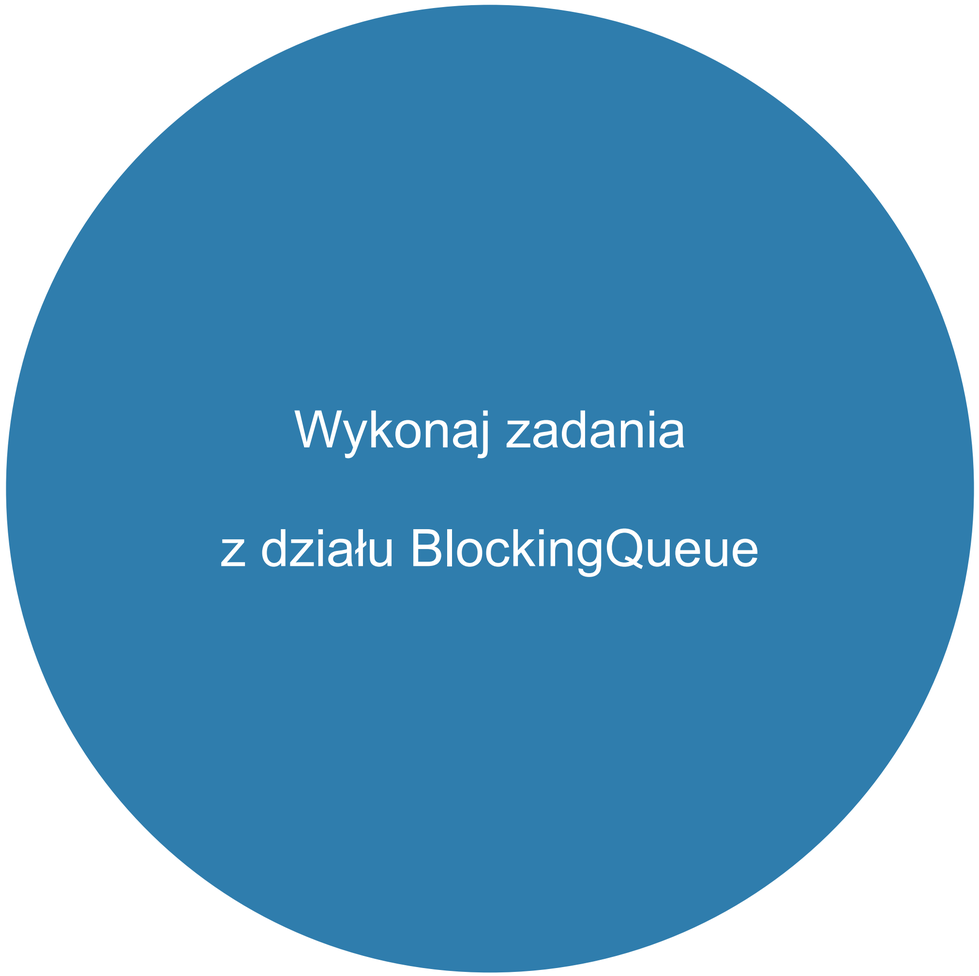
BlockingQueue przykład

Uruchomienie kodu:

```
public static void main(String[] args) throws Exception {
  BlockingQueue queue = new ArrayBlockingQueue(1024);
  Producer producer = new Producer(queue);
  Consumer consumer = new Consumer(queue);
  new Thread(producer).start();
  new Thread(consumer).start();
}
```

uruchomienie wątków

Zadania



ExecutorService

ExecutorService

ExecutorService - jest to interfejs z pakietu java.util.concurrent.ExecutorService który umożliwia uruchamianie zadań w tle.

Konstrukcja ExecutorService sprawia, że kod zostanie wykonany przez jeden z wątków zawartych w obiekcie ExecutorService. Interfejs ExecutorService ma następujące implementacje w pakiecie java.util.concurrent które omówimy później:

- **`ThreadPoolExecutor`**
- **`ScheduledThreadPoolExecutor`**

ExecutorService

Tworzenie obiektu ExecutorService zależy od implementacji, ale można użyć fabryki Executors by utworzyć jej instancję, np.:

- `ExecutorService executorService1 = Executors.newSingleThreadExecutor();`
- `ExecutorService executorService2 = Executors.newFixedThreadPool(10);`
- `ExecutorService executorService3 = Executors.newScheduledThreadPool(10);`

Warto również zapewnić poprawne zamknięcie ExecutorService. Aby nastąpiło to po wykonaniu zadań we wszystkich pracujących wątkach wewnątrz ExecutorService używamy w tym celu metody **`shutdown()`** na obiekcie ExecutorService.

Użycie ExecutorService

Istnieje kilka metod aby delegować zadania do ExecutorService, poniżej kilka metod które opiszemy:

- execute(Runnable)
- submit(Runnable)
- submit(Callable)
- invokeAny()
- invokeAll()

execute(Runnable)

Argumentem metody **`execute(Runnable)`** jest obiekt java.lang.Runnable który wykonywany jest w danej implementacji ExecutorService.

W przypadku użycia tej metody nie ma możliwości uzyskania wyniku działania przekazanego obiektu Runnable.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.execute(new Runnable() {
  public void run() {
    System.out.println("Running task");
  }
});
executorService.shutdown();
```

submit(Runnable)

Argumentem metody **submit(Runnable)** jest również obiekt Runnable, ale zwraca obiekt Future na podstawie którego można określić zakończenie wykonania zadania.

Metoda **`future.get()`** zwraca null jeżeli zadanie w danym wątku zakończyło się poprawnie, w przeciwnym wypadku zostanie rzucony wyjątek **`ExecutionException`**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future future = executorService.submit(new Runnable() {
  public void run() {
    System.out.println("Running task");
  }
});
future.get();
```

submit(Callable)

Metoda **submit(Callable)** jest podobna do metody **`submit(Runnable)`** z wyjątkiem przyjmowanego argumentu w tym przypadku instancji obiektu Callable która za pomocą metody **`call()`** potrafi zwrócić wynik

Wynik działania obiektu Callable może być w tym przypadku uzyskany poprzez metodę **`get()`** obiektu Future

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future future = executorService.submit(new Callable(){
  public Object call() throws Exception {
    System.out.println("Running Callable");
    return "Callable return value";
  }
});
System.out.println("Returned value " + future.get());
```

invokeAny()

Metoda **invokeAny()** pobiera kolekcję obiektów Callable. Wykonanie tej metody nie zwraca obiektu Future, ale zwraca wynik jednego z obiektów Callable z kolekcji - tego który wykona się pierwszy.

W przypadku zakończenia działania przez jeden z obiektów Callable, wykonanie reszty obiektów Callable z kolekcji zostaje anulowana

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 1";
  }
});
```

invokeAny()

Metoda **invokeAny()** pobiera kolekcję obiektów Callable. Wykonanie tej metody nie zwraca obiektu Future, ale zwraca wynik jednego z obiektów Callable z kolekcji - tego który wykona się pierwszy.

W przypadku zakończenia działania przez jeden z obiektów Callable, wykonanie reszty obiektów Callable z kolekcji zostaje anulowana

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 1";
  }
});
```

utworzenie obiektu HashSet typu Callable String

invokeAny()

Metoda **invokeAny()** pobiera kolekcję obiektów Callable. Wykonanie tej metody nie zwraca obiektu Future, ale zwraca wynik jednego z obiektów Callable z kolekcji - tego który wykona się pierwszy.

W przypadku zakończenia działania przez jeden z obiektów Callable, wykonanie reszty obiektów Callable z kolekcji zostaje anulowana

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 1";
  }
});
```

dodanie nowych elementów Callable do kolekcji HashSet

invokeAny()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
String result = executorService.invokeAny(callables);
System.out.println("returned value: " + result);
executorService.shutdown();
```

invokeAny()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
String result = executorService.invokeAny(callables);
System.out.println("returned value: " + result);
executorService.shutdown();
```

wywołanie metody invokeAny() i pobranie wyniku

invokeAny()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
String result = executorService.invokeAny(callables);
System.out.println("returned value: " + result);
executorService.shutdown();
```

wypisanie wyniku działania jednego z obiektów Callable

invokeAll()

Metoda **invokeAll()** wywołuje wszystkie obiekty Callable przekazane jako kolekcja w parametrze. Metoda zwraca listę obiektów Future dzięki którym można poznać wynik działania każdego z przekazanych obiektów Callable(które zwracają taki sam typ danych)

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 1";
  }
});
```

invokeAll()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
  System.out.println("returned value: " + future.get());
}
executorService.shutdown();
```

invokeAll()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
  System.out.println("returned value: " + future.get());
}
executorService.shutdown();
```

wykonanie zadań za pomocą metody invokeAll() i pobranie wyników

invokeAll()

Dalsza część kodu:

```
callables.add(new Callable<String>() {
  public String call() throws Exception {
    return "Task 2";
  }
});
List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
  System.out.println("returned value: " + future.get());
}
executorService.shutdown();
```

wypisanie wyników wszystkich wykonanych zadań

Zadania

ThreadPool Executor

ThreadPoolExecutor

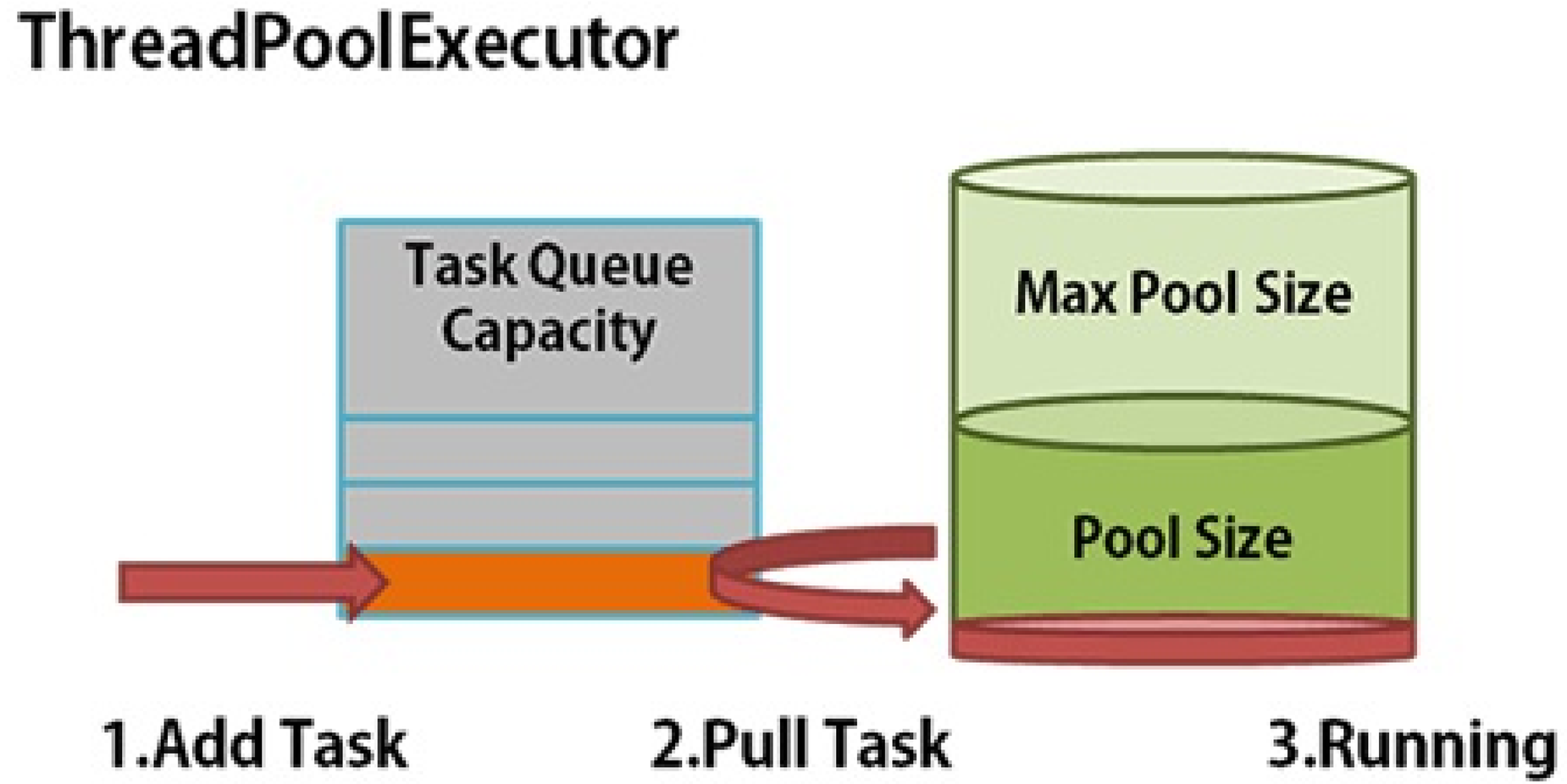
ThreadPoolExecutor z pakietu java.util.concurrent.ThreadPoolExecutor jest implementacją interfejsu ExecutorService który wykonuje dane zadanie(typu Callable lub Runnable) używając jednego ze swoich wewnętrznych wątków

Liczba wątków zawartych w ThreadPoolExecutor jest określona za pomocą parametrów **`corePoolSize`** oraz **`maximumPoolSize`**

Jeżeli podczas przypisywania zadania do wątków ze zbioru wątków w ThreadPoolExecutor, liczba wątków wewnątrz zbioru jest mniejsza od corePoolSize wtedy tworzone są nowe wątki

Jeżeli wewnętrzna kolejka zadań jest pełna i liczba działających wątków jest równa lub większa corePoolSize lecz mniejsza od maximumPoolSize wtedy tworzony jest nowy wątek do wykonania danego zadania

ThreadPoolExecutor



ThreadPoolExecutor

Jeżeli stosowana implementacja nie wymaga specyficznych wartości wątków podawanych w konstruktorze ThreadPoolExecutor wtedy łatwiej jest użyć wcześniej poznanych metod klasy java.util.concurrent.Executors, poniżej przykład konstruktora ze wszystkimi parametrami:

```
ExecutorService threadPoolExecutor =
new ThreadPoolExecutor(
  corePoolSize,
  maxPoolSize,
  keepAliveTime,
  TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<Runnable>()
);
```

Praktyczny przykład użycia ThreadPoolExecutor

Klasa zawierająca zadanie do wykonania:

```
class Task implements Runnable {
  private String name;
  public Task(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
```

Praktyczny przykład użycia ThreadPoolExecutor

Nadpisanie metody run():

```
  @Override
  public void run() {
    try {
      Long duration = (long) (Math.random() * 10);
      TimeUnit.SECONDS.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Praktyczny przykład użycia ThreadPoolExecutor

Wykonanie kodu:

```
public static void main(String[] args) {
  ThreadPoolExecutor ex = (ThreadPoolExecutor) Executors.newCachedThreadPool();
  for(int i = 0; i <= 5; i++) {
    Task task = new Task("Task " + i);
    ex.execute(task);
  }
  ex.shutdown();
}
```

Praktyczny przykład użycia ThreadPoolExecutor

Wykonanie kodu:

```
public static void main(String[] args) {
  ThreadPoolExecutor ex = (ThreadPoolExecutor) Executors.newCachedThreadPool();
  for(int i = 0; i <= 5; i++) {
    Task task = new Task("Task " + i);
    ex.execute(task);
  }
  ex.shutdown();
}
```

uruchomienie w pętli 5 wątków metodą execute()

ThreadPoolExecutor

ThreadPoolExecutor posiada różne typy konstruktorów, ale ze względu na złożoność tego obiektu dostarcza on klasę fabrykę **`Executors`** która ułatwia tworzenie instancji. Wykorzystane w przykładzie **`Executors.newCachedThreadPool()`** tworzy nowe wątki potrzebne do wykonania kolejnych zadań a następnie wykorzystuje istniejące wątki jeżeli zakończyły one już swoje działanie

Minusem tego rozwiązania jest to, że jeżeli przekażemy zbyt wiele zadań do wykonania możemy przeciążyć system, można temu zaradzić używając fixed size thread pool executor

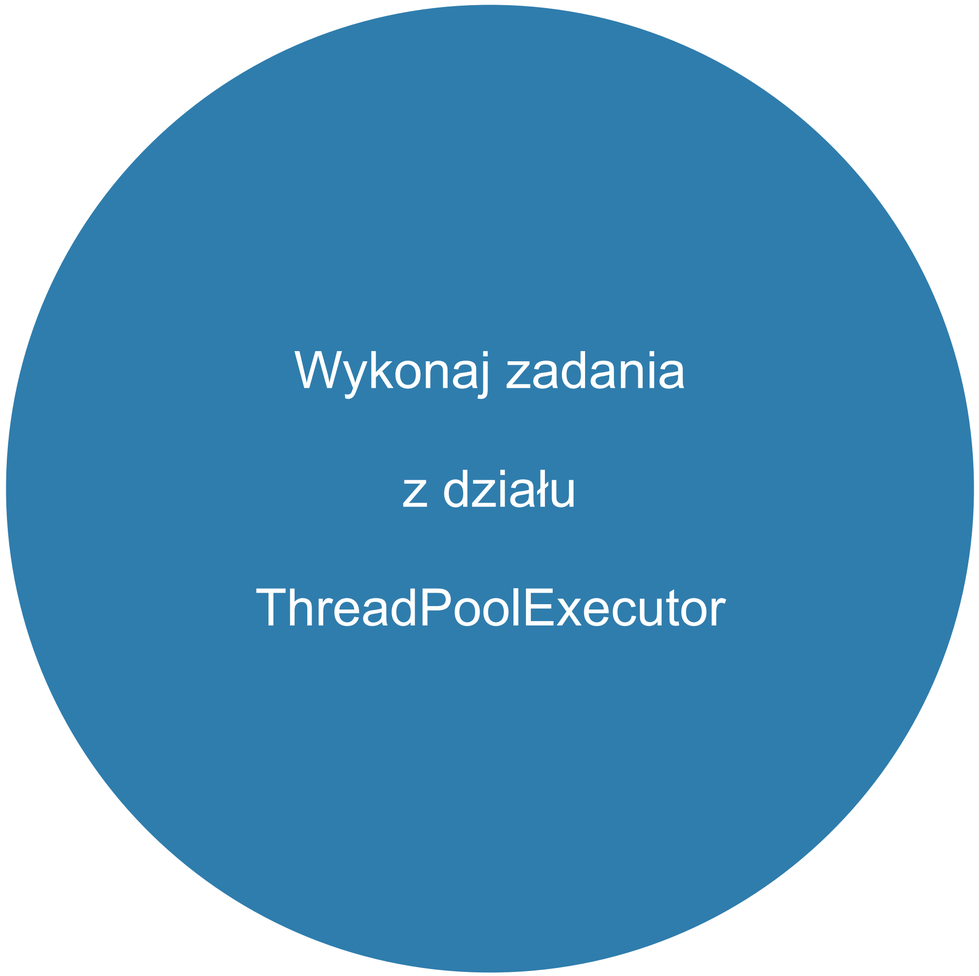
Ważne aby ThreadPoolExecutor zakończyć metodą shutdown(). W przypadku wywołania metody **`shutdown()`** i przekazaniu kolejnego zadania do executor-a zostanie wyrzucony błąd **`RejectedExecutionException`**

ThreadPoolExecutor

Klasa ThreadPoolExecutor oferuje przydatne metody:

- **`getPoolSize()`** - pobiera wielkość puli wątków
- **`getActiveCount()`** - pobiera liczbę wątków
- **`getCompletedTaskCount()`** - pobiera liczbę zakończonych przez executor zadań
- **`getLargestPoolSize()`** - pobiera maksymalną liczbę wątków która znajdowała się w puli wątków
- **`shutdownNow()`** - od razu zamyka executor nie wykonując oczekujących zadań, kontynuując rozpoczęte już zadania
- **`isTerminated()`** - zwraca prawdę po wywołaniu metod shutdown() oraz shutdownNow()
- **`isShutdown()`** - zwraca prawdę po wywołaniu metody shutdown()

Zadania



Scheduled Executor Service

ScheduledExecutorService

ScheduledExecutorService jest obiektem typu ExecutorService który może zaplanować wykonanie zadań z opóźnieniem lub wykonać je wielokrotnie z ustawionym czasem pomiędzy każdym wykonaniem

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},
5,
TimeUnit.SECONDS);
```

ScheduledExecutorService

ScheduledExecutorService jest obiektem typu ExecutorService który może zaplanować wykonanie zadań z opóźnieniem lub wykonać je wielokrotnie z ustawionym czasem pomiędzy każdym wykonaniem

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},
5,
TimeUnit.SECONDS);
```

tworzony jest obiekt ScheduledExecutorService z 5 wątkami

ScheduledExecutorService

ScheduledExecutorService jest obiektem typu ExecutorService który może zaplanować wykonanie zadań z opóźnieniem lub wykonać je wielokrotnie z ustawionym czasem pomiędzy każdym wykonaniem

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},
5,
TimeUnit.SECONDS);
```

tworzona implementacja interfejsu Callable i przekazywana do metody schedule()

ScheduledExecutorService

ScheduledExecutorService jest obiektem typu ExecutorService który może zaplanować wykonanie zadań z opóźnieniem lub wykonać je wielokrotnie z ustawionym czasem pomiędzy każdym wykonaniem

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},
5,
TimeUnit.SECONDS);
```

obiekt Callable powinien być wykonany po 5 sekundach

ScheduledExecutorService

ScheduledExecutorService jest obiektem typu ExecutorService który może zaplanować wykonanie zadań z opóźnieniem lub wykonać je wielokrotnie z ustawionym czasem pomiędzy każdym wykonaniem

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},
5,
TimeUnit.SECONDS);
```

określenie jednostki czasu dla poprzedniego parametru

ScheduledExecutorService

ScheduledExecutorService jest interfejsem dlatego aby go użyć trzeba zastosować implementację pakietu java.util.concurrent package i użyć implementacji ScheduledThreadPoolExecutor

Stworzenie ScheduledExecutorService zależy od implementacji, ale najprościej można użyć w tym celu fabryki Executors np:

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
```

Po utworzeniu instancji ScheduledExecutorService używamy jej za pomocą jednej z metod:

- schedule(Callable task, long delay, TimeUnit timeunit)
- schedule(Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate(Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay(Runnable, long initialDelay, long period, TimeUnit timeunit)

schedule(Callable task, long delay, TimeUnit timeunit)

Metoda ta powoduje zaplanowanie wykonania podanego obiektu Callable i wykonanie go po określonym w parametrze **`delay`** czasie

Metoda zwraca obiekt ScheduledFuture który można użyć do zatrzymania zadania zanim rozpoczął działanie lub otrzymać rezultat po wykonaniu zadania za pomocą metody **`get()`**

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture schFeature = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},5,TimeUnit.SECONDS);
System.out.println("returned value: " + schFeature.get());
sch.shutdown();
```

schedule(Callable task, long delay, TimeUnit timeunit)

Metoda ta powoduje zaplanowanie wykonania podanego obiektu Callable i wykonanie go po określonym w parametrze **`delay`** czasie

Metoda zwraca obiekt ScheduledFuture który można użyć do zatrzymania zadania zanim rozpoczął działanie lub otrzymać rezultat po wykonaniu zadania za pomocą metody **`get()`**

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture schFeature = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},5,TimeUnit.SECONDS);
System.out.println("returned value: " + schFeature.get());
sch.shutdown();
```

obiekt Callable powinien być wykonany po 5 sekundach

schedule(Callable task, long delay, TimeUnit timeunit)

Metoda ta powoduje zaplanowanie wykonania podanego obiektu Callable i wykonanie go po określonym w parametrze **`delay`** czasie

Metoda zwraca obiekt ScheduledFuture który można użyć do zatrzymania zadania zanim rozpoczął działanie lub otrzymać rezultat po wykonaniu zadania za pomocą metody **`get()`**

```
ScheduledExecutorService sch = Executors.newScheduledThreadPool(5);
ScheduledFuture schFeature = sch.schedule(new Callable() {
  public Object call() throws Exception {
    System.out.println("Executed!");
    return "Called!";
  }
},5,TimeUnit.SECONDS);
System.out.println("returned value: " + schFeature.get());
sch.shutdown();
```

pobranie wyników

schedule(Runnable task, long delay, TimeUnit timeunit)

Metoda ta działa działa bardzo podobnie do poprzedniej metody z tym że jako parametr pobiera obiekt typu Runnable(który jak już wiemy nie może zwrócić wartości) zamiast Callable

Metoda **`get()`** obiektu ScheduledFuture zwróci null w przypadku pozytywnego zakończenia zadania, w przypadku niepowodzenia zostanie wyrzucony wyjątek ExecutionException

scheduleAtFixedRate(Runnable, long initialDelay, long period, TimeUnit timeunit)

Metoda ta ma za zadanie zaplanować wykonywanie danego zadania w sposób cykliczny. Zadanie zostanie wykonane pierwszy raz po czasie podanym w parametrze **`initialDelay`**, następnie zadanie wykonywane jest powtarzalnie za każdym razem gdy upływa okres podany w parametrze **`period`**

Jeżeli którekolwiek z wywołań danego zadania wyrzuci błąd to kolejne wywołania nie będą kontynuowane. Natomiast jeżeli wykonanie danego zadania zajmie więcej czasu niż czas określony w parametrze **`period`** kolejne wykonanie zadania rozpocznie się po zakończeniu aktualnie wykonywanego zadania

scheduleWithFixedDelay(Runnable, long initialDelay, long period, TimeUnit timeunit)

Metoda ta działa podobnie do metody **`scheduledAtFixedRate()`** z tym że parametr **`period`** jest inaczej interpretowany ponieważ jest on brany pod uwagę jako czas od zakończenia poprzedniego wykonywania zadania i jest liczony dopiero od tego czasu

Ważne: należy zawsze pamiętać aby tak jak w **`ExecutorService`** tak również w **`ScheduledExecutorService`** poprawnie zamykać jego obiekt. W przeciwnym wypadku JVM będzie ciągle uruchomione nawet jeżeli wszystkie wątki zakończą swoje działanie. Aby zakończyć działanie obiektu **`ScheduledExecutorService`** używamy metod dziedziczonych z interfejsu **`ExecutorService`** czyli poznaną wcześniej metodę **shutdown()** lub ewentualnie **shutdownNow()**

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Klasa wykonująca zadania:

```
class Task implements Runnable {
  private String name;
  public Task(String name) {
    this.name = name;
  }
  @Override
  public void run() {
    try {
      System.out.println("Doing some work here");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek pierwszy - wykonanie zadania po określonym czasie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(2);
  Task task1 = new Task("Demo Task 1");
  Task task2 = new Task("Demo Task 2");
  ex.schedule(task1, 5, TimeUnit.SECONDS);
  ex.schedule(task2, 10, TimeUnit.SECONDS);
  ex.shutdown();
}
```

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek pierwszy - wykonanie zadania po określonym czasie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(2);
  Task task1 = new Task("Demo Task 1");
  Task task2 = new Task("Demo Task 2");
  ex.schedule(task1, 5, TimeUnit.SECONDS);
  ex.schedule(task2, 10, TimeUnit.SECONDS);
  ex.shutdown();
}
```

przekazanie obiektów zadań do obiektu ScheduledExecutorService

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek pierwszy - wykonanie zadania po określonym czasie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(2);
  Task task1 = new Task("Demo Task 1");
  Task task2 = new Task("Demo Task 2");
  ex.schedule(task1, 5, TimeUnit.SECONDS);
  ex.schedule(task2, 10, TimeUnit.SECONDS);
  ex.shutdown();
}
```

obiekt task1 będzie czekał 5 sekund na uruchomienie

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek pierwszy - wykonanie zadania po określonym czasie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(2);
  Task task1 = new Task("Demo Task 1");
  Task task2 = new Task("Demo Task 2");
  ex.schedule(task1, 5, TimeUnit.SECONDS);
  ex.schedule(task2, 10, TimeUnit.SECONDS);
  ex.shutdown();
}
```

obiekt task2 będzie czekał 10 sekund na uruchomienie

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

metoda scheduleAtFixedRate przyjmuje 4 parametry

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

pierwszy parametr - zadanie do wielokrotnego wykonania

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

drugi parametr - opóźnienie przy pierwszym uruchomieniu

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

trzeci parametr - czas pomiędzy kolejnymi wykonaniami

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Przypadek drugi - wykonywanie zadania regularnie:

```
public static void main(String[] args) {
  ScheduledExecutorService ex = Executors.newScheduledThreadPool(1);
  Task task1 = new Task("Task_1");
  ex.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
  try {
    TimeUnit.MILLISECONDS.sleep(20000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ex.shutdown();
}
```

czwarty parametr - jednostkę czasu drugiego i trzeciego parametru

Praktyczny przykład użycia ScheduledThreadPoolExecutor

Jeżeli utworzona przez nas wcześniej klasa wykonująca zadania implementowałaby interfejs **`Callable`** zamiast **`Runnable`** moglibyśmy użyć poniższej konstrukcji do pobrania wyniku działania zadania, np.:

```
ScheduledFuture<String> result =
  executor.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);
```

Zadania

Java Collections

Java Collections i wielowątkowość

Większość kolekcji które oferuje Java czyli np. ArrayList, LinkedList, HashMap, HashSet, TreeMap, TreeSet **nie są synchronizowane** a tym samym **nie są bezpieczne** przy pracy z kodem wielowątkowym

W pakiecie java.util poza **`Vector`** i **`Hashtable`** nie ma kolekcji synchronizowanych. Taka sytuacja jest wynikiem tego, że synchronizacja jest kosztowna w rozumieniu szybkości działania tych kolekcji

Kolekcje takie jak **`List`**, **`Set`** i **`Map`** nie posiadają wsparcia dla wielowątkowości, aby zapewnić maksymalną wydajność w aplikacjach jednowątkowych

Dla porównania wydajności wykonaj kod z dwóch następnych slajdów, aby porównać szybkość wykonywania operacji na kolekcjach Vector i ArrayList przy dużej liczbie operacji.

ArrayList będzie średnio około dwa razy szybsza od kolekcji Vector

Porównanie wydajności Vector i ArrayList

Użycie klasy Vector:

```
public void testVector() {
  long start = System.currentTimeMillis();
  Vector<Integer> vector = new Vector<>();
  for (int i = 0; i < 10000000; i++) {
    vector.addElement(i);
  }
  long end = System.currentTimeMillis();
  long total = end - start;
  System.out.println("Test Vector: " + total + " ms");
}
```

Porównanie wydajności Vector i ArrayList

Użycie klasy ArrayList:

```
public void testArrayList() {
  long startTime = System.currentTimeMillis();
  List<Integer> list = new ArrayList<>();
  for (int i = 0; i < 10000000; i++) {
    list.add(i);
  }
  long endTime = System.currentTimeMillis();
  long totalTime = endTime - startTime;
  System.out.println("Test ArrayList: " + totalTime + " ms");
}
```

Iteratory Fail-Fast

Podczas pracy z kolekcjami i wielowątkowością ważnym pojęciem są iteratory tzw. **Fail-Fast** czyli iteratory które od razu sygnalizują konflikt w danej kolekcji

W przypadku dwóch wątków które będą chciały korzystać z jednej kolekcji gdzie jeden wątek będzie dodawał/usuwał elementy a drugi iterował po niej od razu po wykryciu takiego działania zostanie wyrzucony błąd **`ConcurrentModificationException`**

To właśnie oznacza że iterator jest Fail-Fast czyli od razu informuje o tym że coś jest nie tak z kolekcją którą obsługuje

Dzieje się tak ponieważ iterator zapobiega groźnej sytuacji że kolekcja mogłaby mieć więcej, mniej lub w ogóle mogłaby nie mieć elementów po przejściu iteratora

Iteratory Fail-Fast - przykład

Jedna metoda dodaje elementy do współdzielonej w klasie listy:

```
public void updateThreadList() {
  Thread thread1 = new Thread(new Runnable() {
    public void run() {
      for(int i = 1000; i < 10000; i++) {
        list.add(i);
      }
    }
  });
  thread1.start();
}
```

Iteratory Fail-Fast - przykład

Druga metoda w tym czasie używa iteratora:

```
public void iterateThreadList() {
  Thread thread2 = new Thread(new Runnable() {
    public void run() {
      ListIterator<Integer> iterator = list.listIterator();
      while(iterator.hasNext()) {
        Integer number = iterator.next();
        System.out.println(number);
      }
    }
  });
  thread2.start();
}
```

Synchronized Wrappers

Jak już wiemy, użycie zwykłych kolekcji w kodzie wielowątkowym może wywołać niepożądane działania. Można oczywiście użyć bloków synchronized, które poznaliśmy wcześniej, ale lepiej korzystać zawsze z gotowych mechanizmów.

W tym celu Java oferuje w klasie Collections metody dzięki którym możemy używać zwykłych kolekcji w bezpieczny sposób. Używamy w tym celu **Collections.synchronizedName(collection)** - gdzie zamiast słowa **Name** stosujemy nazwę konkretnej kolekcji np. Collection, List, Map, Set a w argumencie metody podajemy wcześniej utworzony obiekt danej kolekcji

Synchronized Wrappers - przykład użycia

```
List<String> arr = new ArrayList<>();
List<String> list =
  Collections.synchronizedList(arr);
```

lub np:

```
Map<Integer, String> hm =
  new HashMap<>();
Map<Integer, String> safeMap =
  Collections.synchronizedMap(hm);
```

Jak widzimy metody synchronizujące "opakowują"(ang. wrap) utworzone kolekcje, więc stąd nazwa **Synchronized Wrappers**

Należy jeszcze zwrócić uwagę, że mimo stosowania Synchronized Wrappers, iteratory tych obiektów nadal są typu Fail-Fast dlatego należy pamiętać o bloku synchronized korzystając z nich:

```
synchronized(list) {
  while(iterator.hasNext()) {
    String n = iterator.next();
    System.out.println(n);
  }
}
```

Concurrent Collections

Mimo możliwości stosowania jednowątkowych kolekcji i Synchronized Wrappers Java w pakiecie java.util.concurrent udostępnia wielowątkowe kolekcje które podzielone są na 3 grupy bazując na ich mechanizmach bezpieczeństwa:

Pierwsza grupa to copy-on-write collections

- ten typ kolekcji przechowuje wartości w niezmiennej tablicy
- każda zmiana w kolekcji powoduje stworzenie nowej tablicy z odpowiednimi wartościami
- kolekcje te są zaprojektowane do rozwiązań gdzie operacje odczytu występują częściej niż zapisywania
- implementacje tego typu to np. CopyOnWriteArrayList lub CopyOnWriteArraySet
- takie kolekcje zawierają iteratory typu snapshot które nie wyrzucają błędu ConcurrentModificationException ponieważ działają na niezmiennych tablicach jak to było wspomniane na początku

Concurrent Collections

Druga grupa kolekcji to Compare-And-Swap or CAS collections

- udostępniają bezpieczeństwo dla kodu wielowątkowego przez tzw. algorytm Compare-And-Swap (CAS)
- przy wykonywaniu operacji na zmiennych wykonywana jest lokalna kopia zmiennej
- przed aktualizacją wartości zmiennej porównywane są wartości przed działaniem i aktualna, a gdy są takie same następuje zmiana wartości
- jeżeli wartości na początku i aktualna nie są takie same, mechanizm powtarza całą czynność
- przykłady takich kolekcji to ConcurrentLinkedQueue oraz ConcurrentSkipListMap
- ich iteratory również nie wyrzucają błędu ConcurrentModificationException

Concurrent Collections

Trzecia grupa kolekcji to wielowątkowe kolekcje używające specjalnej blokady obiektowej(java.util.concurrent.lock.Lock)

- ten mechanizm jest bardziej elastyczny niż klasyczna synchronizacja
- ma podobne zastosowanie jak klasyczne blokady obiektów ale blokada jest utrzymywana dopóki nie zostanie użyta metoda unlock()
- przykładem jest LinkedBlockingQueue, która ma osobne metody blokady początku i końca kolejki, obiekty w niej zawarte mogą być dodawane i usuwane równolegle.
- innym przykładem jest ConcurrentHashMap i większość implementacji BlockingQueue
- iteratory dla tej grupy również nie wyrzucają błędów ConcurrentModificationException

Podsumowanie klas Java Concurrent Collection

- **BlockingQueue** – interfejs, który jest podstawą dla wszystkich wielowątkowych kolekcji opartych o kolejkę. Kiedy dodajemy element do BlockingQueue i jeżeli nie ma miejsca w kolejce element będzie oczekiwał do czasu zwolnienia miejsca, a przy pobieraniu elementów nastąpi oczekiwanie, dopóki kolejka jest pusta
- **ArrayBlockingQueue** – blokująca kolejka oparta o stałą wielkość tablicy, jednokrotnie zadeklarowana nie może być zmieniona
- **SynchronousQueue** – blokująca kolejka z pojemnością zero - przed dodaniem następuje oczekiwanie na opróżnienie kolejki
- **PriorityBlockingQueue** – blokująca kolejka oparta o priorytety
- **LinkedBlockingQueue** – kolekcja szeregująca elementy na zasadzie FIFO - pierwsze weszło - pierwsze wyszło
- **DelayQueue** – kolekcja, w której każdy element kolejki ma wartość czasową przed upłynięciem której nie może zostać pobrany
- **BlockingDeque** – interfejs, który rozszerza BlockingQueue i dodaje operacje związane z Deque

Podsumowanie klas Java Concurrent Collection

- **LinkedBlockingDeque** – implementacja interfejsu BlockingDequeue
- **TransferQueue** – interfejs rozszerzający interfejs BlockingQueue i dodający metody gdzie producent będzie czekał aż konsument otrzyma elementy w kolejce
- **LinkedTransferQueue** – implementacja interfejsu TransferQueue
- **ConcurrentMap** – wielowątkowy interfejs kolekcji o typie Map który dostarcza bezpieczeństwo pod kątem takiego właśnie kodu
- **ConcurrentHashMap** – implementacja interfejsu ConcurrentMap
- **ConcurrentNavigableMap** – interfejs który rozszerza ConcurrentMap i dodaje operacje z interfejsu NavigableMap
- **ConcurrentSkipListMap** – implementacja interfejsu ConcurrentNavigableMap

HashMap i zamienniki w środowisku wielowątkowym

HashMap

- nie jest bezpieczne przy pracy z wieloma wątkami
- podczas wykonywania metod put() czy remove() przez jeden wątek, i odczycie przez inny może nastąpić niespójność danych
- aby korzystać z HashMap i wielowątkowości należy używać synchronizacji biorąc pod uwagę przypadki DeadLock
- nie jest zalecane korzystanie z HashMap i wielowątkowości

HashMap i zamienniki w środowisku wielowątkowym

HashTable

- jest dosyć starą kolekcją chociaż nadal używaną
- jest bezpieczna pod kątem wielowątkowości
- jednocześnie tylko jeden wątek może czytać lub zapisywać do HashTable
- blokada następuje na poziomie całej HashTable
- wydajność HashTable jest na niskim poziomie

HashMap i zamienniki w środowisku wielowątkowym

Collections.SynchronizedMap

- całkiem podobna w działaniu do HashTable i również blokada zachodzi na całej instancji mapy
- jest nowsza od HashTable
- jednocześnie odczytywać lub zapisywać dane może tylko jeden wątek
- posiada funkcjonalność przekształcenia dowolnej "zwykłej" mapy na bezpieczną pod kątem wielowątkowym

Przykład użycia Collections.synchronizedMap

Metoda Collections.synchronizedMap praktycznie w swojej implementacji opakowuje wszystkie metody interfejsu Map w bloki synchronized

Przykład użycia Collections.synchronizedMap

```
Map<String,String> map = new HashMap<String,String>();
   map.put("1","One");
   map.put("2","Two");
   map.put("3","Three");
Map<String,String> syncmap = Collections.synchronizedMap(map);
```

Przykład użycia Collections.synchronizedMap

Metoda Collections.synchronizedMap praktycznie w swojej implementacji opakowuje wszystkie metody interfejsu Map w bloki synchronized

Przykład użycia Collections.synchronizedMap

```
Map<String,String> map = new HashMap<String,String>();
   map.put("1","One");
   map.put("2","Two");
   map.put("3","Three");
Map<String,String> syncmap = Collections.synchronizedMap(map);
```

utworzenie nowej HashMap-y

Przykład użycia Collections.synchronizedMap

Metoda Collections.synchronizedMap praktycznie w swojej implementacji opakowuje wszystkie metody interfejsu Map w bloki synchronized

Przykład użycia Collections.synchronizedMap

```
Map<String,String> map = new HashMap<String,String>();
   map.put("1","One");
   map.put("2","Two");
   map.put("3","Three");
Map<String,String> syncmap = Collections.synchronizedMap(map);
```

wypełnienie HaspMap-y

Przykład użycia Collections.synchronizedMap

Metoda Collections.synchronizedMap praktycznie w swojej implementacji opakowuje wszystkie metody interfejsu Map w bloki synchronized

Przykład użycia Collections.synchronizedMap

```
Map<String,String> map = new HashMap<String,String>();
   map.put("1","One");
   map.put("2","Two");
   map.put("3","Three");
Map<String,String> syncmap = Collections.synchronizedMap(map);
```

opakowanie HashMap-y metodą Collections.synchronizedMap

ConcurrentHashMap

`ConcurrentHashMap` charakteryzuje się dobrą wydajnością ponieważ więcej niż jeden wątek może odczytywać na raz dane z ConcurrentHashMap

ConcurrentHashMap dzieli mapę na osobne segmenty, domyślnie jest ich 16 każdy obsługiwany przez jeden wątek jednocześnie dlatego jednocześnie do jednego segmentu może zapisywać jeden wątek. ale w tym samym czasie inny wątek może pisać do innego segmentu

Można zwiększać ilość wątków obsługujących segmenty, np. jeżeli mamy większą mapę możemy przydzielić więcej wątków do jej obsługi. Co ważne podczas zapisu do segmentu przez jeden z wątków drugi może odczytywać dane z tego samego segmentu

Podsumowanie Java Concurrent Collections

- należy unikać HashMap w środowisku wielowątkowym
- HashTable i SynchronizedMap są całkowicie blokowane podczas odczytywania, natomiast ConcurrentHashMap jest dzielona na segmenty i blokowana
- HashTable i SynchronizedMap mogą być obsługiwane tylko przez jeden wątek jednocześnie, natomiast ConcurrentHashMap może być obsługiwany przez wiele wątków jednocześnie

Zadania

KONIEC