Java Enterprise Edition

ANUDIP FOUNDATION

Concurrency patterns in JAVA

Objective: • Pattern • Queue • Callable futures , handling Exceptions	**Materials Required:** 1. Eclipse IDE/IntelliJ/STC
Theory: 120 mins	**Practical :60 mins**
Total Duration: 180 mins	

· Concurrent Patterns in Java
o Introducing Executors, What Is Wrong with the Runnable Pattern?
o Defining the Executor Pattern: A New Pattern to Launch Threads
o Defining the Executor Service Pattern, a First Simple Example
o Comparing the Runnable and the Executor Service Patterns
o Understanding the Waiting Queue of the Executor Service
o Wrapping-up the Executor Service Pattern
o From Runnable to Callable: What Is Wrong with Runnables?
o Defining a New Model for Tasks That Return Objects
o Introducing the Callable Interface to Model Tasks
o Introducing the Future Object to Transmit Objects Between Threads
o Wrapping-up Callables and Futures, Handling Exceptions

Concurrent patterns in java

Java concurrency (multi-threading). This article describes how to do concurrent programming with Java. It covers the concepts of parallel programming, immutability, threads, the executor framework (thread pools), futures, callables CompletableFuture and the fork-join framework.

What is concurrency?

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Process vs. threads

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A *thread* is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.

A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Improvements and issues with concurrency

Limits of concurrency gains

Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior. Concurrency promises to perform certain task faster as these tasks can be divided into subtasks and these subtasks can be executed in parallel. Of course the runtime is limited by parts of the task which can be performed in parallel.

The theoretical possible performance gain can be calculated by the following rule which is referred to as *Amdahl's Law*.

If F is the percentage of the program which can not run in parallel and N is the number of processes, then the maximum performance gain is 1 / (F+ ((1-F)/N)).

Concurrency issues

Threads have their own call stack, but can also access shared data. Therefore you have two basic problems, visibility and access problems.

A visibility problem occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.

An access problem can occur if several threads access and change the same shared data at the same time.

Visibility and access problem can lead to:

- Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.
- Safety failure: The program creates incorrect data.

Concurrency in Java

Processes and Threads

A Java program runs in its own process and by default in one thread. Java supports threads as part of the Java language via the `Thread` code. The Java application can create new threads via this class.

Java 1.5 also provides improved support for concurrency with the `java.util.concurrent` package.

Locks and thread synchronization

Java provides *locks* to protect certain parts of the code to be executed by several threads at the same time. The simplest way of locking a certain method or Java class is to define the method or class with the `synchronized` keyword.

The *synchronized* keyword in Java ensures:

- that only a single thread can execute a block of code at the same time

- that each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock

Synchronization is necessary for mutually exclusive access to blocks of and for reliable communication between threads.

You can use the *synchronized* keyword for the definition of a method. This would ensure that only one thread can enter this method at the same time. Another thread which is calling this method would wait until the first thread leaves this method.

```
public synchronized void critial() {
   // some thread critical stuff
   // here
}
```

You can also use the synchronized keyword to protect blocks of code within a method. This block is guarded by a key, which can be either a string or an object. This key is called the *lock*.

All code which is protected by the same lock can only be executed by one thread at the same time.

For example the following data structure will ensure that only one thread can access the inner block of the add() and next() methods.

```
package de.vogella.pagerank.crawler;

import java.util.ArrayList;
import java.util.List;

/**
 * Data structure for a web crawler. Keeps track of the visited sites and keeps
 * a list of sites which needs still to be crawled.
 *
 * @author Lars Vogel
 *
 */
public class CrawledSites {
   private List<String> crawledSites = new ArrayList<String>();
   private List<String> linkedSites = new ArrayList<String>();

   public void add(String site) {
      synchronized (this) {
         if (!crawledSites.contains(site)) {
            linkedSites.add(site);
         }
      }
   }

   /**
    * Get next site to crawl. Can return null (if nothing to crawl)
    */
```

```
    public String next() {
        if (linkedSites.size() == 0) {
            return null;
        }
        synchronized (this) {
            // Need to check again if size has changed
            if (linkedSites.size() > 0) {
                String s = linkedSites.get(0);
                linkedSites.remove(0);
                crawledSites.add(s);
                return s;
            }
            return null;
        }
    }

}
```

Volatile

If a variable is declared with the *volatile* keyword then it is guaranteed that any thread that reads the field will see the most recently written value. The *volatile* keyword will not perform any mutual exclusive lock on the variable.

As of Java 5, write access to a *volatile* variable will also update non-volatile variables which were modified by the same thread. This can also be used to update values within a reference variable, e.g. for a *volatile* variable person. In this case you must use a temporary variable person and use the setter to initialize the variable and then assign the temporary variable to the final variable. This will then make the address changes of this variable and the values visible to other threads.

The Executor Framework

Introduction

With the increase in the number of the cores available in the processors nowadays, coupled with the ever increasing need to achieve more throughput, multi-threading APIs are getting quite popular. Java provides its own multi-threading framework called the Executor Framework.

What is the Executor Framework?

The Executor Framework contains a bunch of components that are used to efficiently manage worker threads. The Executor API de-couples the execution of task from the actual task to be executed via Executors. This design is one of the implementations of the Producer-Consumer pattern.

Introduction

With the increase in the number of the cores available in the processors nowadays, coupled with the ever increasing need to achieve more throughput, multi-threading APIs are getting quite popular. Java provides its own multi-threading framework called the Executor Framework.

What is the Executor Framework?

The Executor Framework contains a bunch of components that are used to efficiently manage worker threads. The Executor API de-couples the execution of task from the actual task to be executed via Executors. This design is one of the implementations of the Producer-Consumer pattern.

The java.util.concurrent.Executors provide factory methods which are be used to create ThreadPools of worker threads.

To use the Executor Framework we need to create one such thread pool and submit the task to it for execution. It is the job of the Executor Framework to schedule and execute the submitted tasks and return the results from the thread pool.

A basic question that comes to mind is why do we need such thread pools when we can create objects of java.lang.Thread or implement Runnable/Callable interfaces to achieve parallelism?

Introduction

With the increase in the number of the cores available in the processors nowadays, coupled with the ever increasing need to achieve more throughput, multi-threading APIs are getting quite popular. Java provides its own multi-threading framework called the Executor Framework.

What is the Executor Framework?

The Executor Framework contains a bunch of components that are used to efficiently manage worker threads. The Executor API de-couples the execution of task from the actual task to be executed via Executors. This design is one of the implementations of the Producer-Consumer pattern.

The java.util.concurrent.Executors provide factory methods which are be used to create ThreadPools of worker threads.

To use the Executor Framework we need to create one such thread pool and submit the task to it for execution. It is the job of the Executor Framework to schedule and execute the submitted tasks and return the results from the thread pool.

A basic question that comes to mind is why do we need such thread pools when we can create objects of java.lang.Thread or implement Runnable/Callable interfaces to achieve parallelism?

The answer comes down to two basic facts:

1. Creating a new thread for a new task leads to overhead of thread creation and tear-down. Managing this thread life-cycle significantly adds to the execution time.
2. Adding a new thread for each process without any throttling leads to the creation of a large number of threads. These threads occupy memory and cause wastage of resources. The CPU starts to spend too much time switching contexts when each thread is swapped out and another thread comes in for execution.

All these factors reduce the throughput of the system. Thread pools overcome this issue by keeping the threads alive and reusing the threads. Any excess tasks flowing in than the threads in the pool can handle are held in a Queue. Once any of threads get free, they pick up the next task from this queue. This task queue is essentially unbounded for the out-of-box executors provided by the JDK.

Types of Executors

Now that we have a good idea of what an executor is, let's also take a look at the different kinds of executors.

SingleThreadExecutor

This thread pool executor has only a single thread. It is used to execute tasks in a sequential manner. If the thread dies due to an exception while executing a task, a new thread is created to replace the old thread and the subsequent tasks are executed in the new one.

```
ExecutorService executorService = Executors.newSingleThreadExecutor()
```

FixedThreadPool(n)

As the name indicates, it is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads and if there is more task they are stored on a LinkedBlockingQueue. This number is usually the total number of the threads supported by the underlying processor.

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

CachedThreadPool

This thread pool is mostly used where there are lots of short-lived parallel tasks to be executed. Unlike the fixed thread pool, the number of threads of this executor pool is not bounded. If all the threads are busy executing some tasks and a new task comes, the pool will create and add a new thread to the executor. As soon as one of the threads becomes free, it will take up the execution of the new tasks. If a thread remains idle for sixty seconds, they are terminated and removed from cache.

However, if not managed correctly, or the tasks are not short-lived, the thread pool will have lots of live threads. This may lead to resource thrashing and hence performance drop.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

ScheduledExecutor

This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a

```
ScheduledExecutorService scheduledExecService = Executors.newScheduledThreadPool(1);
```

The tasks can be scheduled in ScheduledExecutor using either of the two scheduleAtFixedRate scheduleWithFixedDelay

```
scheduledExecService.scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit un
it)
```

```
scheduledExecService.scheduleWithFixedDelay(Runnable command, long initialDelay, long period, TimeUni
t unit)
```

The main difference between the two methods is their interpretation of the delay between consecutive executions of a scheduled job.

scheduleAtFixedRate executes the task with fixed interval, irrespective of when the previous task ended.

scheduleWithFixedDelay will start the delay countdown only after the currently task completes.

Understanding the Future Object

The result of the task submitted for execution to an executor can be accessed using the java.util.concurrent.Future object returned by the executor. Future can be thought of as a promise made to the caller by the executor.

```
Future<String> result = executorService.submit(callableTask);
```

A task submitted to the executor, like above, is asynchronous i.e. the program execution does not wait for the completion of task execution to proceed to next step. Instead, whenever the task execution is completed, it is set in this Future object by the executor.

The caller can continue executing the main program and when the result of the submitted task is needed he can call .get() on this Future object. If the task is complete the result is immediately returned to the

caller or else the caller is blocked until the execution of this is completed by the executor and the result is computed.

If the caller cannot afford to wait indefinitely before retrieving the result, this wait can be timed as well. This is achieved by the Future.get(long timeout, TimeUnit unit) method which throws a TimeoutException if the result is not returned in the stipulated timeframe. The caller can handle this exception and continue with the further execution of the program.

If there is an exception when executing the task, the call to get method will throw an ExecutionException.

An important thing with respect to result being returned by Future.get() method is that it is returned only if the submitted task implements java.util.concurrent.Callable. If the task implements the Runnable interface, the call to .get() will return null once the task is complete.

Another important method is the Future.cancel(boolean mayInterruptIfRunning) method. This method is used to cancel the execution of a submitted task. If the task is already executing, the executor will attempt to interrupt the task execution if the mayInterruptIfRunning flag is passed as true.

Example: Creating and Executing a Simple Executor

We will now create a task and try to execute it in a fixed pool executor:

```
public class Task implements Callable<String> {
    private String message;
    public Task(String message) {
        this.message = message;
    }
    @Override
    public String call() throws Exception {
        return "Hello " + message + "!";
    }
}
```

The Task class implements Callable and is parameterized to String type. It is also declared to throw Exception. This ability to throw an exception to the executor and executor returning this exception back to the caller is of great importance because it helps the caller know the status of task execution.

```
public class ExecutorExample {
    public static void main(String[] args) {
        Task task = new Task("World");

        ExecutorService executorService = Executors.newFixedThreadPool(4);
        Future<String> result = executorService.submit(task);
        try {
```

```
            System.out.println(result.get());
        } catch (InterruptedException | ExecutionException e) {
            System.out.println("Error occured while executing the submitted task");
            e.printStackTrace();
        }
        executorService.shutdown();
    }
}
```

Here we have created a FixedThreadPool executor with a count of 4 threads since this demo is developed on a quad-core processor. The thread count can be more than the processor cores if the tasks being executed perform considerable I/O operations or spend time waiting for external resources.

We have instantiated the Task class and are passing it to the executor for execution. The result is returned by the Future object, which we then print on the screen.

```
Hello World!
```

As expected, the task appends the greeting "Hello" and return the result via the Future object.

Lastly, we call the shutdown on the executorService object to terminate all the threads and return the resources back to the OS.

The .shutdown() method waits for the completion of currently submitted tasks to the executor. However, if the requirement is to immediately shut down the executor without waiting then we can use the .shutdownNow() method instead.

Any tasks pending for execution will be returned back in a java.util.List object.

We can also create this same task by implementing the Runnable interface:

```
public class Task implements Runnable{

    private String message;

    public Task(String message) {

        this.message = message;

    }

    public void run() {
```

```
        System.out.println("Hello " + message + "!");

    }

}
```

There a couple of important changes here when we implement runnable.

1. The result of task execution cannot be returned from the run() method. Hence, we are printing directly from here.
2. The run() method is not configured to throw any checked exceptions.

Java executor framework (java.util.concurrent.Executor), released with the JDK 5 is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads.

Creating a thread in java is a very expensive process which includes memory overhead also. So, it's a good idea if we can re-use these threads once created, to run our future runnables. In this **executor framework tutorial**, I will write some demo programs demonstrating the usage of Executor and then we will talk about some best practices which you need to keep in mind while designing your next multi-threaded application.

1. Java executor framework example

In our demo application, we have two tasks running. Neither is expected to terminate, and both should run for the duration of the application's life. I will try to write a main wrapper class such that:

- If any task throws an exception, the application will catch it and restart the task.
- If any task runs to completion, the application will notice and restart the task.

Below is the code sample of above required application.

DemoExecutorUsage.java

```
package com.howtodoinjava.multiThreading.executors;


import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;
```

```
import java.util.concurrent.Future;

public class DemoExecutorUsage {

    private static ExecutorService executor = null;
    private static volatile Future taskOneResults = null;
    private static volatile Future taskTwoResults = null;

    public static void main(String[] args) {
        executor = Executors.newFixedThreadPool(2);
        while (true)
        {
            try
            {
                checkTasks();
                Thread.sleep(1000);
            } catch (Exception e) {
                System.err.println("Caught exception: " + e.getMessage());
            }
        }
    }

    private static void checkTasks() throws Exception {
        if (taskOneResults == null
                || taskOneResults.isDone()
                || taskOneResults.isCancelled())
        {
            taskOneResults = executor.submit(new TestOne());
        }

        if (taskTwoResults == null
```

```
            || taskTwoResults.isDone()
            || taskTwoResults.isCancelled())
        {
            taskTwoResults = executor.submit(new TestTwo());
        }
    }
}

class TestOne implements Runnable {
    public void run() {
        while (true)
        {
            System.out.println("Executing task one");
            try
            {
                Thread.sleep(1000);
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }

    }
}

class TestTwo implements Runnable {
    public void run() {
        while (true)
        {
            System.out.println("Executing task two");
            try
            {
```

```
                Thread.sleep(1000);
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }
}
```

Please do not forget to read best practices at the end of post.

2. Java executor framework – multi runnable

It's not necessary that each Runnable should be executed in a separate thread. Sometimes, we need to do multiple jobs in a single thread and each job is instance of Runnable. To design this type of solution, a **multi runnable** should be used. This multi runnable is nothing but a collection of runnables which needs to be executed. Only addition is that this multi runnable is also a **Runnable** itself.

Below is the list of tasks which needs to be executed in a single thread.

TaskOne TaskTwo TaskThree

```
package com.howtodoinjava.multiThreading.executors;


public class TaskOne implements Runnable {
    @Override
    public void run() {
        System.out.println("Executing Task One");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}


public class TaskTwo implements Runnable {
```

```
    @Override
    public void run() {
        System.out.println("Executing Task Two");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class TaskThree implements Runnable {
    @Override
    public void run() {
        System.out.println("Executing Task Three");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Lets create out multi runnable wrapper of above Tasks.

MultiRunnable.java

```
package com.howtodoinjava.demo.multiThread;

import java.util.List;

public class MultiRunnable implements Runnable {
```

```
    private final List<Runnable> runnables;

    public MultiRunnable(List<Runnable> runnables) {
        this.runnables = runnables;
    }

    @Override
    public void run() {
        for (Runnable runnable : runnables) {
            new Thread(runnable).start();
        }
    }
}
```

Now above multi runnable can be executed this way as in below program:

MultiTaskExecutor.java

```
package com.howtodoinjava.demo.multiThread;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MultiTaskExecutor {

    public static void main(String[] args) {
```

```
        BlockingQueue<Runnable> worksQueue = new ArrayBlockingQueue<Runnable>(10);
        RejectedExecutionHandler rejectionHandler = new RejectedExecutionHandelerImpl();
        ThreadPoolExecutor executor = new ThreadPoolExecutor(3, 3, 10, TimeUnit.SECONDS,
worksQueue, rejectionHandler);

        executor.prestartAllCoreThreads();
        List<Runnable> taskGroup = new ArrayList<Runnable>();
        taskGroup.add(new TestOne());
        taskGroup.add(new TestTwo());
        taskGroup.add(new TestThree());

        worksQueue.add(new MultiRunnable(taskGroup));
    }
}

class RejectedExecutionHandelerImpl implements RejectedExecutionHandler
{
    @Override
    public void rejectedExecution(Runnable runnable,
            ThreadPoolExecutor executor)
    {
        System.out.println(runnable.toString() + " : I've been rejected ! ");
    }
}
```

3. Java executor framework best practices

1. Always run your java code against static analysis tools like PMD and FindBugs to look for deeper issues. They are very helpful in determining ugly situations which may arise in future.
2. Always cross check and better plan a code review with senior guys to detect and possible deadlock or livelock in code during execution. Adding a health monitor in your application to check the status of running tasks is an excellent choice in most of the scenarios.
3. In multi-threaded programs, make a habit of catching errors too, not just exceptions. Sometimes unexpected things happen and Java throws an error at you, apart from an exception.

4. Use a back-off switch, so if something goes wrong and is non-recoverable, you don't escalate the situation by eagerly starting another loop. Instead, you need to wait until the situation goes back to normal and then start again.
5. Please note that the whole point of executors is to abstract away the specifics of execution, so ordering is not guaranteed unless explicitly stated.

Java Multi-Threading With the ExecutorService

theExeutorService can be used to run multi-threaded asynchronous tasks. We'll start by creating threads directly and then move on to explore the ExeutorService and how it can be used to simplify things.

Creating Threads Directly

Before the Executor API came along, developers were responsible for instantiating and managing threads directly. Let's look at a simple example below.

```
1
/**
2
 * Call 2 expensive methods on separate threads
3
 *
4
 * @throws InterruptedException
5
 */
6
public void doMultiThreadedWork() throws InterruptedException {
7

8
  /* create Runnable using anonymous inner class */
9
  Thread t1 = new Thread(new Runnable() {
10
    public void run() {
11
      System.out.println("starting expensive task thread t1");
12
      doSomethingExpensive();
13
```

```
    System.out.println("finished expensive task thread t1");
14
  }
15
});
16

17
/* start processing on new threads */
18
t1.start();
19

20
/* block current thread until t1 has finished */
21
t1.join();
22
}
```

In the method above, we create a new Thread t1 and pass a Runnable to its constructor. An anonymous inner class implements Runnable where the run() method contains the logic that will be executed by the Thread when it is started. Note that if the code inside run() throws a checked Exception, it must be caught and handled inside the method.

Introducing the Executor Service

Dealing with threads directly can be cumbersome, so Oracle simplified things by providing a layer of abstraction via its Executor API. An Executor allows you to process tasks asynchronously without having to deal with threads directly.

Creating an Executor

The Executors factory class is used to create an instance of an Executor, either an ExecutorService or an ScheduledExecutorService. Some of the most common types of Executor are described below.

- Executors.newCachedThreadPool() — An ExecutorService with a thread pool that creates new threads as required but reuses previously created threads as they become available.
- Executors.newFixedThreadPool(int numThreads) — An ExecutorServicethat has a thread pool with a fixed number of threads. The numThreads parameter is the maximum number of threads that can be active in the ExecutorService at any one time. If the number of requests submitted to the pool exceeds the pool size, requests are queued until a thread becomes available.
- Executors.newScheduledThreadPool(int numThreads) — A ScheduledExecutorServicewith a thread pool that is used to run tasks periodically or after a specified delay.

- Executors.newSingleThreadExecutor() — An ExecutorService with a single thread. Tasks submitted will be executed one at a time and in the order submitted.
- Executors.newSingleThreadScheduledExecutor() — An ExecutorService that uses a single thread to execute tasks periodically or after a specified delay.

The snippet below creates a fixed thread pool ExecutorService with a pool size of 2. I'll use this ExecutorService in the sections that follow.

```
ExecutorService executorService = Executors.newFixedThreadPool(2);
```

In the following sections, we'll look at how ExecutorService can be used to create and manage asynchronous tasks.

execute(Runnable)

The execute method takes a Runnable and is useful when you want to run a task and are not concerned about checking its status or obtaining a result. Think of it as fire and forget asynchronous task.

```
1
executorService.execute(()->{
2
  System.out.println(String.format("starting expensive task thread %s", Thread.currentThread().getName(
)));
3
  doSomethingExpensive();
4
}
```

Unlike the first Thread example, which used an anonymous inner class, the example above creates a Runnable using a lambda expression. The Runnable will be executed as soon as a thread is available from the ExecutorService thread pool.

Future<?> submit(Runnable)

Like execute(), the submit() method also takes a Runnable but differs from execute()because it returns a Future. A Future is an object that represents the pending response from an asynchronous task. Think of it as a handle that can be used to check the status of the task or retrieve its result when the task completes. Futures use generics to allow you to specify the return type of the task. However, given that the Runnablerun() method has the return type void, the Future holds the status of the task rather than a pending result. This is represented as Future<?> in the example below.

```
1
Future<?> taskStatus = executorService.submit(()->{
2
```

```
  System.out.println(String.format("starting expensive task thread %s", Thread.currentThread().getName(
)));
3

  doSomethingExpensive();
4

}
```

The*submit(Runnable)*method is useful when you want to run a task that doesn't return a value but you'd like to check the status of the task after it's been submitted to the ExecutorService.

Checking the Status of a Task

Future has a few useful methods for checking the status of a task that's been submitted to the ExecutorService.

- isCancelled() checks if the submitted task has already been canceled.
- isDone() checks if the submitted task has already completed. When a task has finished, isDone will return true whether the task completed successfully, unsuccessfully, or was canceled.
- cancel() cancels the submitted task. A boolean parameter specifies whether or not the task should be interrupted if it has already started.

```
1
/* check if both tasks have completed - if not sleep current thread
2

 * for 1 second and check again
3

 */
4

while(!task1Future.isDone() || !task2Future.isDone()){
5

  System.out.println("Task 1 and Task 2 are not yet complete....sleeping");
6

  Thread.sleep(1000);
7

}
```

Future<T> submit(Callable)

The submitmethod is overloaded to take a Callable as well as a Runnable. Like a Runnable, a Callable represents a task that is executed on another thread. A Callable differs from a Runable because it returns a value and can throw a checked Exception. The Callable interface has a single abstract

method public T call() throws Exception and like Runable can be implemented with an anonymous inner class or lambda. The return type of the call() method is used to type the Future returned by the ExecutorService. Two code snippets below show how a Callable can be created via an anonymous inner class and a lambda expression.

```
1
Future<Double> task1Future = executorService.submit(new Callable<Double>() {
2

3
  public Double call() throws Exception {
4

5
    System.out.println(String.format("starting expensive task thread %s",
6
        Thread.currentThread().getName()));
7
    Double returnedValue = someExpensiveRemoteCall();
8

9
    return returnedValue;
10
  }
11
});
1
Future<Double> task2Future = executorService.submit(()->{
2

3
  System.out.println(String.format("starting expensive task thread %s", Thread.currentThread().getName(
)));
4
  Double returnedValue = someExpensiveRemoteCall();
5

6
```

```
  return returnedValue;
7
});
```

Both examples create a Callable and pass it to the execute method. The Callable is executed as soon as a thread is available.

Getting a Result from a Future

When a Callable is submitted to the ExecutorService, we receive a Future with the return type of the call() method. In the example above, call() returns a Double so we get a Future<Double>. One way of retrieving the result from a Future is by calling its get() method. get() will block indefinitely waiting on the submitted task to complete. If the task doesn't complete or takes a long time to complete, the main application thread will remain blocked.

Waiting indefinitely for a result is usually not ideal. We'd rather have more control over how we retrieve the result and take some action if a task doesn't complete within a certain amount of time. Luckily there's an overloaded get(long timeout, TimeUnit unit) method that waits for the specified period of time and if the task hasn't finished (result not available), throws a TimeoutException.

```
1
Double value1 = task1Future.get();
2
Double value2 = task2Future.get(4,  TimeUnit.SECONDS); // throws TimeoutException
```

Submitting Multiple Callables

As well as allowing you to submit of a single Callable, the ExecutorService allows you to submit a Collection of Callable using the invokeAll method. As you might expect, instead of returning a single Future, a Collection of Futures is returned. A Future is returned representing the pending result of each submitted task.

```
1
Collection<Callable<Double>> callables = new ArrayList<>();
2
IntStream.rangeClosed(1, 8).forEach(i-> {
3
  callables.add(createCallable());
4
});
5

6
```

```
/* invoke all supplied Callables */
7
List<Future<Double>> taskFutureList = executorService.invokeAll(callables);
8

9
/* call get on Futures to retrieve result when it becomes available.
10
 * If specified period elapses before result is returned a TimeoutException
11
 * is thrown
12
 */
13
for (Future<Double> future : taskFutureList) {
14

15
  /* get Double result from Future when it becomes available */
16
  Double value = future.get(4, TimeUnit.SECONDS);
17
  System.out.println(String.format("TaskFuture returned value %s", value));
18
}
```

The code snippet above submits 8 Callable to the ExecutorService and retrieves a List containing 8 Future. The list of Future returned is in the same order as the Callables were submitted. Note that submitting multiple Callable s will require the size of the thread pool to be tweaked if we want most or all of the submitted tasks can be executed in parallel. In the example above, we'd need a thread pool with 8 threads to run all tasks in parallel.

Shutting Down the ExecutorService

After all the tasks have completed, its important to shut down the ExecutorService gracefully so that resources used can be reclaimed. There are two methods available, shutDown() and shutDownNow(). shutDown() triggers a shutdown of the ExecutorService, allowing currently processing tasks to finish but rejecting newly submitted tasks.

shutDownNow() also triggers a shutdown of the ExecutorService, but does not allow currently executing tasks to complete and attempts to terminate them immediately. shutDownNow() returns a list of tasks that

were queued for execution when the shutdown was initiated. To ensure the ExecutorService is shut down in all cases and to avoid potential resource leaks, it's important that shutDown() or shutDownNow() is called inside a finally block.

```
1
ExecutorService executorService = null;
2

3
try{
4
  executorService = Executors.newFixedThreadPool(2);
5

6
  executorService.execute(()->{
7
    System.out.println(String.format("starting expensive task thread %s", Thread.currentThread().getName
()));
8
    doSomethingExpensive();
9
  });
10

11
}
12
finally{
13
  executorService.shutdown();
14
}
```

Wrapping Up

In this post, we looked at theExecutorService and how it can be used to simplify the creation and management of asynchronous tasks. The source code that accompanies this post is available on GitHub so why not pull the code and have a play around.

MCQ

1. Which of these packages contain all the collection classes?

- **A.** java.lang
- **B.** java.util
- **C.** java.net
- **D.** java.awt

- **Answer & Explanation**

- Answer: Option B
- Explanation:
- java.util

2. Which of these classes is not part of Java's collection framework?

- **A.** Maps
- **B.** Array
- **C.** Stack
- **D.** Queue

Answer & Explanation

Answer: Option D

Explanation:

Queue is not a part of collection framework.

3. Which of these interface is not a part of Java's collection framework?

- **A.** List
- **B.** Set
- **C.** SortedMap
- **D.** SortedList

Answer & Explanation

Answer: Option D

Explanation:

SortedList is not a part of collection framework.

4. Which of these methods deletes all the elements from invoking collection?

- **A.** clear()
- **B.** reset()
- **C.** delete()
- **D.** refresh()

Answer & Explanation

Answer: Option A

Explanation:

clear() method removes all the elements from invoking collection.

5. What is Collection in Java?

- **A.** A group of objects
- **B.** A group of classes
- **C.** A group of interfaces
- **D.** None of the mentioned

Answer & Explanation

Answer: Option A

Explanation:

A collection is a group of objects, it is similar to String Template Library (STL) of C++ programming language.