ВИКОРИСТАННЯ ЕКЗЕКУТОРІВ ПРИ РОЗРОБЦІ БАГАТОПОТОЧНИХ ДОДАТКІВ

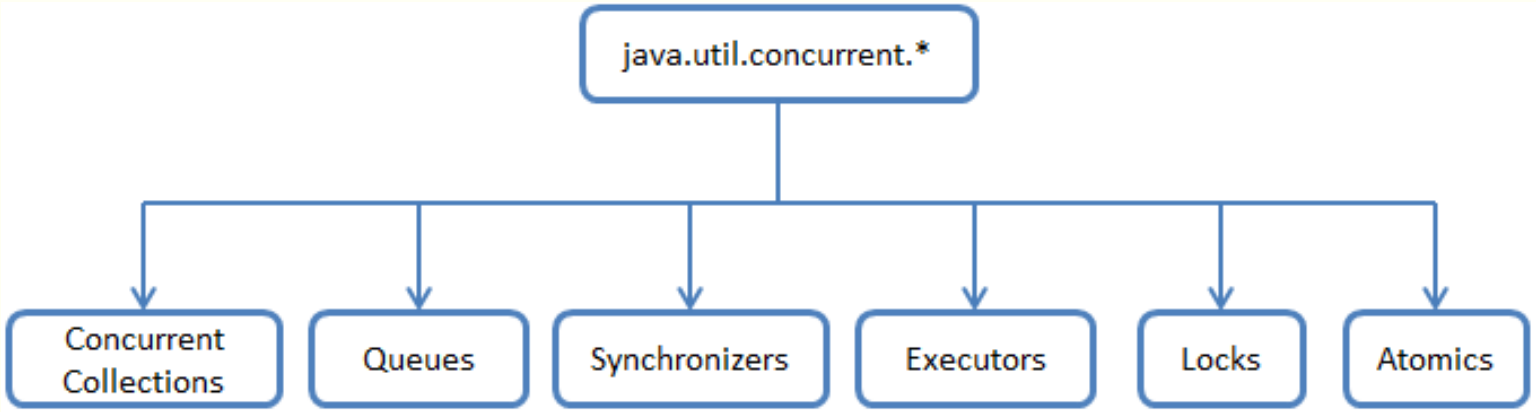
Питання 6.3.

Складнощі з Java Threads API призвели до створення більш потужного Concurrency Utilities Framework

- У Java Threads API закладені наступні проблеми:
  - ***Низькорівневі примітиви багатопоточності, наприклад, synchronized і wait()/notify(), часто важко використовувати коректно***. Результат некоректного використання - race conditions, thread starvation, deadlock та інші небезпеки, які важко відстежити.
  - ***Якщо занадто покладатись на synchronized, можуть виникнути проблем з продуктивністю, які впливають на масштабованість (scalability) додатку***. Це значна проблема для високо багатопоточних додатків, наприклад, веб-серверів.
  - ***Розробникам часто потрібно використовувати високорівневі конструкти, наприклад, пули потоків та семафори***. Розробники були змушені створити власні конструкти.

Для усунення цих проблем у Java 5 було представлено Concurrency Utilities

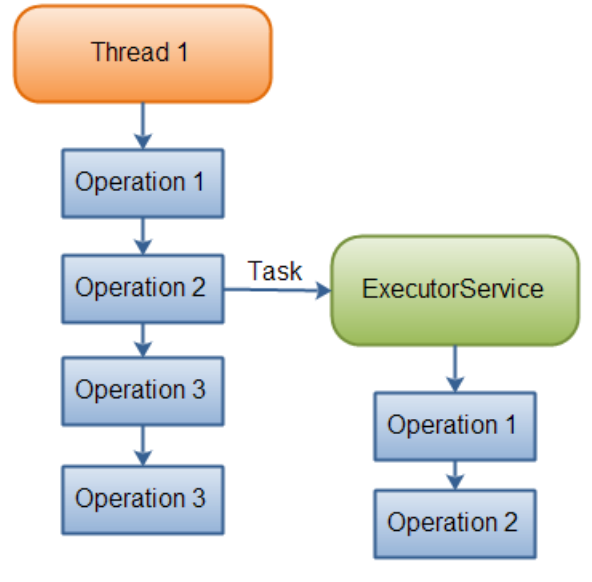
- Потужний та розширюваний фреймворк з можливостями високопродуктивного розпаралелювання, наприклад, пулами потоків та блокуючими чергами.
- Фреймворк складається з наступних пакетів:
  - ***java.util.concurrent*** містить типи утиліт, які часто використовуються в багатопоточному програмуванні, наприклад, екзекутори, пули потоків та concurrent hashmaps.
  - ***java.util.concurrent.atomic*** містить utility-класи, які підтримують потокобезпечне програмування на єдиних змінних без замків.
  - ***java.util.concurrent.locks*** містить utility-типи для блокування та очікування на основі *умов (conditions)*.
    - Блокування та очікування за допомогою цих типів надає кращу продуктивність та гнучкість, ніж застосування базових механізмів синхронізації на основі моніторів та очікування/сповіщення.

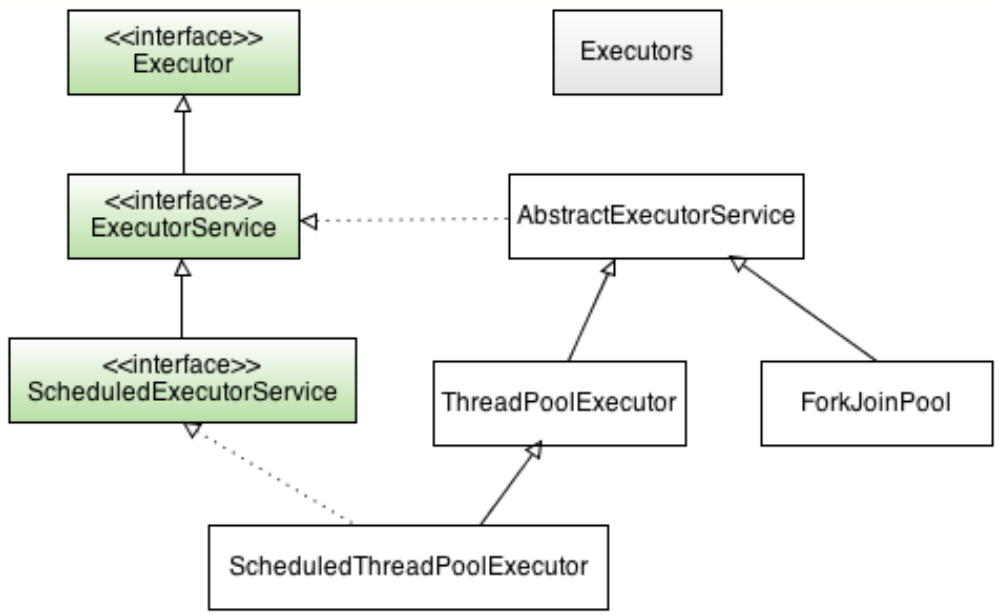


Екзекутори

- Threads API дозволяє виконувати runnable-задачі за допомогою виразів на зразок
  - new java.lang.Thread(new RunnableTask()).start();
- Такі вирази тісно пов’язують постановку задач (task submission) з механікою виконання задачі (запуск на конкретному потоці, новому потоці або на довільному потоці з pool[group] потоків).
  - **Зауважте!** Задача (*task)* – об’єкт, чий клас реалізує інтерфейс java.lang.Runnable (runnable task) або інтерфейс java.util.concurrent.Callable (callable task).
- The concurrency-oriented utilities постачають екзекутори в якості високорівневої альтернативи виразів низькорівневого Threads API для виконання runnable-задач.
  - Екзекутор – це об’єкт, чий клас напряму або опосередковано реалізує інтерфейс java.util.concurrent.Executor, який відділяє розподіл задач від їх виконання.
  - **Зауважте!** Використання інтерфейсів екзекуторами для decouple task submission from task-execution mechanics аналогічне до використання інтерфейсів у Collections Framework, щоб відокремити списки, сети, черги та мепи від їх реалізацій.
  - Результат відокремлення – гнучкий код, який легко підтримувати.

- Екзекутор оголошує єдиний метод void execute(Runnable runnable), який виконає runnable-задачу runnable в деякий момент в майбутньому.
  - execute() викидає java.lang.NullPointerException, коли runnable не null, і java.util.concurrent.RejectedExecutionException, якщо не може виконати runnable.
- **Зауважте!** RejectedExecutionException може викидатись, коли екзекутор завершує роботу і не хоче приймати нових задач.
  - Також це виключення може викидатись, коли екзекутор не володіє достатнім місцем, щоб зберегти задачу





- Приклад показує еквівалентність виразу
  new Thread(new RunnableTask()).start(); вже згаданому виразу:
  Executor executor = ...; // ... Представляє створення екзекутора
  executor.execute(new RunnableTask());

- Хоч Executor простий у використанні, цей інтерфейс обмежений:
  - ***Executor фокусується лише на Runnable***. Оскільки метод Runnable.run() не повертає значення, зручного способу для runnable-задачі повернути значення тому, хто її викликав, немає.
  - ***Executor не постачає спосіб відстеження прогресу чи відміни runnable-задач***, які виконуються, і не дає можливість визначити, чи завершила виконання runnable-задача.
  - ***Executor не може виконувати колекцію runnable-задач***.
  - ***Executor не забезпечує застосунку можливості коректного припинення роботи екзекутора***.

- Ці обмеження враховуються інтерфейсом java.util.concurrent.ExecutorService, який розширяє Executor і зазвичай реалізується у вигляді пулу потоків.

Методи ExecutorService

Method	Description
`boolean awaitTermination(long timeout, TimeUnit unit)`	Blocks (waits) until all tasks have finished after a shutdown request, the `timeout` (measured in `unit` time units) expires, or the current thread is interrupted, whichever happens first. Returns true when this executor has terminated and false when the `timeout` elapses before termination. This method throws `java.lang.InterruptedException` when interrupted.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`	Executes each callable task in the `tasks` collection and returns a `java.util.List` of `java.util.concurrent.Future` instances that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception. The `List` of `Futures` is in the same sequential order as the sequence of tasks returned by `tasks`' iterator. This method throws `InterruptedException` when it's interrupted while waiting, in which case unfinished tasks are canceled; `NullPointerException` when `tasks` or any of its elements is `null`; and `RejectedExecutionException` when any one of `tasks`' tasks cannot be scheduled for execution.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`	Executes each callable task in the `tasks` collection and returns a `List` of `Future` instances that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception—or the `timeout` (measured in `unit` time units) expires. Tasks that are not completed at expiry are canceled. The `List` of `Futures` is in the same sequential order as the sequence of tasks returned by `tasks`' iterator. This method throws `InterruptedException` when it's interrupted while waiting, in which case unfinished tasks are canceled. It also throws `NullPointerException` when `tasks`, any of its elements, or `unit` is `null`; and throws `RejectedExecutionException` when any one of `tasks`' tasks cannot be scheduled for execution.

`<T> T invokeAny(Collection<? extends Callable<T>> tasks)`	Executes the given `tasks`, returning the result of an arbitrary task that's completed successfully (in other words, without throwing an exception), if any does. On normal or exceptional return, tasks that haven't completed are canceled. This method throws `InterruptedException` when it's interrupted while waiting, `NullPointerException` when `tasks` or any of its elements is `null`, `java.lang.IllegalArgumentException` when `tasks` is empty, `java.util.concurrent.ExecutionException` when no task completes successfully, and `RejectedExecutionException` when none of the tasks can be scheduled for execution.
`<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`	Executes the given `tasks`, returning the result of an arbitrary task that's completed successfully (in other words, without throwing an exception), if any does before the `timeout` (measured in `unit` time units) expires—tasks that are not completed at expiry are canceled. On normal or exceptional return, tasks that have not completed are canceled. This method throws `InterruptedException` when it's interrupted while waiting; `NullPointerException` when `tasks`, any of its elements, or `unit` is `null`; `IllegalArgumentException` when `tasks` is empty; `java.util.concurrent.TimeoutException` when the `timeout` elapses before any task successfully completes; `ExecutionException` when no task completes successfully; and `RejectedExecutionException` when none of the tasks can be scheduled for execution.

Method	Description
`boolean isShutdown()`	Returns true when this executor has been shut down; otherwise, returns false.
`boolean isTerminated()`	Returns true when all tasks have completed following shutdown; otherwise, returns false. This method will never return true prior to `shutdown()` or `shutdownNow()` being called.
`void shutdown()`	Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Calling this method has no effect after the executor has shut down. This method doesn't wait for previously submitted tasks to complete execution. Use `awaitTermination()` when waiting is necessary.
`List<Runnable> shutdownNow()`	Attempts to stop all actively executing tasks, halt the processing of waiting tasks, and return a list of the tasks that were awaiting execution. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via `Thread.interrupt()`, so any task that fails to respond to interrupts may never terminate.
`<T> Future<T> submit(Callable<T> task)`	Submits a callable `task` for execution and returns a `Future` instance representing `task`'s pending results. The `Future` instance's `get()` method returns `task`'s result on successful completion. This method throws `RejectedExecutionException` when `task` cannot be scheduled for execution and `NullPointerException` when `task` is `null`. If you would like to block immediately while waiting for a task to complete, you can use constructions of the form `result = exec.submit(aCallable).get();`.
`Future<?> submit(Runnable task)`	Submits a runnable `task` for execution and returns a `Future` instance representing `task`'s pending results. The `Future` instance's `get()` method returns `task`'s result on successful completion. This method throws `RejectedExecutionException` when `task` cannot be scheduled for execution and `NullPointerException` when `task` is `null`.
`<T> Future<T> submit(Runnable task, T result)`	Submits a runnable `task` for execution and returns a `Future` instance whose `get()` method returns `result` on successful completion. This method throws `RejectedExecutionException` when `task` cannot be scheduled for execution and `NullPointerException` when `task` is `null`.

- java.util.concurrent.TimeUnit – перелічення, яке представляє одиниці вимірювання часу: DAYS, HOURS, MICROSECONDS, MILLISECONDS, MINUTES, NANOSECONDS, SECONDS.
  - Крім того, TimeUnit оголошує методи для конвертації одиниць вимірювання (наприклад, long toHours(long duration)) та виконання операцій timing and delay (such as void sleep(long timeout)).
- *callable-задачі* аналогічні runnable-задачам.
  - На відміну від Runnable, Callable<V> оголошує метод V call(), що повертає значення та може викидати checked-виключення,.
- інтерфейс Future представляє результат асинхронних обчислень.
  - Результат називають *future*, оскільки зазвичай він не буде доступним до певного моменту в майбутньому.
  - Future (узагальнений тип Future<V>) постачає методи для відміни задачі, повернення значення від задачі та визначення того, чи завершена вона.

Table 10-2. Future Methods

Method	Description
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel execution of this task and returns true when the task was canceled; otherwise, returns false (perhaps the task completed normally before this method was called). The cancellation attempt fails when the task is done, canceled, or couldn't be canceled for another reason. If successful and this task hadn't started when `cancel()` was called, the task should never run. If the task has started, `mayInterruptIfRunning` determines whether (true) or not (false) the thread executing this task should be interrupted in an attempt to stop the task. After returning, subsequent calls to `isDone()` always return true; `isCancelled()` always return true when `cancel()` returns true.
`V get()`	Waits if necessary for the task to complete and then returns the result. This method throws `java.util.concurrent.CancellationException` when the task was canceled prior to this method being called, `ExecutionException` when the task threw an exception, and `InterruptedException` when the current thread was interrupted while waiting.
`V get(long timeout, TimeUnit unit)`	Waits at most `timeout` units (as specified by `unit`) for the task to complete and then returns the result (if available). This method throws `CancellationException` when the task was canceled prior to this method being called, `ExecutionException` when the task threw an exception, `InterruptedException` when the current thread was interrupted while waiting, and `TimeoutException` when this method's `timeout` value expires (the wait times out).
`boolean isCancelled()`	Returns true when this task was canceled before it completed normally; otherwise, returns false.
`boolean isDone()`	Returns true when this task completed; otherwise, returns false. Completion may be due to normal termination, an exception, or cancellation; this method returns true in all of these cases.

Сценарій використання екзекутора

- Припустимо, необхідно написати додаток, чий графічний інтерфейс дозволяє користувачу ввести слово.
  - Після вводу додаток доповнює цим словом кілька онлайн-словників та отримує each dictionary's entry.
  - Ці входження послідовно виводяться користувачу.
  - Оскільки онлайн-доступ може бути повільним, а інтерфейс користувача має залишатись респонсивним, задача "отримати word entries" розвантажується за допомогою екзекутора, який запускає її в окремому потоці.

```
ExecutorService executor = ...; //  ... represents some executor creation
Future<String[]> taskFuture = executor.submit(new Callable<String[]>()
                                              {
                                                  @Override
                                                  public String[] call()
                                                  {
                                                     String[] entries = ...;
                                                     // Access online dictionaries
                                                     // with search word and populate
                                                     // entries with their resulting
                                                     // entries.
                                                     return entries;
                                                  }
                                              });
// Do stuff.
String entries = taskFuture.get();
```

- Після отримування екзекутора головний потік передає callable-задачу екзекутору.
  - Метод submit() негайно повертає посилання на об'єкт Future для контролю виконання задачі та доступу до результатів.
  - Головний потік викликає метод get() цього об'єкту для отримування результатів.

Зауважте!

- Інтерфейс java.util.concurrent.ScheduledExecutorService розширює ExecutorService та описує екзекутор, який дозволяє планувати задачі для одиничного чи періодичного запуску.
- Хоч можна створити власні реалізації Executor, ExecutorService та ScheduledExecutorService, наприклад,

```
class DirectExecutor implements Executor {
    @Override
    public void execute(Runnable r) { r.run(); }
}
```

запускає екзекутор напряму в calling thread,

існує простіша альтернатива: java.util.concurrent.Executors.

- **Підказка** Якщо Ви маєте на меті створити власні реалізації ExecutorService, розгляньте роботу з класами java.util.concurrent.AbstractExecutorService та java.util.concurrent.FutureTask.

- Utility-клас Executors оголошує кілька методів класу, які повертають екземпляри різних реалізацій ExecutorService та ScheduledExecutorService (та інші екземпляри).
- Статичні методи класу виконують наступні завдання:
  - Створюють та повертають екземпляри ExecutorService або ScheduledExecutorService, який сконфігуровано на базі поширених налаштувань.
  - Створюють та повертають “wrapped” екземпляри ExecutorService або ScheduledExecutorService, які відключають реконфігурацію служби-екзекутора, роблячи implementation-specific методи недоступними.
  - Створюють та повертають екземпляр java.util.concurrent.ThreadFactory (екземпляр класу, що реалізує інтерфейс ThreadFactory) для створення нових потоків.
  - Створюють та повертають екземпляр Callable, щоб його можна було використовувати в execution-методах, які вимагають Callable-аргументів (як метод submit(Callable) класу ExecutorService).

- Наприклад, static ExecutorService newFixedThreadPool(int nThreads) створює пул потоків, який повторно використовує фіксовану кількість потоків operating off of a shared unbounded queue.
  - Максимум nThreads потоків, що активно обробляють задачі.
  - Якщо відправляються додаткові задачі, коли всі потоки активні, вони чекають у черзі на доступний потік.
  - Якщо якийсь потік переривається через падіння в процесі виконання до завершення роботи екзекутора, новий потік займе його місце, коли буде потрібно виконувати наступні задачі.
- Потоки в пулі будуть існувати, поки екзекутор не буде явно завершено (shut down).
  - Метод викидає IllegalArgumentException, коли передається 0 або від'ємне значення nThreads.
- **Зауважте!** Пули потоків використовуються для усунення накладних витрат на створення нового потоку для кожної поставленої (submitted) задачі.
  - Створення потоку недешеве, а потреба в створенні багатьох потоків може суттєво вплинути на продуктивність додатку.

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CalculateE
{
   final static int LASTITER = 17;

   public static void main(String[] args)
   {
      ExecutorService executor = Executors.newFixedThreadPool(1);
      Callable<BigDecimal> callable;
      callable = new Callable<BigDecimal>()
                 {
                    @Override
                    public BigDecimal call()
                    {
                       MathContext mc = new MathContext(100,
                                                        RoundingMode.HALF_UP);
                       BigDecimal result = BigDecimal.ZERO;
                       for (int i = 0; i <= LASTITER; i++)
                       {
                          BigDecimal factorial = factorial(new BigDecimal(i));
                          BigDecimal res = BigDecimal.ONE.divide(factorial, mc);
                          result = result.add(res);
                       }
                       return result;
                    }
```

- Використовуються executors, runnables, callables та futures в контексті файлового та мережевого вводу/виводу.
  - Виконання тривалих обчислень пропонує інший сценарій, де можна використовувати ці типи.
  - Наступний лістинг використовує екзекутор, callable та future в контексті обчислення числа Ейлера e (2.71828. . .).
- Головний потік, який виконує метод main() з лістингу, спочатку отримує екзекутор за допомогою методу newFixedThreadPool().
  - Потім отримується екземпляр анонімного класу, який реалізує інтерфейс Callable і передає цю задачу (submit task) в екзекутор, отримуючи навзаєм екземпляр класу Future.

```
                public BigDecimal factorial(BigDecimal n)
                {
                   if (n.equals(BigDecimal.ZERO))
                      return BigDecimal.ONE;
                   else
                      return n.multiply(factorial(n.subtract(BigDecimal.ONE)));
                }
             };
    Future<BigDecimal> taskFuture = executor.submit(callable);
    try
    {
       while (!taskFuture.isDone())
          System.out.println("waiting");
       System.out.println(taskFuture.get());
    }
    catch(ExecutionException ee)
    {
       System.err.println("task threw an exception");
       System.err.println(ee);
    }
     catch(InterruptedException ie)
     {
        System.err.println("interrupted while waiting");
     }
     executor.shutdownNow();
  }
}
```

- Після передачі задачі потік зазвичай виконує іншу роботу, поки не знадобляться результати задачі.
  - Для симуляції такої роботи періодично буде виводитись повідомлення про очікування до того моменту, поки методу isDone() екземпляру класу Future не поверне true.

В реальних застосунках такого циклу уникають.
На даному етапі головний потік викликає геттер екземпляру класу для отримання результатів, які потім виводяться. У кінці головний потік припиняє роботу екзекутора.
Дуже важливо це виконати, оскільки застосунок може не завершити роботу.
З цією метою використовується не лише метод shutdownNow(), а і, як альтернатива, shutdown().

Робота програми та результати виводу

- Метод call() обчислює e, знаходячи суму ряду
  - e = 1 / 0! + 1/ 1! + 1 / 2! + ...
- Спочатку call() інстанціює java.math.MathContext для інкапсуляції точності (*precision*) та режиму округлення.
  - Обрано 100 в якості верхньої межі точності для e і режим округлення HALF_UP.
- Потім call() ініціалізує локальну змінну result типу java.math.BigDecimal значенням BigDecimal.ZERO.
  - Далі вона входить у цикл, який обчислює факторіал, ділить BigDecimal.ONE на значення факторіалу та додає результат ділення до суми.
- Метод divide() бере екземпляр MathContext в якості другого аргументу, щоб переконатись, що результат ділення не буде нескінченним дробом, через що могло б викидатись виключення java.lang.ArithmeticException.

```
waiting
waiting
waiting
waiting
waiting
2.7182818284590450705160477958486050611789796352510326989007350040652250425048433140558879743442457
41730039454062711
```