Стримы и потоки

Когда в дело вступает тяжелый код

Содержание

1. Стрим данных, try catch finally

В предыдущих лекциях мы подошли к тому, что даже если мы пишем код, в котором дизайнер делает все свои задачи и независимо от него свои задачи делает программист, то все равно в консоли мы видим будто все это происходит последовательно. Как будто один ждет пока другой закончит свою работу. С чем это связано? А с тем, что наш код выполняется синхронно, т.е. последовательно. Ведь он работает в 1 потоке исполнения. Что это такое мы подробно обсудим в третьей части. И чтобы вам доказать это давайте рассмотрим следующую задачу. Но перед этим вспомните вот что – любой код который мы писали до сих пор при выполнении просто завершался и высвечивал Programm finished with exit code 0. Т.е. как только выполнение кода было завершено мы выходили из программы. В этом и суть одного потока.

Теперь, задача у нас следующая – скачать с сети файл, пусть это будет видео файл чтобы мы смогли увидеть его и проверить что он скачалася.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("starting at " + new Date());
        String url = "https://file-examples-com.github.io/uploads/2017/04/file_example_MP4_1920_18MG.mp4";

        try (BufferedInputStream in = new BufferedInputStream(new URL(url).openStream());
             FileOutputStream fileOutputStream = new FileOutputStream( s: "someVideo.mp4")) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer,  i: 0,  i1: 1024)) != -1) {
                fileOutputStream.write(dataBuffer,  i: 0, bytesRead);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            System.out.println("finishing at " +  new Date());
        }
    }
}
```

Первой линией у нас идет логирование времени. Я хочу знать точно сколько секунд ушло на скачивание файла и просто время в начале и в конце. Обратите внимание, это делается элементарно через класс Date. Далее я создаю ссылку на файл по которой будем качать. А дальше начинается новая тема. Помните мы говорили про исключения? Как только что-то идет не по плану – бросить исключение. Вот та же история и со скачиванием файла – если в

процессе скачивания например у меня отвалится интернет – то файл не скачается или скачается битым и я хотел бы об этом узнать. Например чтобы дать юзеру понять что нужно повторить действие. Я понимаю, у нас сейчас нет юая – графического интерфейса, где мы бы нажали на кнопку и получили сообщение об ошибке и нажали еще раз. Но далее мы немного поменяем этот код так, чтобы мы добились скачивания. Итак, чтобы наша программа не закончилась ошибкой, а мы смогли бы попробовать еще раз качать файл, нам нужно ошибку, которая возникнет в ходе работы отловить. Помните мы писали throw new Exception так вот, в рамках этих понятий кто-то бросил ошибку, а кто-то может ее поймать. Как мяч. Да. Так вот, я могу отловить ошибку и что-то сделать. Пока что в коде у нас ничего не происходит кроме того, что я пишу в консоль стек ошибки. e.printStackTrace() где e это сама ошибка, а метод просто выводит в консоль информацию по ней в виде пути – где она началась, какие классы и методы затронула и как дошла до нас. Это и называется трейс. Т.е. возникла ошибка в каком-то классе A, на линии 24 например в методе openConnection() из-за того например что нет интернет соединения. Далее эта ошибка была выброшена в наш класс где мы ее поймали на линии 12 например. Теперь, вся схема выглядит так try – catch т.е. мы пытаемся выполнить некий кусок кода и можем поймать те исключения которые в ней могут быть брошены. Как видите мы ловим IOException это наследник Exception. Если в нашем блоке кода мы обратимся к ссылке в которой null то наше приложение завершится ошибкой. Чтобы такого не было можно ловить любой вид исключения и написать catch(Exception e), но мы уж точно знаем, что во время скачивания файла максимум будет IOException где IO означает input output т.е. ввод вывод. Ведь мы пишем файл на диск, т.е. в наше хранилище. В конкретном случае это хард HDD компьютера. Еще конечно на вашем харде может закночиться место, это тоже ошибка которая может возникнуть.

Ладно, в блоке try мы пишем тот код, который может бросить исключение, а в блоке catch пишем код что делать если ошибка все же возникла. А что за finally ? Это третий блок, который можно написать после первых двух (можно и сразу после try) и он гарантированно будет вызван даже если блок в try бросил исключение а вы не написали блок catch. Мы конечно же ловим исключение, но в любом случае я бы хотел узнать время завершения чтобы посчитать сколько секунд длилось скачивание.

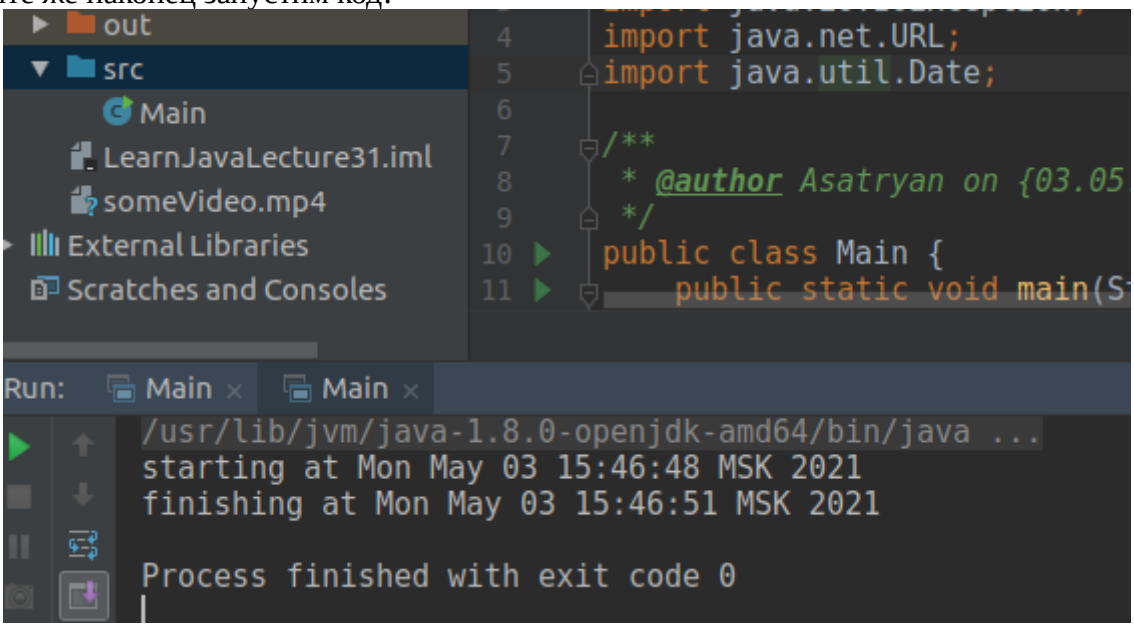
Теперь давайте перейдем к самим стримам. Мы создаем BufferedInputStream что означает буферный стрим (не будем использовать слово поток, оно для другой штуки) ввода и инициализируем его через URL ссылку по которой открываем стрим. Чтобы вы понимали почему мы ловим именно IOException зажмите Ctrl и мышкой кликните на метод openStream() и вы увидете следующее.

```
public final InputStream openStream() throws IOException {
    return this.openConnection().getInputStream();
}
```

Да, открытие стрима может выбросить исключение. Это исходный код класса URL. Точно так же как мы можем посмотреть код наших других классов, так же мы можем посмотреть что в исходниках других классов которые идут вместе с языком java в пакете(папке) net. Вы заметили импорт в начале класса? Java.net.URL ?

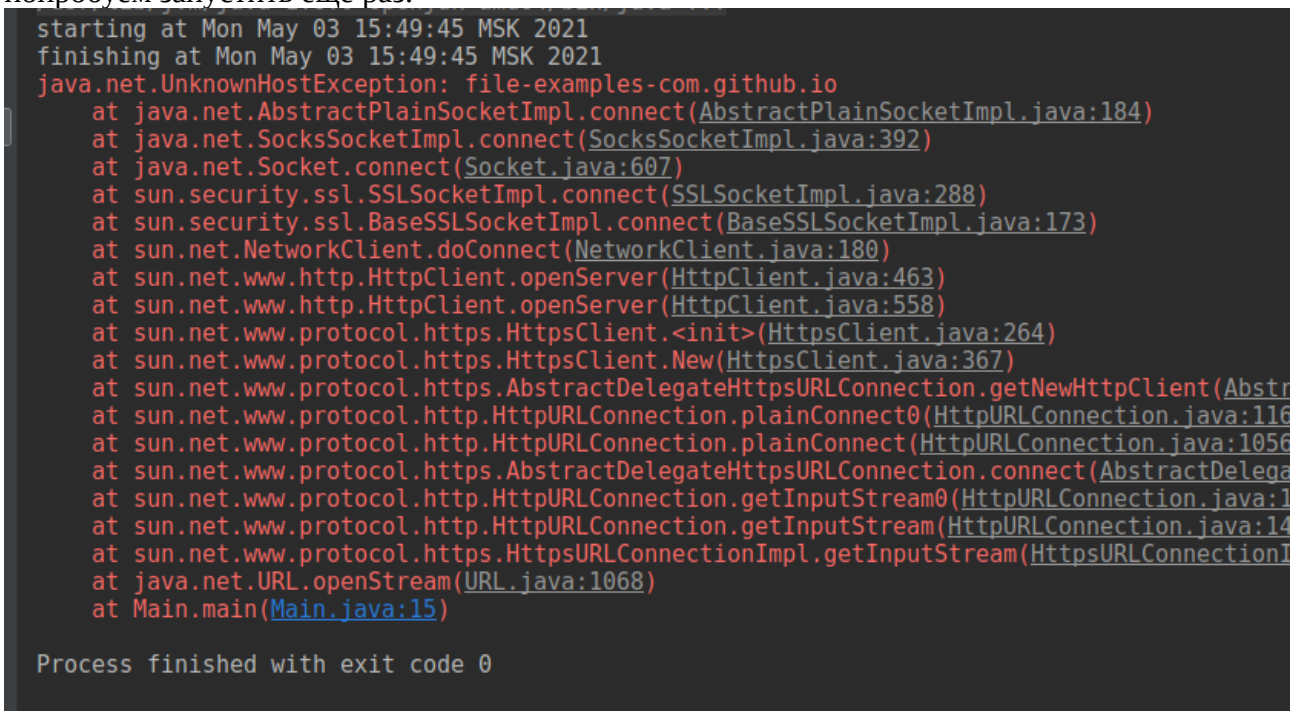
```
import java.io.BufferedInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.net.URL;
import java.util.Date;
```

Мы используем в нашем коде классы из других пакетов языка. Но вернемся к первой линии в блоке try – мы создали буферный стрим ввода из ссылки, т.е. мы сможем скачивать файл по этой ссылке не весь сразу, а с помощью буфера. Чтобы понять что это такое представьте ванную на половину полную водой. Ваша задача перелить содержимое в одно большое ведро. Нет, ведро макнуть в ванную невозможно. И вы берете ковшик или кружку и понемногу переливаете из одного в другое. Это и есть буфер. Помните я говорил что есть очень популярное место где используются как массивы так и тип byte? Это вот буфер. Массив из байтов (да да, если ваш файл весит 18мегабайт, то байт это вот оно самое). Теперь, мы создаем байтовый массив чтобы получить кусками весь видеофайл. Размер нашего буфера 1 мб, т.е. 1024 байт. Далее мы в цикле while получаем кусками весь видеофайл и пишем в файл. Ах да, я забыл упомянуть. Чтобы записать файл нам нужен будет FileOutputStream. т.е. если у нас был стрим ввода, то будет и стрим вывода. Т.е. входит файл который доступен по ссылке, а на выходе будет файл на нашем компьютере. Заметьте, что мы обьявили 2 переменные в самом начале блока try в аргументах как бы. Я обьясню почему так: вообще нужно понимать стримы как ручей или как вода, которая течет по трубам. Вы открываете стрим – вентиль краника и вода (информация) течет по трубам. А когда вы закончите наливать воду в ваше ведро нужно будет закрыть кран. Иначе соседи снизу придут к вам в гости. Самая крутая фишка языка джава что нам не нужно делать это самостоятельно, т.е. вызывать метод close() самостоятельно у обоих стримов. Для этого мы обьявили и передали в блок try эти 2 стрима и они автоматически закроются – поэтому и в джава 8 это называется try with resources. Можно написать просто try { и уже внутри обьявить стримы, но тогда придется закрывать их руками. Поэтому запомните сразу как делать хорошо.
Теперь, у нас стрим ввода и стрим вывода, и мы в ковшик (буфер) кладем по 1 мб данных до тех пор пока наш ввод выдает нам что-либо. Как только мы перекачаем из ввода в вывод все то и выйдем из цикла. in.read → out.write все логично, неправда ли?
Давайте же наконец запустим код!



Как видите у нас на все это ушло 3 секунды. И вуаля, там где у нас лежит код возник файл с именем someVideo.mp4 ведь мы передали имя файла когда создавали стрим вывода. Можете

проверить что там видео. И вы реально можете смотреть в консоль и видеть как сначала стартует программа и как она завершается спустя некоторое время. Чтобы быть уверенным что наш блок try catch работает давайте просто выключим интернет на компьютере и попробуем запустить еще раз.



Как я и обещал блок finally выполнился несмотря на выброс ошибки (вы можете удалить блок catch и проверить). Видите ? Начало программы и завершение одно и то же время. Потому что мы мгновенно перешли от блока try к блоку catch и вывели стек трейс ошибки. Как видите ошибка у нас UnknownHostException это ошибка ввода вывода которая говорит что ей неизвестен хост, т.е. путь. Далее идет цепочка вызовов и кончается нашим мейн классом линией 15. Это вот линия где мы написали try. Т.е мы столкнулись с ошибкой в первой же линии и отловили ее и вывели в консоль ее стектрейс.
И к чему это все? Если помните мы говорили о том, что можем повторить попытку при ошибке. Давайте немного поменяем код и сделаем это.

2. Рекурсия

Давайте выделим метод для скачивания файла, уберем блок finally чтобы не логировать время каждый раз при ошибке. Теперь мы логируем старт, вызываем метод скачивания и логируем время завершения. А что у нас в самом методе скачивания. Все то же что и раньше, кроме того, что мы вместо логирования стек ошибки создаем сканер и просим юзера ответить на вопрос – повторить попытку или нет? Если вы заметили мы читаем 1 строку от юзера и если он ввел “y” неважно в каком регистре – equalsIgnoreCase делает именно это – сравнивает строки вне зависимости от того в верхнем они регистре или нижнем. Можно ввести y или Y. И если юзер хочет повторить попытку то вызываем этот же метод еще раз.
Чаво? Это как так? А вот так. Заметьте что рядом с номером линии возник значок, если навести мышку то там будет описание – Recursive call это значит рекурсивный вызов. Помните циклы? Их можно заменить на рекурсивный вызов функции. т.е. мы при тех или иных условиях вызываем себя еще раз до тех пор пока не удовлетворено условие.

Чтобы протестировать это все можете выключить интернет и запустить код, ввести “y” несколько раз подряд и потом уже включить интернет и скачать наконец-то этот файл.

```
public class Main {
    private static final String URL = "https://file-examples-com.github.io/uploads/2017/04/fi

    public static void main(String[] args) {
        System.out.println("starting at " + new Date());
        downloadFile();
        System.out.println("finishing at " + new Date());
    }

    private static void downloadFile() {
        System.out.println("starting downloading file");
        try (BufferedInputStream in = new BufferedInputStream(new URL(URL).openStream());
             FileOutputStream fileOutputStream = new FileOutputStream( s: "someVideo.mp4")) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer, i: 0, i1: 1024)) != -1) {
                fileOutputStream.write(dataBuffer, i: 0, bytesRead);
            }
        } catch (IOException e) {
            System.out.println("something went wrong, try again? y/n");
            Scanner scanner = new Scanner(System.in);
            String answer = scanner.nextLine();
            if (answer.equalsIgnoreCase( s: "y")) {
                downloadFile();
            }
        }
    }
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/j
starting at Mon May 03 15:56:49 MSK 2021
starting downloading file
something went wrong, try again? y/n
y
starting downloading file
finishing at Mon May 03 15:57:05 MSK 2021

Process finished with exit code 0
```

Но конечно же с рекурсивной функцией могут быть опасные ситуации, если вы написали ее так, что она будет бесконечно вызывать себя. Тогда ваша программа вылетит с ошибкой. Это то же самое что и сделать while(true) без break внутри или без выброса исключения. Так что всегда будьте осторожны с этим.
Давайте я покажу вам лучшее использование рекурсивной функции. Для этого напишем класс скачивания файла. И у него будет метод скачивания с количеством попыток.

```
public class Main {
    private static final String URL = "https://file-examples-com.github.io/u

    public static void main(String[] args) {
        System.out.println("starting at " + new Date());
        new DownloadFile(URL,  fileName: "video.mp4").start( attempts: 2);
        System.out.println("finishing at " + new Date());
    }
}
```

```java
public class DownloadFile {

    private final String url;
    private final String fileName;

    public DownloadFile(String url, String fileName) {
        this.url = url;
        this.fileName = fileName;
    }

    public void start(int attempts) {
        System.out.println("starting downloading file");
        try (BufferedInputStream in = new BufferedInputStream(new URL(url).openStream());
             FileOutputStream fileOutputStream = new FileOutputStream(fileName)) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer,  i: 0,  i1: 1024)) != -1) {
                fileOutputStream.write(dataBuffer,  i: 0, bytesRead);
            }
        } catch (IOException e) {
            if (attempts > 1)
                start(--attempts);
            else
                System.out.println("downloading file failed");
        }
    }
}
```

```
starting at Mon May 03 16:34:55 MSK 2021
starting downloading file
starting downloading file
downloading file failed
finishing at Mon May 03 16:34:55 MSK 2021

Process finished with exit code 0
```

Мы вынесли код скачивания в файл и пусть он работает с любыми ссылками и именами файлов. И чтобы сделать гибким скачивание пусть мы принимаем аргументом количество попыток. Мы передали количество попыток 2. Когда поймали ошибку то вызвали этот же метод но уже с количеством попыток на 1 меньше. Если заметили у нас преинкремент. Т.е. мы сначала уменьшили число попыток на 1 и стало 1 и потом уже вызвали этот же метод. Чтобы сделать более гибким можно дать колбек наружу когда не удалось загрузить и тогда можно написать метод принятия решения от юзера в консоли как было раньше. Но это неважно. Самое лучшее использование рекурсивной функции когда мы уменьшаем аргумент который он проверяет внутри себя. Можете попробовать с помощью рекурсии переписать старый код где у вас были циклы.

Но мы все же вернемся к потокам.

3. Потоки исполнения

Итак, мы убедились в том, что у нас программа имеет 1 поток выполнения. Чтобы понять что это такое подумайте о беговой дорожке. Вы бежите по одной дорожке, но когда вы например пытаетесь загрузить файл то вы будто останавливаетесь и ждете ее окончания.

Если вы бежите на одной дорожке, то когда на вашем пути возникнут преграды вы должны будете их устранить перед тем как бежать дальше. А теперь представьте, что у вас не 1 дорожка для бега, а несколько. И если на вашей дорожке возникла преграда, то вы можете легко перейти на другую дорожку. Это не совсем подходит под описание потоков, но более менее думаю понятно и применимо. В нашем конкретном случае у нас вся программа делает 1 вещь – качает файл. Но ведь в реальности мало таких случаев. Мы всегда делаем несколько вещей одновременно. Вот например вы слушаете музыку в приложении, но одновременно с этим вы листаете ленту новостей. Если бы все наши программы исполнялись в 1 потоке, то вы бы не могли делать ничего больше пока не послушаете музыку. Благо у нас в 21 веке появились многоядерные процессоры и мы можем выполнять код на разных ядрах. Прямо сейчас я слушаю музыку в вебе и пишу этот текст. Если бы мой компьютер был одноядерным с малым количеством ОЗУ то я бы или слушал музыку или же писал текст, одно из двух. Или же мой текстовый редактор зависал каждый раз как я писал бы строчку. И одновременно с этим лагала бы музыка. Ведь все ресурсы компьютера тогда бы разделились между 2 задачам и попеременно выполняли бы то или иное действие.

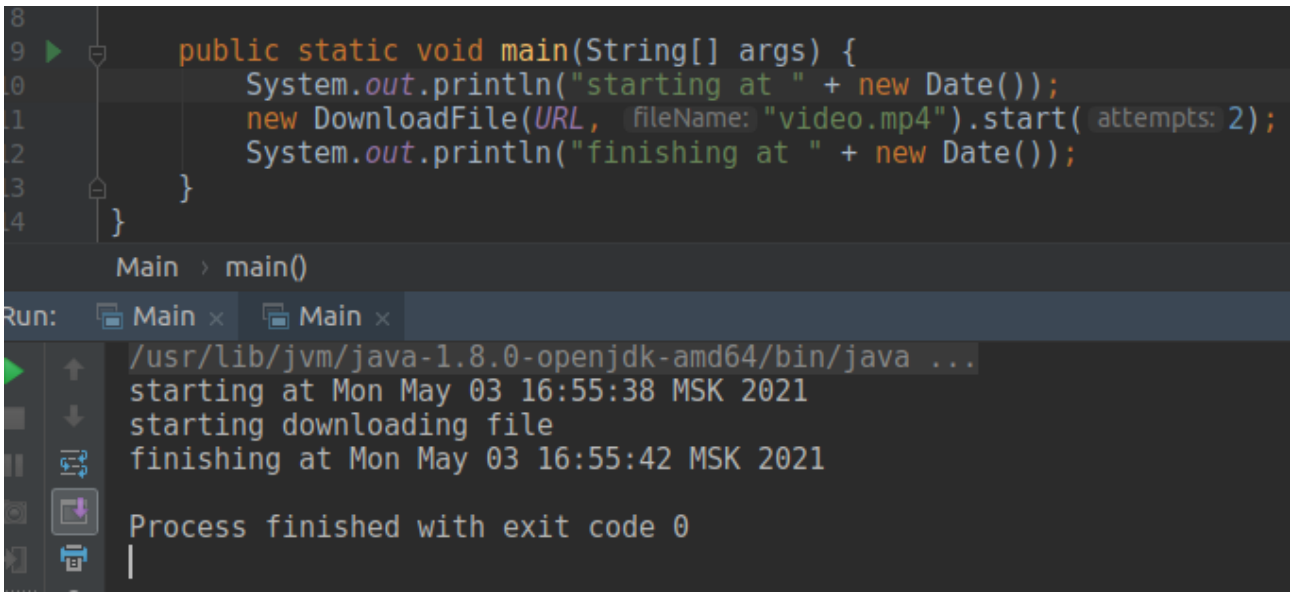
Представьте человека, который не может делать одновременно более 1 действия. Это вот однопоточный процесс. Вот он ест и пока он ест он не может делать больше ничего. На самом деле люди могут делать одновременно несколько действий – как Цезарь 3. Можно писать один текст, говорить другой и слушать третий. Все потому что у нас 2 полушария мозга. Шучу. На самом деле многие люди однопоточны и они переключаются между задачами. Вряд ли вы можете поддержать разговор с одним человеком и в это же время переписываться с другим. Вы скорей всего переспросите человека что он говорил или попросите его подождать пока вы отвечаете на вопрос другого в чате.

Тем не менее компьютеры стали действительно многоядерными и на каждом ядре можно выполнить по задаче. Так почему бы нам не использовать эту возможность? В нашем конкретном случае представим что вы в чате с ботом и хотите скачать музыку параллельно. Мы начнем скачивать музыку и начнем чат с ботом одновременно и пусть когда файл скачается мы просто об этом узнаем между делом. Согласитесь, жизнь была бы трудной если бы вы ставили на скачивание файл и не могли бы пользоваться программой. Кстати, в начале века так и было. Мы ставили на скачивание файл и ни в каких сайтах не серфили больше чтобы единственный поток был занят этой задачей и трафик не делился. А тогда скорость интернета была действительно слабой. Ты мог бы скачивать 1 картинку более минуты. Но если ты одновременно еще и листал почту, то картинка скачивалась уже не 1 минуты, а все 5.

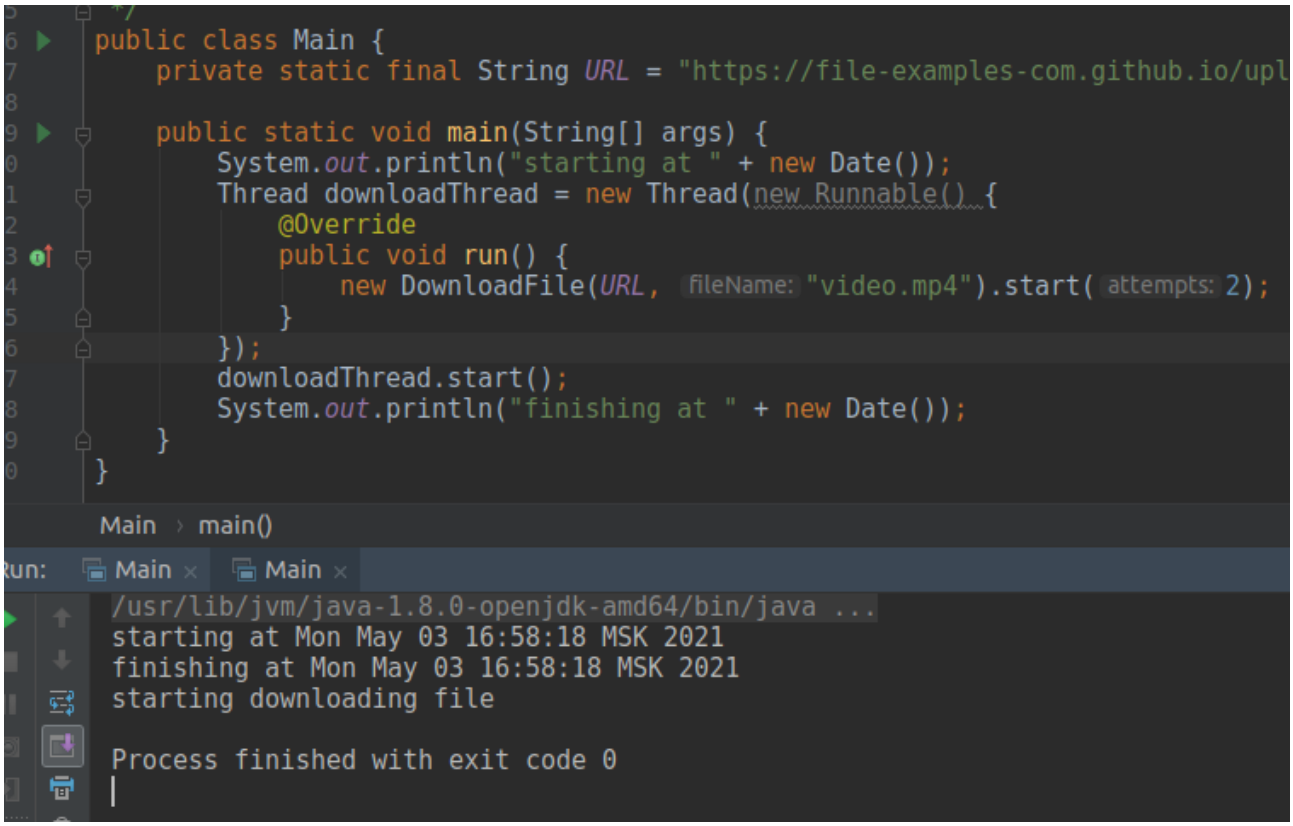
Ладно, перейдем же к коду. Как мы можем выполнить наш код быстрее? Для этого в джава есть класс Thread и интерфейс Runnable. Давайте посмотрим на это.

Сейчас у нас так – мы логируем время старта и логируем время окончания и качаем файл между этими 2 событиями и потому время отличается на несколько секунд.

Теперь давайте уже создадим новый поток в котором отдельно скачаем файл и освободим наш главный поток (в котором мейн метод находится).



Мы создаем новый поток и в него передаем то, что должно выполняться. Как видите Runnable это интерфейс с 1 методом, значит можно будет поменять на лямбду. Далее мы запускаем поток с помощью метода start. Теперь посмотрите в консоль. Время старта и время финиша – одно и то же. Потому что мы перешли с линии где логируем стартовое время на линию старта потока и сразу же дальше. А файл начал скачиваться уже после. Потому что джава немного времени потратила на создание и запуск потока.



Да, создать поток это дорогое удовольствие. Вы можете залогировать время начала скачивания и увидете сколько уходит секунд на создание потока и старт скачивания. Здесь у нас программа опять же закончилась, но только потому что у нас в мейн методе больше ничего нет. Мы могли бы написать бесконечный цикл с вводом каких-нибудь данных в

консоли или же тот же чатбот из предыдущих задач. Но суть в том, что одно не мешало бы другому.

Теперь же когда у нас есть новый поток для загрузки мы можем скачивать хоть тысячу файлов по очереди или же паралелльно пока у нас есть свободные ядра и ОЗУ.

Об этом всем подробнее позже. А пока запомните это – если вам нужно сделать сразу несколько вещей то создайте новый поток и не ждите окончания одного десйтвия чтобы запустить новую задачу. Если вернуться к аналогии с беговой дорожке – представьте что вы можете клонировать себя и ваш клон будет бежать рядом с вами некоторое время, но по ощущениям будто эту дистанцию пробежали вы. Т.е. за меньшее время вы пробежите больше расстояния. Применимо к программированию – за меньшее время вы сделаете больше задач если распараллелите их по потокам. В одном потоке у вас может играть музыка, в другом потоке у вас будет скачиваться видео-файл, в третьем и основном потоке у вас будет лента новостей, а в четвертом обновление приложений с гугл плей маркета. Поверьте, мобилки из 2010 года не справились бы со всем этим, им бы не хватило ядер.

Закрепите знания – попробуйте скачать 4 файла одинаковой длины за кратчайшее врмея. Для этого можете использовать тот же URL но дать разные имена чтобы ваша файловая система не конфликтовала. А в главном потоке можете запустить секундомер через тот же Timer например. По завершению скачивания отлогируйте в консоль время.

Попробуйте увеличить количество потоков до максимума пока время остается тем же самым. Проверьте экспериментально на сколько потоков тянет ваш компьютер. Можете в список положить потоки и выполнить их сразу все в цикле. В итоге увидете что лучше – создать 10 потоков и в каждом скачать по 1 файлу или же условно 5 потоков в каждом качать по 2 файла. Так как потоки дорого (долго и тяжело) создавать, то можно переиспользовать уже готовый поток, если он не завершился конечно же. А любой поток если он начался то будет звершен когда в его главном методе не будет больше кода. То что мы пишем в run().

Ну и классическая задача на рекурсию – написать метод вывода чисел Фибоначи.

Но вот вам еще одна. Пусть суммируются все числа от 0 до введенного через рекурсию.

Опытным путем установить когда стек переполнится (отдавать в цикле все числа в метод и посмотреть когда выкинет ошибку)

```
for (int i=0;i<Integer.MAX_VALUE;i++) {
recursiveMethod(i);

}
```

Напишите еще один метод который может бросить исключение 2 разных типов и попробуйте обьединить их в catch блоке там где вызвали через “|” - унарный оператор или.

И последнее задание – положите в проект свой файл и попробуйте его скопировать в коде. Т.е. если раньше мы открывали стрим ввода из ссылки, то теперь будет стрим из файла и в другой файл. И попробуйте использовать простой try { и закрыть стрим руками в конце.