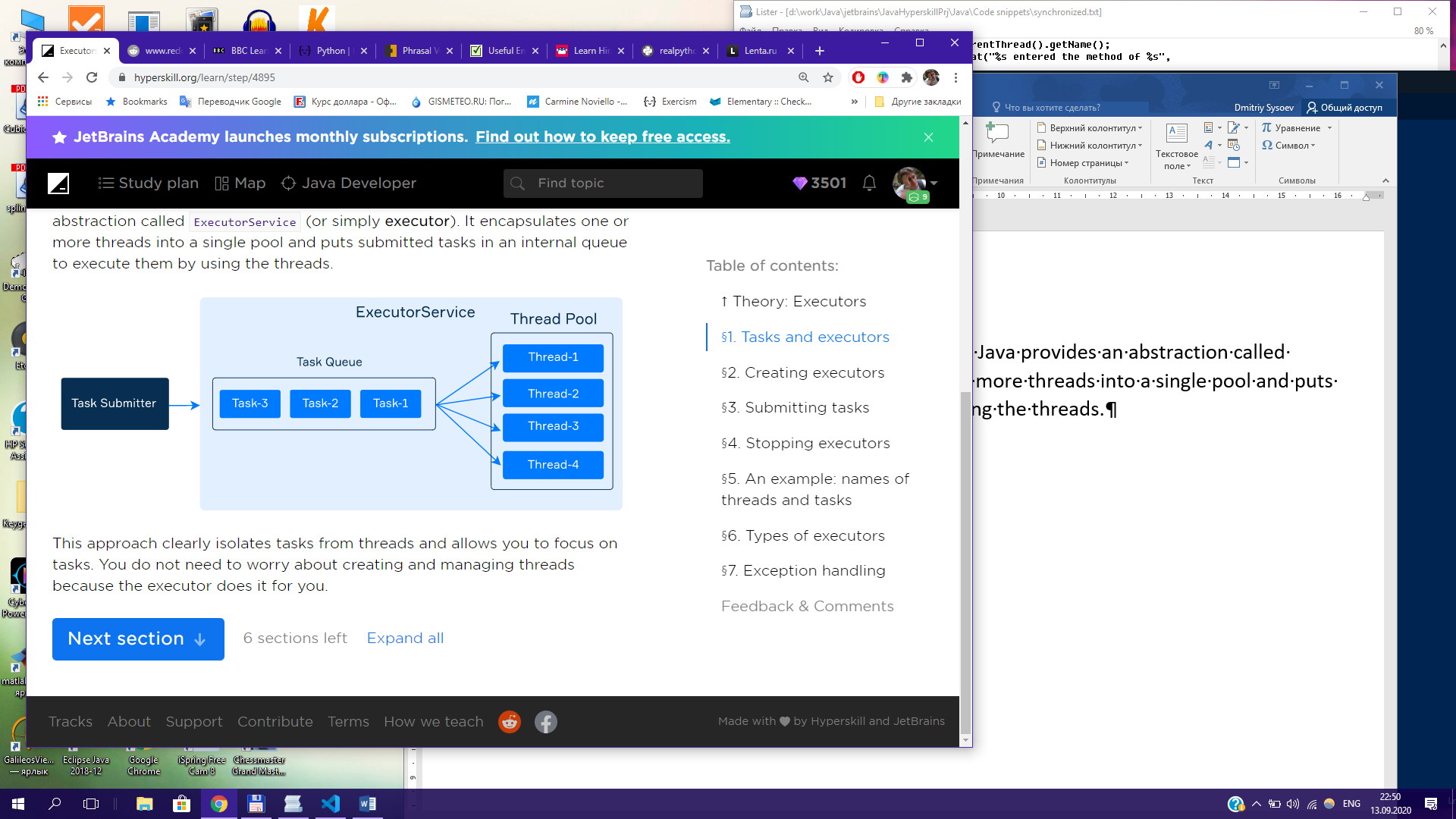
# Task and executors

To simplify the development of multi-threaded applications, Java provides an abstraction called ExecutorService (or simply executor). It encapsulates one or more threads into a single pool and puts submitted tasks in an internal queue to execute them by using the threads.



This approach clearly isolates tasks from threads and allows you to focus on tasks. You do not need to worry about creating and managing threads because the executor does it for you.

All types of executors are located in the **java.util.concurrent** package. This package also contains a convenient utility class Executors for creating different types of ExecutorServices’s.

ExecutorService executor = Executors.newFixedThreadPool(4);

An executor has the submit method that accepts a Runnable task to be executed. Since Runnable is a functional interface, it is possible to use a lambda expression as a task.

As an example, here we submit a task that prints “Hello!” to the standard output.

Executor.submit( () -> System.out.println(“Hello!”));

After invoking submit, the current thread does not wait for the task to complete. It just adds the task to the executor’s internal queue to be executed asynchronously by one of the threads.

# There are two methods for stopping executors:

void shutdown() – waits until all running task completes and prohibits submitting of new tasks;

List<Runnable> shutdownNow() immediately stops all running tasks and returns a list of the tasks that were awaiting execution.

Note that shutdown() does not block the current thread unlike join() of Thread. If you need to wait until the execution is complete, you can invoke awaiteTermination(…) with the specified waiting time.

awaitTermination(…)

ExecutorService executor = Executor.newFixedThreadPool(4);

Executor.shutdown();

boolean terminated = executor.awaitTemination(60, TimeUnit.MILLISECONDS);

if (terminated) {

System.out.println(“The executor was successfully stoped”);

} else {

System.out.println(“Timeout elapsed before termination”);

}

In the following example, we create one executor with a pool consisting of four threads. We submit ten tasks to it and then analyze the results. Each task prints the name of a thread that executes it, as well as the name of the task.

Import java.util.concurrent.ExecutorService;

Import java.util.concurrent.Executor;

public class ExecutorDemo {

private final static int POOL\_SIZE = 4;

private final static int NUMBER\_OF\_TASKS = 10;

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(POOL\_SIZE);

for (int I = 0; i < NUMBER\_OF\_TASK; i++) {

int taskNumber = i;

executor.submit(() -> {

String taskName = “task” + taskNumber;

String threadName= Thread.currentThread().getName();

System.out.println(“%s executes %s\n”, threadName, taskName);

});

}

Executor.shutdown();

}

}

Output:

pool-1-thread-1 executes task-0  
pool-1-thread-2 executes task-1  
pool-1-thread-4 executes task-3  
pool-1-thread-3 executes task-2  
pool-1-thread-3 executes task-7  
pool-1-thread-3 executes task-8  
pool-1-thread-3 executes task-9  
pool-1-thread-1 executes task-6  
pool-1-thread-4 executes task-5  
pool-1-thread-2 executes task-4

if you do not know how many threads are needed in your pool, you can take the number of available processors as the pool size.

int poolSize = Runtime.getRuntime().availableProcessors();  
ExecutorService executor = Executors.newFixedThreadPool(poolSize);

# Types of Executors

* An executor with a single thread

The simplest executor has only a single thread in the pool. It may be enough for async execution of rare submitted and small tasks.

ExecutorService executor = Executors.newSingleThreadExecutor();

* An executor with growing pool

There is also an executor that automatically increases the number of threads as it needed and reuse previously constructed threads.

ExecutorService executor = Executors.newCachedThreadPool();

It can typically improve the performance of programs that perform many short-lived asynchronous tasks. But it can also lead to problems when the number of threads increases too much. It is preferable to choose the fixed thread-pool executor whenever possible.

* An executor that schedules a task

If you need to perform the same task periodically or only once after the given delay, use the following executor:

ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

The method scheduleAtFixedRate submits a periodic Runnable task that becomes enabled first after the given initDelay, and subsequently with the given period.

Here is a quick example with scheduling:

Here is a fragment of the output:

02:30:06.375392: Hello!  
02:30:07.375356: Hello!  
02:30:08.375376: Hello!  
...and even more...

# Exception handling

What do you think the following code will print?

ExecutorService executor = Executors.newSingleThreadExecutor();

Executor.submit(() -> System.out.println(2 / 0));

It does not print anything at all, including the exception! This is why it is common practice to wrap a task in the try-catch block not to lose the exception.