Java Concurrency: The Basics

The power of multi-threading using Java

Tiago Albuquerque · Follow
Published in The Startup
8 min read · Oct 23, 2019

Listen Share

This article describes the basics of Java Concurrency API, and how to assign work to be done asynchronously.

Why not do the work concurrently?

A *thread* is the smallest unit of execution that can be scheduled by the operating system, while a process is a group of associated threads that execute in the same, shared environment.

We call “system thread” the thread created by the JVM that runs in the background of the application (e.g. garbage-collector), and a “user-defined thread” those which are created by the developer.

First, we need to know that in a standard Java program, all the work we implement is done in the *main thread,* so the work is done sequentially.

To change that behavior, we can create our own threads and program them to do the work “at the same time”, creating a multi-threaded environment.

We can create threads extending the *Thread class* or implement the *Runnable interface*. Both use the method *run()* to execute asynchronous work. A thread must be started by its *start()* method, otherwise the work is not forked in a different thread.

Extending Thread class

By extending the Thread class, we shall override the “run()” method.

Given the two classes “Ping” and “Pong” below, they simulates 3 time processing job that takes from 0 to 4 seconds to accomplish each one:

```
class Ping extends Thread {
   @Override
   public void run() {
      for (int i = 0; i < 3; i++) {
         // simulates work that takes between from 0 to 4 sec
         try { Thread.sleep(new Random().nextInt(4000)); }
         catch (InterruptedException e) { }
         System.out.println("ping");
      }
   }
}
```

```
class Pong extends Thread {
   @Override
   public void run() {
      for (int i = 0; i < 3; i++) {
         // simulates work that takes between from 0 to 4 sec
         try { Thread.sleep(new Random().nextInt(4000)); }
         catch (InterruptedException e) { }
         System.out.println("pong");
      }
   }
}
```

If we instantiate them and call each method run(), the work will be done sequentially in the main thread, which is not our intention here:

```
public class MainClass {
   public static void main(String[] args) {
      System.out.println("Start of main thread");
      Thread ping = new Ping();
      Thread pong = new Pong();
      ping.run();   // wrong
      pong.run();   // wrong
      System.out.println("End of main thread");
   }
}
```

Output is always:

```
Start of main thread
ping
ping
ping
pong
pong
pong
End of main thread
```

But if we start the threads calling its start method, the work will be done asynchronously and there is no guarantee of the execution order.

```
public class MainClass {
   public static void main(String[] args) {
      System.out.println("Start of main thread");
      Thread ping = new Ping();
      Thread pong = new Pong();
      ping.start();   // async
      pong.start();   // async
      System.out.println("End of main thread");
   }
}
```

Output may be:

```
Start of main thread
End of main thread
pong
ping
ping
pong
pong
ping
```

Implementing Runnable interface

Another way to create a thread is to implement the Runnable interface, which can be an advantage if your worker class already extends another class. Since there is no multiple-inheritance in Java, it would be impossible to extend to the Thread class.

The only modifications in the previous code is the *implements* keyword in worker classes:

```
class Ping implements Runnable {
   @Override
   public void run() {
      for (int i = 0; i < 3; i++) {
         // simulates work that takes between from 0 to 4 sec
```

```
            try { Thread.sleep(new Random().nextInt(4000)); }
            catch (InterruptedException e) { }
            System.out.println("ping");
        }
    }
}

class Pong implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            // simulates work that takes between from 0 to 4 sec
            try { Thread.sleep(new Random().nextInt(4000)); }
            catch (InterruptedException e) { }
            System.out.println("pong");
        }
    }
}
```

And the instantiation of the thread is passing by the Runnable implementation in its constructor.

```
public class MainClass {
    public static void main(String[] args) {
        System.out.println("Start of main thread");
        Thread ping = new Thread(new Ping());
        Thread pong = new Thread(new Pong());
        ping.start();   // async
        pong.start();   // async
        System.out.println("End of main thread");
    }
}
```

Again there is no guarantee of the execution order.

Since the Runnable interface is a *functional interface,* we can use a lambda expression to create the Runnable instance. The readability of the code gets a little messy (in my opinion), but still a way to go.

```
public class MainClass {
    public static void main(String[] args) {
        System.out.println("Start of main thread");
        Thread ping = new Thread(() -> {
```

```
            for (int i = 0; i < 3; i++) {
               try { Thread.sleep(new Random().nextInt(2000)); }
               catch (InterruptedException e) { }
               System.out.println("ping");
            }
         });
         Thread pong = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
               try { Thread.sleep(new Random().nextInt(2000)); }
               catch (InterruptedException e) { }
               System.out.println("pong");
            }
         });
         ping.start();   // async
         pong.start();   // async
         System.out.println("End of main thread");
      }
   }
```

Threads attributes and methods

To ensure that the main thread finish its execution after the worker threads, we can *join* it, pausing its execution while the worker threads are alive.

```
public class MainClass {
   public static void main(String[] args) throws InterruptedException
{
      System.out.println("Start of main thread");
      Thread ping = new Thread(() -> {
         for (int i = 0; i < 3; i++) {
            try { Thread.sleep(new Random().nextInt(2000)); }
            catch (InterruptedException e) { }
            System.out.println("ping");
         }
      });
      Thread pong = new Thread(() -> {
         for (int i = 0; i < 3; i++) {
            try { Thread.sleep(new Random().nextInt(2000)); }
            catch (InterruptedException e) { }
            System.out.println("pong");
         }
      });
```

```
        ping.start();
        pong.start();
        ping.join();
        pong.join();
        // will be printed last:
        System.out.println("End of main thread");
    }
}
```

The output may be as follows, and its guaranteed that the last line will be the "*End of main thread*" in the example above.

```
Start of main thread
pong
pong
ping
pong
ping
ping
End of main thread
```

Threads has an attribute *name*, that we can use to identify them during program execution. The default name is "*Thread-n*", where 'n' is the incremental number of creation.

```
// setting thread name by constructor or setter
new Thread(runnableInstante,"My Thread Name");
threadInstance.setName("Ping Thread");
// getting thread name
threadInstance.getName();
```

Thread ***priority*** is a numeric (integer) value associated with a thread that is taken into consideration by the scheduler when determining which thread should currently be executing. There are static constants that helps us out:

- Thread.**MIN_PRIORITY** // *int value = 1*
- Thread.**NORM_PRIORITY** // *int value = 5, it's the default value*

- Thread.**MAX_PRIORITY** // *int value =10*

An issue to be aware of is the synchronization between data accessed by threads. If an value is accessed by many threads to make computation, its results can varies depending on the concurrent read made.

For example, let's assume a simple counter class:

```
class Counter {
   int count;
   public void increment() {
      count++; // count = count + 1
   }
}
```

And the main class that consumes it using two async threads:

```
public class MainClass {
   public static void main(String[] args) throws InterruptedException
{
      Counter counter = new Counter();

      Thread c1 = new Thread(() -> {
         for (int i = 0; i < 500; i++) {
            counter.increment();
         }
      });
      Thread c2 = new Thread(() -> {
         for (int i = 0; i < 500; i++) {
            counter.increment();
         }
      });

      c1.start(); // async counter thread 1 start
      c2.start(); // async counter thread 1 start
      // awating async threads to finish
      c1.join();
      c2.join();
      System.out.println(counter.count);
   }
}
```

The output varies on a number equals or less than 1000. The issue is on the '*cont*' attribute, that can had been incremented by the other thread while it's being read by the first one. To ensure that just one thread will be executing the '*increment()*' method, we can use the **synchronized** keyword, so we always get the correct sum of values ('1000', in this case).

```
class Counter {
   int count;
   public synchronized void increment() {
      count++; // count = count + 1
   }
}
```

Interface ExecutorService and Threads / Callables

The *ExecutorService* interface is the easiest and recommended way to create and manage async tasks. We must first obtain an instance of an *ExecutorService* by using some *factory methods*, and then send the service tasks to be processed.

To instantiate an *ExecutorService*, we should use one of the *Executors* static methods, depending of the way we want to schedule the tasks and the amount of threads used to do the work.

- *Executors.newSingleThreadExecutor()* : Creates a single-threaded executor. Results are processed sequentially.
- *Executors.newFixedThreadPool(int nrOfThreads)* : Creates a thread pool that reuses a fixed number of threads.
- *Executors.newCachedThreadPool()* : Creates a thread pool that creates new threads as needed, but will reuse previous threads when they are available.
- *Executors.newSingleThreadScheduledExecutor()* : Creates a single-threaded executor that can schedule commands to run after a given delay or to execute periodically.

- *Executors.newScheduledThreadPool(int nrOfThreads)* : Creates a thread pool that can schedule commands to run after a given delay or to execute periodically.

First we create a Runnable implementation:

```
class TaskRunn implements Runnable {
   @Override
   public void run() {
      System.out.println("I am " +
          Thread.currentThread().getName());
   }
}
```

And then assign the work to an ExecutorService to manage:

```
public class MainClass {
   public static void main(String[] args) {
      // pool of 4 threads
      ExecutorService executorService =
           Executors.newFixedThreadPool(4);
      // creates 4 runnables to be executed
      for (int i = 0; i < 4; i++) {
         TaskRunn runnable = new TaskRunn();
         executorService.execute(runnable);
      }
      executorService.shutdown(); // shuts down the executor service
}
```

The output may be (no order guaranteed):

```
I am pool-1-thread-2
I am pool-1-thread-4
I am pool-1-thread-1
I am pool-1-thread-3
```

When we need to get a returned value, we should implement a ***Callable*** instead of a *Runnable*. The *Callable* interface has the "*call()*" method that returns a value and can throw a checked exception.

```
class TaskCall implements Callable<String> {
   @Override
   public String call() throws Exception {
      return "I am " + Thread.currentThread().getName();
   }
}
```

When invoking a Callable task within ExecutorService, we get a *Future* object, which represents the result of async computation when it is over:

```
public class MainClass {
   public static void main(String[] args) {
      // pool of 4 threads
      ExecutorService executorService =
         Executors.newFixedThreadPool(4);

      List<Future<String>> futures = new ArrayList<>();
      // creates 4 callables to be executed
      for (int i = 0; i < 4; i++) {
         TaskCall callable = new TaskCall();
         Future<String> future = executorService.submit(callable);
         futures.add(future);
      }
      // getting tasks results
      futures.forEach(f -> {
         try { System.out.println(f.get()); }
         catch (Exception e) { }
      });

      executorService.shutdown(); // shuts down the executor service
}
```

We can user the "*invokeAll()*" method to send a list of callables to be executed asynchronoulsy:

```
public class BasicConcurrencyTests {
   public static void main(String[] args)
      throws InterruptedException {
      // pool of 4 threads
      ExecutorService executorService =
            Executors.newFixedThreadPool(4);
```

```
        // creates 3 callables to be executed
        List<Callable<String>> callables = Arrays.asList(
              new TaskCall(), new TaskCall(), new TaskCall());
        List<Future<String>> futures =
              executorService.invokeAll(callables);
```



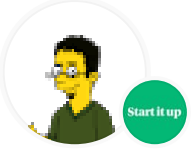
```
        executorService.shutdown(); // shuts down the executor service
  }
```

Example of output:

```
I am pool-1-thread-1
I am pool-1-thread-2
I am pool-1-thread-3
```

There is a lot more, like thread safe data structures and scheduled tasks, but I guess this content covers the very basics of Java concurrent API.

Java Threads Concurrency

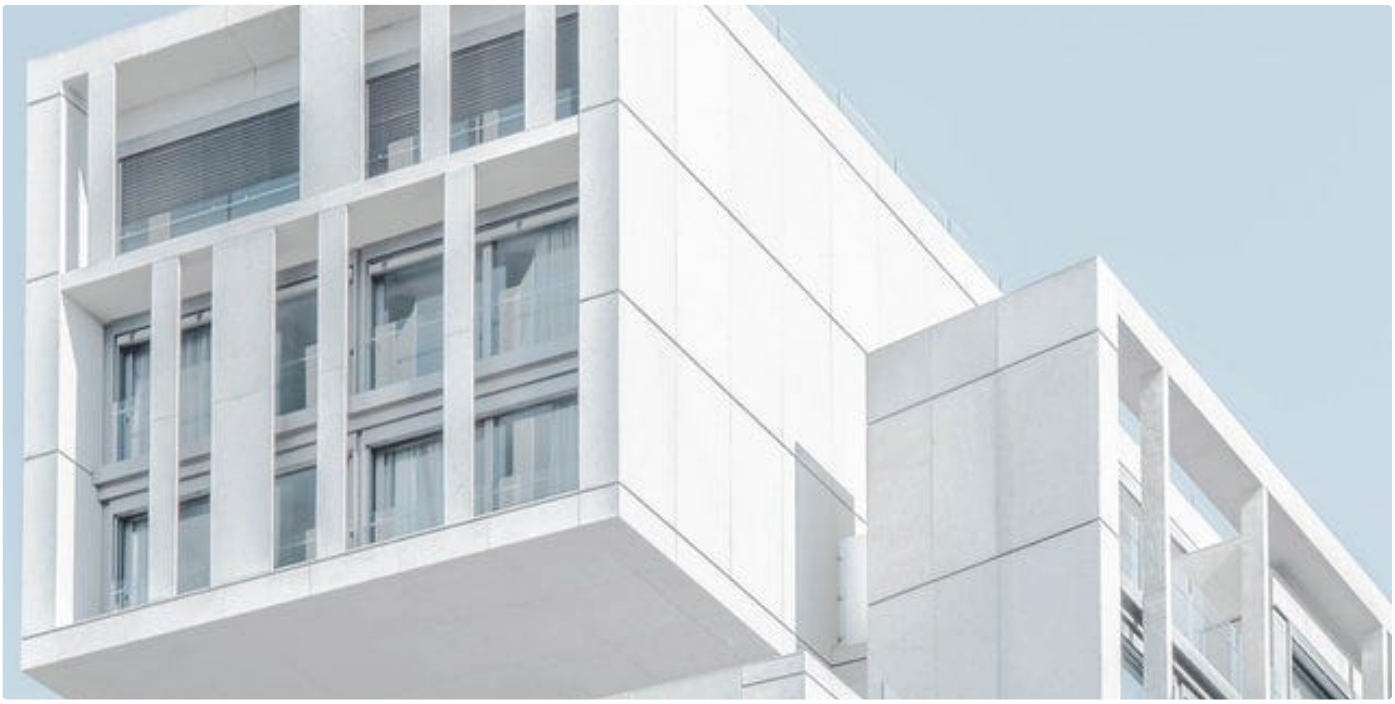


Written by Tiago Albuquerque

136 Followers · Writer for The Startup

Software Craftsman

More from Tiago Albuquerque and The Startup



Tiago Albuquerque in The Dev Project

Clean Architecture: a basic example of folders organization

Implementing a practical demo application using Typescript

9 min read · Apr 5, 2022

97 5

Tim Denning in The Startup

I Met a Quiet Millionaire Who Operates a $2.5m Tiny Business While Working 2-3 Hours a Day

This is bizarrely what he taught me

· 6 min read · Jul 31

6.1K 143

Jano le Roux in The Startup

7 (Miserable) Tell-Tell Signs AI Wrote Your Article

ChatGPT is wrecking writing as we know it.

· 4 min read · Jul 13

8.5K 228

Tiago Albuquerque

REST API with Java — Part 4

API Authentication and Authorization with Spring Security

8 min read · Dec 2, 2021

3

See all from Tiago Albuquerque

See all from The Startup

Recommended from Medium

Prabhash

35 very important interview questions

For Java, Spring Boot, Microservices Engineer Position

✦ · 2 min read · Mar 5

110

Bahaddin Ahmadov in Javarevisited

Java 8, 9, 11, and 17: What You Need to Know About the Key Differences Between These Versions

Introduction

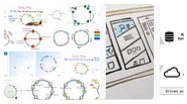
✦ · 18 min read · Feb 15

120

Lists

It's never too late or early to start something

13 stories · 72 saves

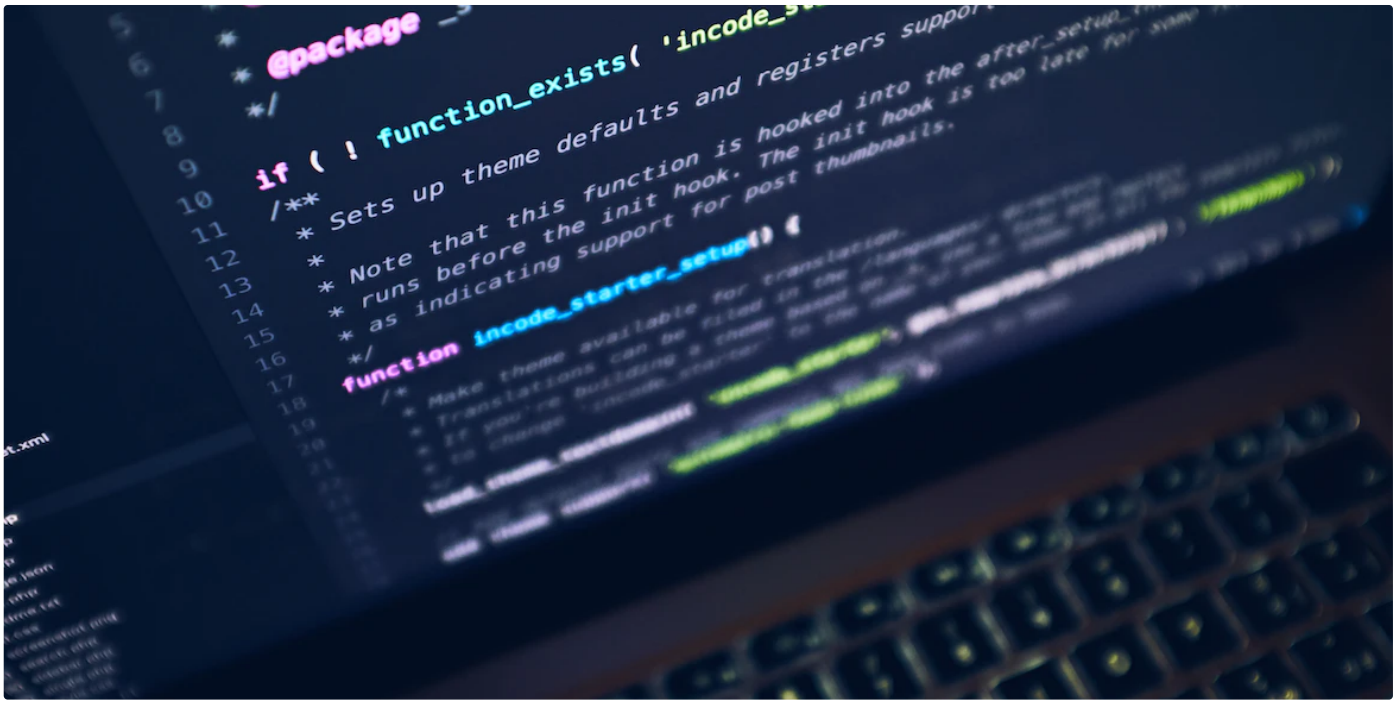


General Coding Knowledge

20 stories · 213 saves

New_Reading_List

174 stories · 67 saves



Tech Is Beautiful in Dev Genius

Mastering Concurrency in Java: 8 Best Practices Every Java Developer Should Follow

Boost Your Java Development Skills with These Essential Concurrency Best Practices

· 8 min read · Feb 19

73

Daryl Goh

Spring Boot: RequestEntity vs ResponseEntity | RequestBody vs ResponseBody

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.

· 5 min read · May 21


```
commit ffcf2c01b7ef612893529cef188cc1961ed64521 (HEAD -> master, origin/master, origin/bors/staging, origin/HEAD)
Merge: fc991bf81 5159211da
Author: iohk-bors[bot] <43231472+iohk-bors[bot]@users.noreply.github.com>
Date:   Tue Nov 8 17:44:34 2022 +0000

    Merge #4563

    4563: New p2p topology file format r=coot a=coot

    Fixes #4559.


    Co-authored-by: Marcin Szamotulski <coot@coot.me>
    Co-authored-by: olgahryniuk <67585499+olgahryniuk@users.noreply.github.com>

commit fc991bf814891a9349f22cf278632d39b04d4628
Merge: 5633d1c05 5cd94d372
Author: iohk-bors[bot] <43231472+iohk-bors[bot]@users.noreply.github.com>
Date:   Tue Nov 8 13:07:58 2022 +0000

    Merge #4613

    4613: Update building-the-node-using-nix.md r=CarlosLopezDeLara a=CarlosLopezDeLara

    Build the cardano-node executable. No default configuration.

    Co-authored-by: CarlosLopezDeLara <carlos.lopezdelara@iohk.io>

commit 5159211da7a644686a973e4fb316b64ebb1aa34c
Author: olgahryniuk <67585499+olgahryniuk@users.noreply.github.com>
Date:   Tue Nov 8 13:25:10 2022 +0200
```

Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

· 4 min read · Nov 14, 2022

8.9K 102

Joey

Top 100 Java Interview Questions for 1 to 3 Years Experienced Programmers

containers, multithreading, reflection, object copy, Java Web, exceptional cases, internet, Spring MVC.

· 4 min read · Feb 28

153 1

See more recommendations