Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»
Филиал
«Минский радиотехнический колледж»

Учебный предмет
«Разработка кроссплатформенных приложений»

# Инструкция
по выполнению лабораторной работы
«Разработка, отладка и испытание асинхронных потоков»

Минск 2025 г.

## Лабораторная работа № 21

## Тема работы: «Разработка, отладка и испытание асинхронных потоков»

### 1 Цель работы

Сформировать умение разрабатывать программы с асинхронными потоками.

### 2 Задание

Необходимо на базе лабораторной работы №14 реализовать шифрование и дешифрование используя Future, Callable и ExecutorService.

### 3 Оснащение работы

Задание по варианту, ЭВМ, среда разработки IntelliJ IDEA.

### 4 Основные теоретические сведения

Время от времени появляется необходимость выполнить асинхронно (т.е. одновременно с основным действием программы выполняется другая задача) какой-нибудь код. Например, отправить письмо по e-mail или что-нибудь из базы данных прочитать. В Java для подобных операций можно создать поток и выполнить в нём нужное действие. Например, так:

```
private void simpleRun() throws ExecutionException,
InterruptedException {
        final CompletableFuture<String> future = new
CompletableFuture<>();
        new Thread( () -> {
            System.out.println("job started");
            sleep(3);
            future.complete("feature done");
            System.out.println("job done");
        }).start();
        System.out.println("waiting...");
        String result = future.get();
        System.out.println("finished,   result:"   +
result);
    }
```

**Что здесь происходит?**

Просто создаётся поток и имитируется задача длительностью 3 сек.: вызов **Thread.sleep(TimeUnit.SECONDS.toMillis(sec));** с перехваченным исключением. Задача возвращает результат через CompletableFuture.

Это вполне рабочий вариант, но можно сделать заметно проще:

```
private void asyncRun() throws ExecutionException,
InterruptedException {
        final CompletableFuture<String> future =
CompletableFuture.supplyAsync(()-> {
            System.out.println("job started");
            sleep(3);
            System.out.println("job done");
            return "feature done";
        });
        System.out.println("waiting...");
        String result = future.get();
        System.out.println("finished, result:" +
result);
    }
```

В этом коде используется интересная возможность **CompletableFuture** из пакета **java.util.concurrent**. **CompletableFuture** «умеет» запускать код в параллельном потоке, и при этом использует встроенный **thread pool**. Его использование упрощает код и даёт прирост в производительности приложения.

Ведь на создание нового потока не расходуются системные ресурсы. Более того, встроенный **thread pool** уже оптимизирован и работает эффективнее большинства пулов, созданных пользователем.

Однако надо помнить и об ограничениях. Например, разные модули программы могут конкурировать за потоки из общего пула, что может привести к дополнительным ожиданиям.

У **CompletableFuture** есть ещё интересные функции. Например, надо построить цепочку из асинхронных вызовов. Т.е. после завершения первой асинхронной функции запустить вторую, после второй третью и т.д. В JavaScript для этого применяются **promise**. В Java можно использовать **CompletableFuture**.

```
private void thenApply() throws ExecutionException,
InterruptedException {
        final CompletableFuture<String> future =
CompletableFuture.supplyAsync(()-> {
            System.out.println("job started");
            sleep(3);
            System.out.println("job done");
            return "feature done|";
        }).thenApply(result -> {
            System.out.println("applay result:" +
result);
            return result + " applied";
```

```
        });
        System.out.println("waiting...");
        String result = future.get();
        System.out.println("finished,   result:"   +
result);
    }
```

Сначала выполняется первая задача **(sleep(3))** и только после её завершения запустится вторая задача. Причём второй задаче в качестве входного параметра можно передать результаты первой задачи.

А что делать с ошибками? Как их обрабатывать? У **CompletableFuture** есть встроенный механизм для обработки ошибок, применить его можно так:

```
private     void     thenApplyException()     throws
ExecutionException, InterruptedException {
        final   CompletableFuture<String>   future   =
CompletableFuture.supplyAsync(()-> {
            System.out.println("job started");
            sleep(3);
            throw    new    RuntimeException("runTime
exception");
        }).thenApply(result -> {
            System.out.println("applay   result:"   +
result);
            return result + " applied";
        }).exceptionally(exception -> {
            System.out.println("got  exception,  err:"
+ exception.getMessage());
            return exception.getMessage();
        });
        System.out.println("waiting...");
        String result = future.get();
        System.out.println("finished,   result:"   +
result);
    }
```

*Обратите внимание на блок **exceptionally**, он запустится, если одна из задач выбросит исключение.*

У **CompletableFuture** есть ещё интересная возможность: запускать несколько асинхронных задач, подождать, когда они завершатся и обработать полученные результаты.

Пример:

```
private void acceptBoth() {
        final  CompletableFuture<String>  futureOne  =
CompletableFuture.supplyAsync(()-> {
```

```
            System.out.println("job one started");
            sleep(3);
            System.out.println("job one is done");
            return "feature done one";
        });
        final CompletableFuture<String> futureTwo =
CompletableFuture.supplyAsync(()-> {
            System.out.println("job two started");
            sleep(7);
            System.out.println("job two is done");
            return "feature done two";
        });
        System.out.println("waiting...");
        futureOne.thenAcceptBothAsync(futureTwo,
                (result1,         result2)         ->
System.out.println("join:"  +  result1  +  "  "  +
result2));
        System.out.println("end");
    }
```

Обратите внимание на структуру **futureOne.thenAcceptBothAsync**. Ждём, когда завершатся обе задачи, и обрабатываем итоговый результат. В отличие от предыдущих примеров, в этом фрагменте кода поток выполнения программы не ждёт, когда завершатся **future**, а идёт дальше.

**CompletableFuture** из пакета **java.util.concurrent** предоставляет полезный и простой в использовании функционал, который помогает ускорить разработку и упростить код. В то же время надо помнить, что в **java.util.concurrent** много тонких моментов, требующих понимания и некоторой сноровки в использовании.

## 5 Порядок выполнения работы

1. Выделить ключевые моменты задачи;
2. Построить алгоритм и теоритическую объектную модель решения задачи;
3. Запрограммировать полученный алгоритм и объектную модель;
4. Провести тестирование полученной программы.

## 6 Форма отчета о работе

*Лабораторная работа № ___*

*Номер учебной группы* _____

*Фамилия, инициалы учащегося* _____

*Дата выполнения работы* _____

*Тема работы:* _____
*Цель работы:* _____
*Оснащение работы:* _____
*Результат выполнения работы:* _____
_____
_____

**7. Контрольные вопросы и задания**

1. Какие потоки называются асинхронными?
2. Как создать асинхронные потоки?
3. Какое имеют преимущество асинхронные потоки?
4. Какие имеют недостатки асинхронные потоки?
5. Чем Callable отличается от Runnable?
6. Какие методы есть у Future и для чего они нужны?
7. Какие методы есть у ExecutorService и для чего они нужны?

**8. Рекомендуемая литература**


1. **Урванов, Ф. В.** Java. Состояние языка и его перспективы. / Ф.В. Урванов. - Санкт-Петербург : БХВ-Петербург, 2023. - 368 с.
2. **Копец Дэвид.** Классические задачи Computer Science на языке Java. - Санкт-Петербург : Питер, 2022. - 288 с
3. **Васильев А. Н.** Java. Объектно-ориентированное программирование: Учебное пособие / А.Н. Васильев. - Санкт-Петербург : Питер, 2021. - 400 с.
4. **Эванс Бенджамин**. Java для опытных разработчиков. 2-е изд. — (Серия «Библиотека программиста»). - Санкт-Петербург : Питер, 2024. - 736 с.
5. **Лой Марк.** Программируем на Java. 5-е межд. изд. . - Санкт-Петербург : Питер, 2023. - 544 с.
6. **Гетц Брайан**. Java Concurrency на практике. - Санкт-Петербург : Питер, 2021. - 464 с.