Baeldung

Java Concurrency Guide

V 1.0.2

1. Life Cycle of a Thread in Java

2. How to Start a Thread in Java

3. wait and notify() Methods in Java

1. Life Cycle of a Thread in Java

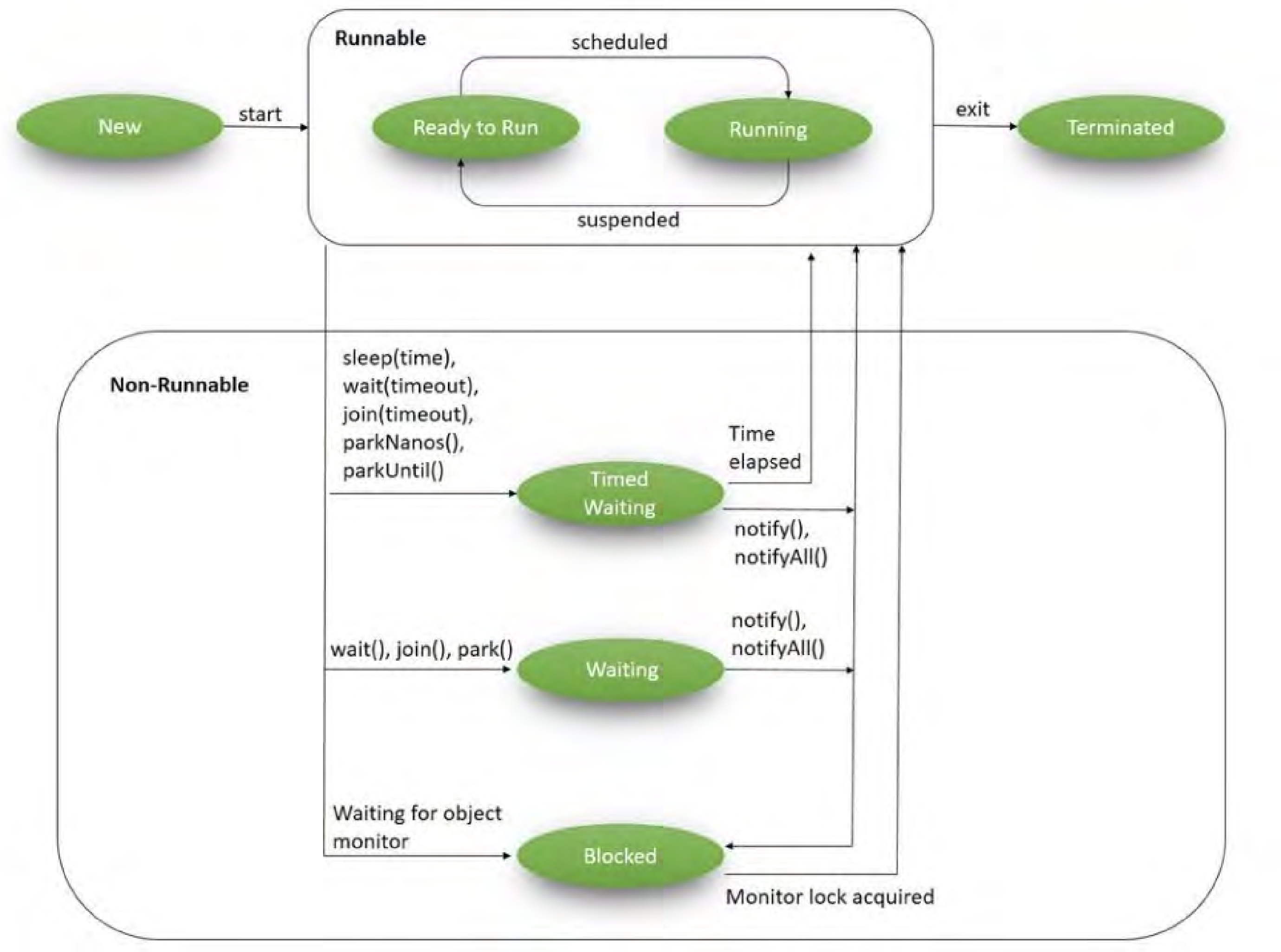
1. Introduction

In this chapter, we'll discuss a core concept in Java in detail: the lifecycle of a thread.

We'll use a quick illustrated diagram and practical code snippets to understand these states during the thread execution better.

To better understand Threads in Java, this article on creating a thread is a good place to start.

In the Java language, multithreading is driven by the core concept of a Thread. During their lifecycle, threads go through various states:



The java.lang.Thread class contains a static State enum that defines its potential states. During any given point in time, the thread can only be in one of these states:

NEW – a newly created thread that hasn't yet started the execution
RUNNABLE – either running or ready for execution but waiting for resource allocation
BLOCKED – waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
WAITING – waiting for some other thread to perform a particular action without any time limit
TIMED_WAITING – waiting for some other thread to perform a specific action for a specified period
TERMINATED – has completed its execution

All of these states are covered in the diagram above; let's discuss each in detail.

3.1. New

A *NEW Thread* (or a Born *Thread*) is a thread that's been created but not yet started. It remains in this state until we start it using the *start()* method. The following code snippet shows a newly created thread that's in the NEW state:

```java
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
System.out.println(t.getState());
```

Since we've not yet started the mentioned thread, the method *t.getState()* prints:

```
NEW
```

3.2. Runnable

Once we've created a new thread and called the *start()* method on it, it's moved from the NEW to RUNNABLE state. **Threads in this state are either running or ready to run, but they're waiting for resource allocation from the system.**

In a multi-threaded environment, the Thread-Scheduler (part of JVM) allocates a fixed amount of time to each thread. So it runs for a particular time, then relinquishes the control to other RUNNABLE threads.

For example, let's add the *t.start()* method to our previous code and try to access its current state:

```
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
t.start();
System.out.println(t.getState());
```

This code is most likely to return the output as:

RUNNABLE

Note that in this example, it's not always guaranteed that by the time our control reaches *t.getState()*, it will still be in the *RUNNABLE* state.

It may be that it was immediately scheduled by the Thread-Scheduler and has already finished execution. In such cases, we may get a different output.

3.3. Blocked

A thread is in the *BLOCKED* state when it's not currently eligible to run. **It enters this state while waiting for a monitor lock and trying to access a section of code that's locked by some other thread.**

Let's try to reproduce this state:

```
public class BlockedState {
    public static void main(String[] args) throws
InterruptedException {
        Thread t1 = new Thread(new DemoBlockedRunnable());
        Thread t2 = new Thread(new DemoBlockedRunnable());

        t1.start();
        t2.start();

        Thread.sleep(1000);

        System.out.println(t2.getState());
        System.exit(0);
    }
}

class DemoBlockedRunnable implements Runnable {
    @Override
    public void run() {
        commonResource();
    }

    public static synchronized void () {
        while(true) {
            // Infinite loop to mimic heavy processing
            // 't1' won't leave this method
            // when 't2' tries to enter this
        }
    }
}
```

In this code:

We've created two different threads: t1 and t2.
t1 starts and enters the synchronized *commonResource()* method, meaning only one thread can access it. All other subsequent threads that try to access this method will be blocked from further execution until the current one finishes processing.
When t1 enters this method, it's kept in an infinite while loop; this is just to imitate heavy processing so that all other threads can't enter this method.
Now, when we start t2, it tries to enter the *commonResource()* method, which is already being accessed by t1; thus, t2 will be kept in the *BLOCKED* state.

Being in this state, if we call *t2.getState()*, we'll get the output as:

```
BLOCKED
```

3.4. Waiting

A thread is in the *WAITING* state when it's waiting for some other thread to perform a particular action. According to JavaDocs, any thread can enter this state by calling any one of the following three methods:

object.wait()
thread.join() or
LockSupport.park()

Note that in *wait()* and *join()*, we don't define any timeout period, as that scenario is covered in the next section.

We have a separate tutorial that discusses in detail the use of *wait()*, *notify()*, and *notifyAll()*.

For now, let's try to reproduce this state:

```java
public class ingState implements Runnable {
    public static Thread t1;

    public static void main(String[] args) {
        t1 = new Thread(new ingState());
        t1.start();
    }

    public void run() {
        Thread t2 = new Thread(new DemoingStateRunnable());
        t2.start();

        try {
            t2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}

class DemoingStateRunnable implements Runnable {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }

        System.out.println(ingState.t1.getState());
    }
}
```

Let's discuss what we're doing here:

We've created and started the *t1*.
t1 creates a *t2* and starts it.
While the processing of *t2* continues, we call *t2.join()*, which puts *t1* in the *WAITING* state until *t2* has finished execution.
Since *t1* is waiting for *t2* to complete, we're calling *t1.getState()* from *t2*.

As we expect, the output here is:

```
WAITING
```

3.5. Timed Waiting

A thread is in the *TIMED_WAITING* state when it s for another thread to perform a particular action within a stipulated time.

According to JavaDocs, there are five ways to put a thread in the TIMED_WAITING state:

thread.sleep(long millis)
(int timeout) or (int timeout, int nanos)
thread.join(long millis)
LockSupport.parkNanos
LockSupport.parkUntil

To read more about the differences between *wait()* and *sleep()* in Java, look at this dedicated article here.

For now, let's try to reproduce this state quickly:

```java
public class TimedingState {
    public static void main(String[] args) throws InterruptedException {
        DemoTimeingRunnable runnable= new DemoTimeingRunnable();
        Thread t1 = new Thread(runnable);
        t1.start();

        // The following sleep will give enough time for ThreadScheduler
        // to start processing of thread t1
        Thread.sleep(1000);
        System.out.println(t1.getState());
    }
}

class DemoTimeingRunnable implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Here, we've created and started a thread *t1*, which is entered into the sleep state with a timeout period of 5 seconds. The output here will be:

TIMED_WAITING

3.6. Terminated

This is the state of a dead thread. **It's in the *TERMINATED* state when it has either finished execution or was terminated abnormally**.

We have a dedicated article that discusses different ways of stopping the thread.

Let's try to achieve this state in the following example:

```
public class TerminatedState implements Runnable {
    public static void main(String[] args) throws
InterruptedException {
        Thread t1 = new Thread(new TerminatedState());
        t1.start();
        // The following sleep method will give enough time for
        // thread t1 to complete
        Thread.sleep(1000);
        System.out.println(t1.getState());
    }

    @Override
    public void run() {
        // No processing in this block
    }
}
```

Here, while we've started thread *t1*, the very next statement, *Thread.sleep(1000),* gives enough time for *t1* to complete, so this program gives us the output:

TERMINATED

In addition to the thread state, we can check the *isAlive()* method to determine whether the thread is alive. For instance, if we call the *isAlive()* method on this thread, we'll get:

```
Assert.assertFalse(t1.isAlive());
```

It returns *false*. Simply put, a **thread is alive if, and only if, it's been started and hasn't yet died.**

4. Conclusion

In this chapter, we learned about the life cycle of a thread in Java. We looked at all six states defined by the *Thread.State enum*, and reproduced them with quick examples.

Although the code snippets will give the same output in almost every machine, in some exceptional cases, we may get different outputs, as the exact behavior of Thread Scheduler can't be determined.

As always, the code snippets used here are available on GitHub.

2. How to Start a Thread in Java

1. Introduction

In this chapter, we'll explore different ways to start a thread and execute parallel tasks.

This is very useful, particularly when dealing with long or recurring operations that can't run on the main thread or where the UI interaction can't be put on hold while waiting for the operation's results.

To learn more about the details of threads, read our chapter on the Life Cycle of a Thread in Java.

We can easily write some logic that runs in a parallel thread by using the Thread framework.

Let's try a basic example by extending the *Thread* class:

```
public class NewThread extends Thread {
    public void run() {
        long startTime = System.currentTimeMillis();
        int i = 0;
        while (true) {
            System.out.println(this.getName() + ": New Thread is 
running..." + i++);
            try {
                // for one sec so it doesn't print too fast
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            ...
        }
    }
}
```

Next, we can write a second class to initialize and start our thread:

```
public class SingleThreadExample {
    public static void main(String[] args) {
        NewThread t = new NewThread();
        t.start();
    }
}
```

We should call the *start()* method on threads in the *NEW* state (the equivalent of not started). Otherwise, Java will throw an instance of *IllegalThreadStateException* exception.

Now, let's assume we need to start multiple threads:

```
public class MultipleThreadsExample {
    public static void main(String[] args) {
        NewThread t1 = new NewThread();
        t1.setName("MyThread-1");
        NewThread t2 = new NewThread();
        t2.setName("MyThread-2");
        t1.start();
        t2.start();
    }
}
```

Our code still looks quite simple and very similar to the examples we can find online.

Of course, **this is far from production-ready code, where it's critical to manage resources correctly to avoid too much context switching or too much memory usage**.

To get production-ready, we now need to write additional boilerplate to deal with:

- the consistent creation of new threads
- the number of concurrent live threads
- the threads deallocation - very important for daemon threads to avoid leaks

We can write our code for all of these case scenarios if we want to, but why should we reinvent the wheel.

3. The *ExecutorService* Framework

The ExecutorService implements the Thread Pool design pattern (also called a replicated worker or worker-crew model) and takes care of the thread management we mentioned above. Plus, it adds some very useful features, like thread reusability and task queues.

Thread reusability, in particular, is very important. In a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

With worker threads, we minimize the overhead caused by thread creation.

To ease the pool configuration, ExecutorService comes with an easy constructor and some customization options, such as the type of queue, the minimum and maximum number of threads, and their naming convention.

For more details about the *ExecutorService*, please read our Guide to the Java ExecutorService.

Thanks to this powerful framework, we can switch our mindset from starting threads to submitting tasks.

Let’s look at how we can submit an asynchronous task to our executor:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
...
executor.submit(() -> {
    new Task();
});
```

There are two methods we can use: *execute*, which returns nothing, and submit, which returns a *Future* encapsulating the computation’s result.

For more information about Futures, please read our Guide to java.util.concurrent.Future.

To retrieve the final result from a Future object, we can use the get method available in the object, but this would block the parent thread until the end of the computation.

Alternatively, we could avoid the block by adding more logic to our task, but we'd have to increase the complexity of our code.

Java 1.8 introduced a new framework on top of the Future construct to better work with the computation's result: the *CompletableFuture*.

CompletableFuture implements ***CompletableStage***, which adds a vast selection of methods to attach callbacks and avoid all the plumbing needed to run operations on the result after it's ready.

The implementation to submit a task is a lot simpler:

```
CompletableFuture.supplyAsync(() -> “Hello”);
```

supplyAsync takes a *Supplier* containing the code we want to execute asynchronously, which is the lambda parameter in our case.

The task is now implicitly submitted to the *ForkJoinPool.commonPool()*, or we can specify the *Executor* we prefer as a second parameter.

To know more about CompletableFuture, please read our Guide To CompletableFuture.

When working with complex web applications, we may need to run tasks at specific times, maybe even regularly.

Java has a few tools that can help us to run delayed or recurring operations:

- *java.util.Timer*
- *java.util.concurrent.ScheduledThreadPoolExecutor*

6.1. *Timer*

The *timer* is a facility to schedule tasks for future execution in a background thread.

Tasks may be scheduled for one-time execution or repeated execution at regular intervals.

Let's see what the code looks like if we want to run a task after one second of delay:

```
TimerTask task = new TimerTask() {
    public void run() {
        System.out.println("Task performed on: " + new Date() +
"n"
          + "Thread's name: " + Thread.currentThread().getName());
    }
};
Timer timer = new Timer("Timer");
long delay = 1000L;
timer.schedule(task, delay);
```

Now, let's add a recurring schedule:

```
timer.scheduleAtFixedRate(repeatedTask, delay, period);
```

This time, the task will run after the delay specified, and it'll be recurrent after the period of time has passed.

For more information, please read our guide to Java Timer.

6.2. *ScheduledThreadPoolExecutor*

ScheduledThreadPoolExecutor has methods similar to the *Timer* class:

```
ScheduledExecutorService executorService = Executors.
newScheduledThreadPool(2);
ScheduledFuture<Object> resultFuture
  = executorService.schedule(callableTask, 1, TimeUnit.SECONDS);
```

To end our example, we'll use *scheduleAtFixedRate()* for recurring tasks:

```
ScheduledFuture<Object> resultFuture
 = executorService.scheduleAtFixedRate(runnableTask, 100, 450,
TimeUnit.MILLISECONDS);
```

The code above will execute a task after an initial delay of 100 milliseconds, and after that, it'll execute the same task every 450 milliseconds.

If the processor can't finish processing the task in time before the next occurrence, the *ScheduledExecutorService* will until the current task is completed before starting the next.

To avoid this waiting time, we can use *scheduleWithFixedDelay()*, which, as described by its name, guarantees a fixed length delay between iterations of the task.
For more details about *ScheduledExecutorService*, please read our Guide to the Java ExecutorService.

6.3. Which Tool is Better?

If we run the examples above, the computation's result looks the same.

So, **how do we choose the right tool?**

When a framework offers multiple choices, it's important to understand the underlying technology to make an informed decision.

Let's try to dive a bit deeper under the hood.

Timer:

- doesn't offer real-time guarantees; it schedules tasks using the *Object.(long)* method
- there's a single background thread, so tasks run sequentially, and a long-running task can delay others
- runtime exceptions thrown in a *TimerTask* would kill the only thread available, thus killing *Timer*

ScheduledThreadPoolExecutor:

- can be configured with any number of threads
- can take advantage of all available CPU cores
- catches runtime exceptions and lets us handle them if we want to (by overriding the afterExecute method from ThreadPoolExecutor)
- cancels the task that threw the exception while letting others continue to run
- relies on the OS scheduling system to keep track of time zones, delays, solar time, etc.
- provides collaborative API if we need coordination between multiple tasks, like waiting for the completion of all tasks submitted
- provides better API for management of the thread life cycle

The choice is obvious now, right?

In our code examples, we can observe that *ScheduledThreadPoolExecutor* returns a specific type of *Future*: *ScheduledFuture*.

ScheduledFuture extends both *Future* and *Delayed* interfaces, thus inheriting the additional method *getDelay*, which returns the remaining delay associated with the current task. It's extended by *RunnableScheduledFuture*, which adds a method to check if the task is periodic.

***ScheduledThreadPoolExecutor* implements all of these constructs through the inner class *ScheduledFutureTask*, and uses them to control the task life cycle.**

8. Conclusions

In this chapter, we experimented with the different frameworks available to start threads and run tasks in parallel.

Then, we explored in greater detail the differences between *Timer* and *ScheduledThreadPoolExecutor*.

The source code for the chapter is available over on GitHub.

3. wait and notify() Methods in Java

1. Overview

In this chapter, we'll look at one of the most fundamental mechanisms in Java: thread synchronization.

First, we'll discuss some essential concurrency-related terms and methodologies.

Then, we'll develop a simple application where we'll deal with concurrency issues, with the goal of better understanding *wait()* and *notify()*.

In a multithreaded environment, multiple threads might try to modify the same resource. Not managing threads properly will lead to consistency issues.

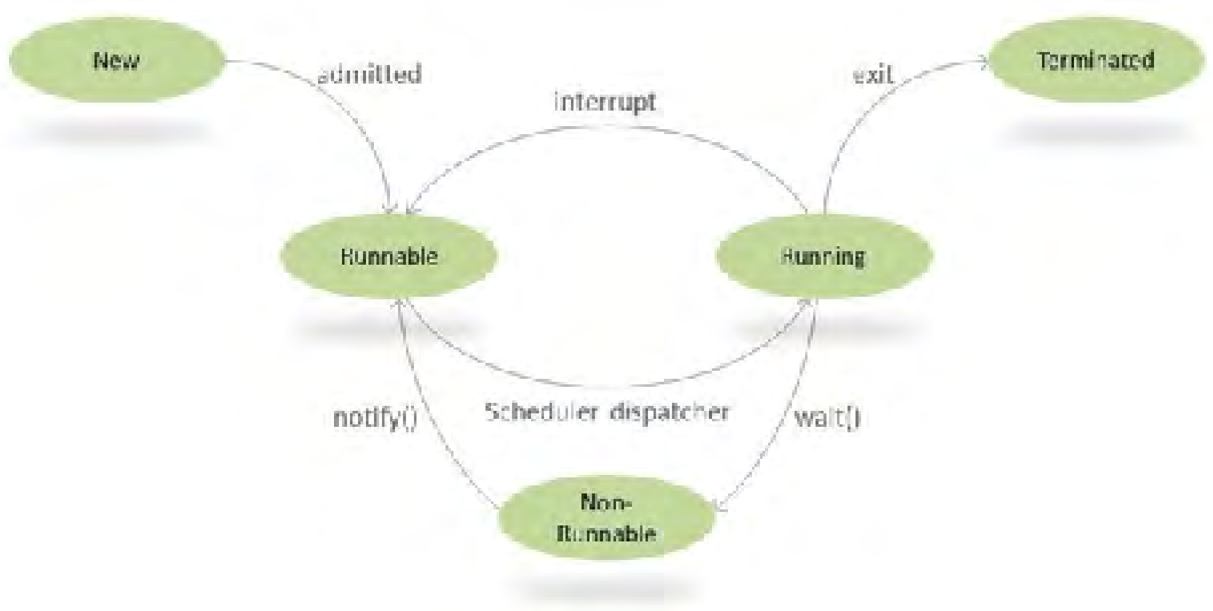
2.1. Guarded Blocks in Java

One tool we can use to coordinate the actions of multiple threads in Java is guarded blocks. Such blocks keep a check for a particular condition before resuming the execution.

With that in mind, we'll make use of the following:

- ***Object.()*** to suspend a thread
- ***Object.notify()*** to wake up a thread

We can better understand this through the following diagram depicting the life cycle of a *Thread*:



Please note that there are many ways of controlling this life cycle. However, in this article, we will focus only on *wait()* and *notify()*.

Simply put, calling *wait()* forces the current thread to until some other thread invokes *notify()* or *notifyAll()* on the same object.

For this, the current thread must own the object's monitor. According to Javadocs, this can happen in the following ways:

- when we've executed the *synchronized* instance method for the given object
- when we've executed the body of a *synchronized* block on the given object
- by executing *synchronized static* methods for objects of type *Class*

Note that only one active thread can own an object's monitor at a time.

This *wait()* method comes with three overloaded signatures. Let's have a look at these.

3.1. *wait()*

The *wait()* method causes the current thread to indefinitely until another thread either invokes *notify()* for this object or *notifyAll()*.

3.2. *wait(long timeout)*

We can specify a timeout using this method, after which a thread will be woken up automatically. A thread can be woken up before reaching the timeout using *notify()* or *notifyAll()*.

Note that calling *(0)* is the same as calling *wait()*.

3.3. *wait(long timeout, int nanos)*

This is yet another signature providing the same functionality. The only difference here is that we can provide higher precision.

The total timeout period (in nanoseconds) is calculated as *1_000_000*timeout + nanos.*

We use the *notify()* method to wake up threads waiting for access to this object's monitor.

There are two ways of notifying waiting threads.

4.1. *notify()*

For all threads waiting on this object's monitor (by using any one of the *wait()* methods), the method *notify()* notifies any one of them to wake up arbitrarily. The choice of which thread to wake is nondeterministic and depends on implementation.

Since *notify()* wakes up a single random thread, we can use it to implement mutually exclusive locking where threads are doing similar tasks. But in most cases, it would be more viable to implement *notifyAll()*.

4.2. *notifyAll()*

This method simply wakes all threads waiting on this object's monitor.

The awakened threads will compete in the usual manner, like any other thread trying to synchronize on this object.

But before we allow their execution to continue, we should always **define a quick check for the condition required to proceed with the thread.** This is because there may be some situations where the thread woken up without receiving a notification (this scenario is discussed later in an example).

5. Sender-Receiver Synchronization Problem

Now that we understand the basics let's go through a simple *Sender–Receiver* application that will make use of the *wait()* and *notify()* methods to set up synchronization between them:

- The *Sender* is supposed to send a data packet to the Receiver.
- The *Receiver* can't process the data packet until the *Sender* finishes sending it.
- Similarly, the *Sender* shouldn't attempt to send another packet unless the *Receiver* has already processed the previous packet.

First, Let's create a Data class consisting of the data packet sent from the *Sender* to the *Receiver*. We'll use *wait()* and *notifyAll()* to set up synchronization between them:

```java
public class Data {
    private String packet;

    // True if receiver should
    // False if sender should
    private boolean transfer = true;

    public synchronized String receive() {
        while (transfer) {
            try {
                 ();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println(“Thread Interrupted”);
            }
        }
        transfer = true;

        String returnPacket = packet;
        notifyAll();
        return returnPacket;
    }

    public synchronized void send(String packet) {
        while (!transfer) {
            try {
                 ();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println(“Thread Interrupted”);
            }
        }
        transfer = false;

        this.packet = packet;
        notifyAll();
    }
}
```

Let's break down what's going on here:

- The packet variable denotes the data that's being transferred over the network.
- We have a boolean variable transfer, which the Sender and Receiver will use for synchronization:
- If this variable is true, the Receiver should for the Sender to send the message.
- If it's false, the Sender should for the Receiver to receive the message.
- The Sender uses the *send()* method to send data to the Receiver:
- If the transfer is false, we'll by calling *wait()* on this thread.
- But when it's true, we toggle the status, set our message, and call notifyAll() to wake up other threads to specify that a significant event has occurred, and they can check if they can continue execution.
- Similarly, the Receiver will use the *receive()* method:
- If the transfer was set to false by the Sender, only then will it proceed; otherwise, we'll call *wait()* on this thread.
- When the condition is met, we toggle the status, notify all waiting threads to wake up, and return the received data packet.

5.1. Why Enclose *wait()* in a while Loop?

Since *notify()* and *notifyAll()* randomly wake up threads waiting on this object's monitor, it's not always important to meet the condition. Sometimes, the thread is woken up, but the condition isn't actually satisfied yet.

We can also define a check to save us from spurious wakeups, where a thread can wake up from waiting without ever having received a notification.

5.2. Why Do We Need to Synchronize *send()* and *receive()* Methods?

We placed these methods inside *synchronized* methods to provide intrinsic locks. If a thread calling *wait()* method doesn't own the inherent lock, an error will be thrown.

We'll now create the *Sender* and *Receiver* and implement the *Runnable* interface on both so that their instances can be executed by a thread.

First, we'll see how *Sender* will work:

```
public class Sender implements Runnable {
    private Data data;

    // standard constructors

    public void run() {
        String packets[] = {
          "First packet",
          "Second packet",
          "Third packet",
          "Fourth packet",
          "End"
        };

        for (String packet : packets) {
            data.send(packet);

            // Thread.sleep() to mimic heavy server-side
processing
            try {
                Thread.sleep(ThreadLocalRandom.current().
nextInt(1000, 5000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread Interrupted");
            }
        }
    }
}
```

Let's take a closer look at this *Sender*:

- We're creating some random data packets that will be sent across the network in *packets[]* array.
- For each packet, we're merely calling *send()*.
- Then, we call *Thread.sleep()* with random intervals to mimic heavy server-side processing.

Finally, let's implement our *Receiver*:

```
public class Receiver implements Runnable {
    private Data load;

    // standard constructors

    public void run() {
        for(String receivedMessage = load.receive();
          !"End".equals(receivedMessage);
          receivedMessage = load.receive()) {

            System.out.println(receivedMessage);

            //Thread.sleep() to mimic heavy server-side processing
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(1000,
5000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread Interrupted");
            }
        }
    }
}
```

Here, we're simply calling *load.receive()* in the loop until we get the last "*End*" data packet. Let's now see this application in action:

```
public static void main(String[] args) {
    Data data = new Data();
    Thread sender = new Thread(new Sender(data));
    Thread receiver = new Thread(new Receiver(data));

    sender.start();
    receiver.start();
}
```

We'll receive the following output:

```
First packet
Second packet
Third packet
Fourth packet
```

And here we are. **We've received all data packets in the right, sequential order** and successfully established the correct communication between our sender and receiver.

6. Conclusion

In this chapter, we discussed some core synchronization concepts in Java. More specifically, we focused on using *wait()* and *notify()* to solve interesting synchronization problems. Finally, we went through a code sample where we applied these concepts in practice.

Before we close, it's worth mentioning that all these low-level APIs, such as *wait(), notify()* and *notifyAll()*, are traditional methods that work well. Still, higher-level mechanisms are often simpler and better, such as Java's native *Lock* and *Condition* interfaces (available in *java.util.concurrent.locks* package).

For more information on the *java.util.concurrent* package, visit our overview of the java.util.concurrent article. Lock and Condition are covered in the guide to java.util.concurrent.Locks.

The source code for the chapter is available over on GitHub.

4. The Thread.join() Method in Java

1. Overview

In this chapter, we'll discuss the different *join()* methods in the *Thread* class. We'll go into the details of these methods and some example codes.

Like the *wait()* and *notify()* methods, *join()* is another mechanism of inter-thread synchronization.

You can quickly look at this chapter to read more about *wait*() and *notify()*.

The join method is defined in the Thread class:

public final void join() throws InterruptedException
s for this thread to die.

When we invoke the *join()* method on a thread, the calling thread goes into a waiting state. It remains in a waiting state until the referenced thread terminates.

We can see this behaviour in the following code:

```
class SampleThread extends Thread {
    public int processingCount = 0;

    SampleThread(int processingCount) {
        this.processingCount = processingCount;
        LOGGER.info("Thread Created");
    }

    @Override
    public void run() {
        LOGGER.info("Thread " + this.getName() + " started");
        while (processingCount > 0) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                LOGGER.info("Thread " + this.getName() + "
interrupted");
            }
            processingCount--;
            LOGGER.info("Inside Thread " + this.getName() + ",
processingCount = " + processingCount);
        }
        LOGGER.info("Thread " + this.getName() + " exiting");
    }
}

@Test
public void givenStartedThread_whenJoinCalled_sTillCompletion()
  throws InterruptedException {
    Thread t2 = new SampleThread(1);
    t2.start();
    LOGGER.info("Invoking join");
    t2.join();
    LOGGER.info("Returned from join");
    assertFalse(t2.isAlive());
}
```

We should expect results similar to the following when executing the code:

```
[main] INFO: Thread Thread-1 Created
[main] INFO: Invoking join
[Thread-1] INFO: Thread Thread-1 started
[Thread-1] INFO: Inside Thread Thread-1, processingCount = 0
[Thread-1] INFO: Thread Thread-1 exiting
[main] INFO: Returned from join
```

The *join()* method may also return if the referenced thread is interrupted. In this case, the method throws an *InterruptedException.*

Finally, **if the referenced thread has already been terminated or hasn't been started, the call to *join()* method returns immediately.**

```
Thread t1 = new SampleThread(0);
t1.join();  //returns immediately
```

The *join()* method will keep waiting if the referenced thread is blocked or takes too long to process. This can become an issue, as the calling thread will become non-responsive. To handle these situations, we'll use overloaded versions of the *join()* method that allows us to specify a timeout period.

There are two timed versions that overload the *join()* method:

"public final void join(long millis) throws InterruptedException
s at most millis milliseconds for this thread to die. A timeout of 0 means to forever."

"public final void join(long millis,intnanos) throws InterruptedException
s at most millis milliseconds plus nanos nanoseconds for this thread to die."

We can use the timed *join()* as below:

```
@Test
public void givenStartedThread_whenTimedJoinCalled_
sUntilTimedout()
  throws InterruptedException {
    Thread t3 = new SampleThread(10);
    t3.start();
    t3.join(1000);
    assertTrue(t3.isAlive());
}
```

In this case, the calling thread s roughly 1 second for the thread t3 to finish. If the thread t3 doesn't finish in this period, the join() method returns control to the calling method.

Timed *join()* is dependent on the OS for timing. So, we can't assume that *join()* will exactly as long as specified.

In addition to waiting until termination, calling the *join()* method has a synchronization effect. ***join()* creates a happens-before relationship**:

"All actions in a thread happen-before any other thread successfully returns from a join() on that thread."

This means that when a thread t1 calls t2.join(), all changes done by t2 are visible in t1 on return. However, if we don't invoke *join()* or use other synchronization mechanisms, then we don't have any guarantee that changes in the other thread will be visible to the current thread, even if the other thread has been completed.

Therefore, even though the *join()* method calls to a thread in the terminated state returns immediately, we still need to call it in some situations.

We can see an example of improperly synchronized code below:

```
SampleThread t4 = new SampleThread(10);
t4.start();
// not guaranteed to stop even if t4 finishes.
do {

} while (t4.processingCount > 0);
```

To properly synchronize the above code, we can add timed *t4.join()* inside the loop or use another synchronization mechanism.

5. Conclusion

The *join()* method is quite useful for inter-thread synchronization. In this chapter, we discussed the different *join()* methods and their behaviour. We also reviewed the code using the *join()* method.

As always, the full source code can be found on GitHub.

5. Guide to the Synchronized Keyword in Java

1. Overview

In this chapter, we'll learn how to use the synchronized block in Java.

Simply put, in a multi-threaded environment, a race condition occurs when two or more threads attempt to update mutable shared data at the same time. Java offers a mechanism to avoid race conditions by synchronizing thread access to shared data.

A piece of logic marked with *synchronized* becomes a synchronized block, **allowing only one thread to execute at any given time.**

Let's consider a typical race condition where we calculate the sum, and multiple threads execute the *calculate()* method:

```
public class SynchronizedMethods {

    private int sum = 0;

    public void calculate() {
        setSum(getSum() + 1);
    }

    // standard setters and getters
}
```

Then, let's write a simple test:

```
@Test
public void givenMultiThread_whenNonSyncMethod() {
    ExecutorService service = Executors.newFixedThreadPool(3);
    SynchronizedMethods summation = new SynchronizedMethods();

    IntStream.range(0, 1000)
      .forEach(count -> service.submit(summation::calculate));
    service.aTermination(1000, TimeUnit.MILLISECONDS);

    assertEquals(1000, summation.getSum());
}
```

We're using an *ExecutorService* with a 3-threads pool to execute the *calculate()* 1000 times.

If we executed this serially, the expected output would be 1000, but **our multi-threaded execution fails almost every time** with an inconsistent actual output:

```
java.lang.AssertionError: expected:<1000> but was:<965>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotEquals(Assert.java:834)
...
```

Of course, we don't find this result unexpected.

A simple way to avoid the race condition is to make the operation thread-safe using the *synchronized* keyword.

We can use the *synchronized* keyword on different levels:

- Instance methods
- Static methods
- Code blocks

When we use a *synchronized* block, Java internally uses a monitor, also known as a monitor lock or intrinsic lock, to provide synchronization. These monitors are bound to an object; therefore, all synchronized blocks of the same object can have only one thread executing them at the same time.

3.1. *Synchronized* Instance Methods

We can add the *synchronized* keyword in the method declaration to make the method synchronized:

```java
public synchronized void synchronisedCalculate() {
    setSum(getSum() + 1);
}
```

Notice that once we synchronize the method, the test case passes with the actual output as 1000:

```java
@Test
public void givenMultiThread_whenMethodSync() {
    ExecutorService service = Executors.newFixedThreadPool(3);
    SynchronizedMethods method = new SynchronizedMethods();

    IntStream.range(0, 1000)
      .forEach(count -> service.
submit(method::synchronisedCalculate));
    service.aTermination(1000, TimeUnit.MILLISECONDS);

    assertEquals(1000, method.getSum());
}
```

Instance methods are *synchronized* over the instance of the class owning the method, which means only one thread per instance of the class can execute this method.

3.2. *Synchronized* Static Methods

Static methods are synchronized just like instance methods:

```
public static synchronized void syncStaticCalculate() {
     staticSum = staticSum + 1;
 }
```

These methods are *synchronized* on the *Class* object associated with the class. Since only one *Class* object exists per JVM per class, only one thread can execute inside a *static synchronized* method per class, irrespective of the number of instances it has.

Let's test it:

```
@Test
public void givenMultiThread_whenStaticSyncMethod() {
    ExecutorService service = Executors.newCachedThreadPool();

    IntStream.range(0, 1000)
      .forEach(count ->
        service.submit(SynchronizedMethods::syncStaticCalculate));
    service.aTermination(100, TimeUnit.MILLISECONDS);

    assertEquals(1000, SynchronizedMethods.staticSum);
}
```

3.3. *Synchronized* Blocks Within Methods

Sometimes, we don't want to synchronize the entire method, only some instructions within it. We can achieve this by *applying* synchronized to a block:

```
public void performSynchronisedTask() {
    synchronized (this) {
        setCount(getCount()+1);
    }
}
```

Then we can test the change:

```
@Test
public void givenMultiThread_whenBlockSync() {
    ExecutorService service = Executors.newFixedThreadPool(3);
    SynchronizedBlocks synchronizedBlocks = new
SynchronizedBlocks();

    IntStream.range(0, 1000)
      .forEach(count ->
        service.
submit(synchronizedBlocks::performSynchronisedTask));
    service.aTermination(100, TimeUnit.MILLISECONDS);

    assertEquals(1000, synchronizedBlocks.getCount());
}
```

Notice that we passed the parameter this to the synchronized block. This is the monitor object. The code inside the block gets synchronized on the monitor object. Simply put, only one thread per monitor object can execute inside that code block.

If the method was static, we would pass the class name in place of the object reference, and the class would be a monitor for synchronization of the block:

```
public static void performStaticSyncTask(){
    synchronized (SynchronisedBlocks.class) {
        setStaticCount(getStaticCount() + 1);
    }
}
```

Let’s test the block inside the *static* method:

```
@Test
public void givenMultiThread_whenStaticSyncBlock() {
    ExecutorService service = Executors.newCachedThreadPool();

    IntStream.range(0, 1000)
      .forEach(count ->
        service.
submit(SynchronizedBlocks::performStaticSyncTask));
    service.aTermination(100, TimeUnit.MILLISECONDS);

    assertEquals(1000, SynchronizedBlocks.getStaticCount());
}
```

3.4. Reentrancy

The lock behind the *synchronized* methods and blocks is a reentrant. This means the current thread can acquire the same *synchronized* lock over and over again while holding it:

```
Object lock = new Object();
synchronized (lock) {
    System.out.println("First time acquiring it");

    synchronized (lock) {
        System.out.println("Entering again");

         synchronized (lock) {
             System.out.println("And again");
         }
    }
}
```

As shown above, while in a *synchronized* block, we can repeatedly acquire the same monitor lock.

4. Conclusion

In this brief chapter, we explored different ways of using the synchronized keyword to achieve thread synchronization.

We also learned how a race condition can impact our application and how synchronization helps us avoid that. For more about thread safety using locks in Java, refer to our *java.util.concurrent.Locks* article.

The complete code for this chapter is available over on GitHub.

6. Guide to the Volatile Keyword in Java

1. Overview

Without necessary synchronizations, the compiler, runtime, or processors may apply all sorts of optimizations. Even though these optimizations are usually beneficial, sometimes they can cause subtle issues.

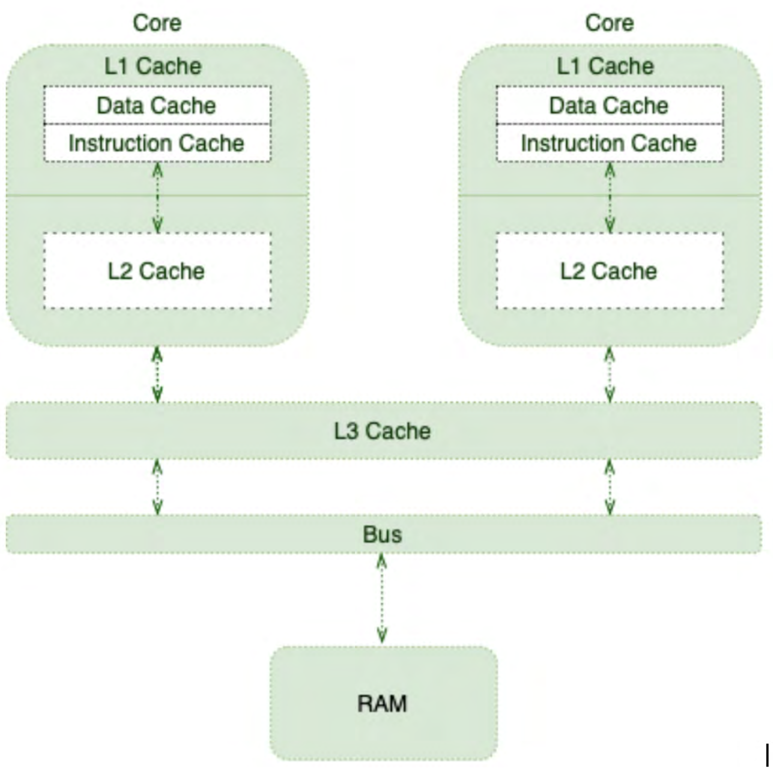
Caching and reordering are optimizations that may surprise us in concurrent contexts. Java and the JVM provide many ways to control memory order, and the volatile keyword is one of them.

In this chapter, we'll focus on Java's foundational, but often misunderstood, concept, the volatile keyword. First, we'll start with some background of how the underlying computer architecture works, and then we'll familiarize ourselves with memory order in Java. Finally, we'll discuss the challenges of concurrency in multiprocessor shared architecture, and how volatile helps fix them.

Processors are responsible for executing program instructions. Therefore, they must retrieve the program instructions and required data from RAM.

As CPUs can carry many instructions per second, fetching from RAM isn't ideal. To improve this situation, processors use tricks like Out of Order Execution, Branch Prediction, Speculative Execution, and Caching.

This is where the following memory hierarchy comes into play:



As different cores execute more instructions and manipulate more data, they fill their caches with more relevant data and instructions. **This will improve the overall performance, at the expense of introducing cache coherence challenges.**

We should think twice about what happens when one thread updates a cached value.

To expand more on cache coherence, we'll borrow an example from the book Java Concurrency in Practice:

```java
public class TaskRunner {

    private static int number;
    private static boolean ready;

    private static class Reader extends Thread {

        @Override
        public void run() {
            while (!ready) {
                Thread.yield();
            }

            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new Reader().start();
        number = 42;
        ready = true;
    }
}
```

The *TaskRunner* class maintains two simple variables. Its main method creates another thread that spins on the *ready* variable as long as it's *false*. When the variable becomes *true*, the thread prints the *number* variable.

The specific function of *Thread.yield()* is to allocate the current thread's resources to other threads. In simple terms, it just delays *Reader* thread's execution. Basically, *Reader* "lets" other threads use its "future" resources.

When "ready=true" is finally written in the cache, the *Reader* thread will execute and "get back" its resources.

Many may expect this program to print 42 after a short delay; however, the delay may be much longer. It may even hang forever or print zero.

The cause of these anomalies is the lack of proper memory visibility and reordering. Let's evaluate them in more detail.

3.1. Memory Visibility

This simple example has two application threads: the main and reader threads. Let's imagine a scenario in which the OS schedules those threads on two different CPU cores, where:

- The main thread has its copy of ready and number variables in its core cache.
- The reader thread ends up with its copies too.
- The main thread updates the cached values.

Most modern processors write requests that won't be applied right after they're issued. Processors **tend to queue those writes in a special write buffer**. After a while, they'll apply those writes to the main memory all at once.

With all that being said, **when the main thread updates the number and *ready* variables, there's no guarantee about what the reader thread may see. In other words, the reader thread may see the updated value immediately, with some delay, or never at all**.

This memory visibility may cause liveness issues in programs relying on visibility.

3.2. Reordering

To make matters even worse, **the reader thread may see those writes in an order other than the actual program order**. For instance, since we first update the *number* variable:

```java
public static void main(String[] args) {
    new Reader().start();
    number = 42;
    ready = true;
}
```

We may expect the reader thread to print 42, **but it's actually possible to see zero as the printed value**.

Reordering is an optimization technique for performance improvements. Interestingly, different components may apply this optimization:

- The processor may flush its write buffer in an order other than the program order.
- The processor may apply an out-of-order execution technique.
- The JIT compiler may optimize via reordering.

We can use *volatile* to tackle the issues with Cache Coherence.

To ensure that updates to variables propagate predictably to other threads, we should apply the *volatile* modifier to those variables.

This way, we can communicate with runtime and processor to not reorder any instruction involving the volatile variable. Also, processors understand that they should immediately flush any updates to these variables:

```
public class TaskRunner {

    private volatile static int number;
    private volatile static boolean ready;

    // same as before
}
```

For multithreaded applications, we need to ensure a couple of rules for consistent behavior:

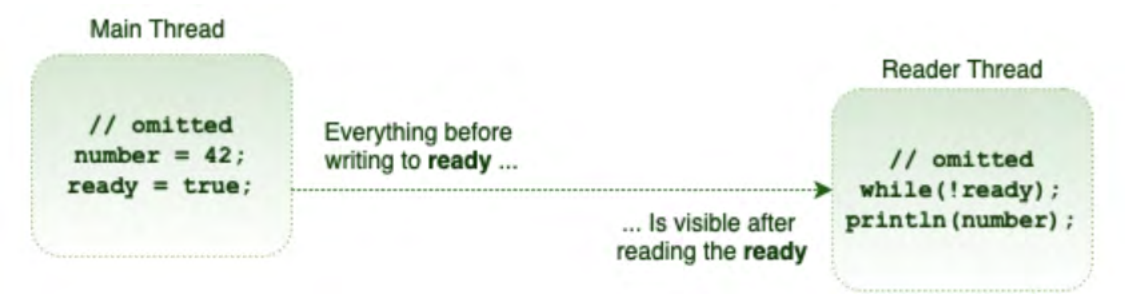
Mutual Exclusion – only one thread executes a critical section at a time
Visibility – changes made by one thread to the shared data are visible to other threads to maintain data consistency

synchronized methods and blocks provide both of the above properties at the cost of application performance.

volatile is quite a useful keyword because it **can help ensure the visibility aspect of the data change without providing mutual exclusion**. Thus, it's useful where we're ok with multiple threads executing a block of code in parallel, but we need to ensure the visibility property.

The memory visibility effects of *volatile* variables extend beyond the *volatile* variables themselves.

To make matters more concrete, suppose thread A writes to a *volatile* variable, and then thread B reads the same volatile variable. In such cases, **the values that were visible to A before writing the volatile variable will be visible to B after reading the volatile variable:**



Technically, any write to a *volatile* field happens-before every subsequent read of the same field. This is the volatile variable rule of the Java Memory Model (JMM).

6.1. Piggybacking

Because of the strength of the happens-before memory ordering, sometimes we can piggyback on the visibility properties of another volatile variable. For instance, in our particular example, we just need to mark the ready variable as volatile:

```
public class TaskRunner {

    private static int number; // not volatile
    private volatile static boolean ready;

    // same as before
}
```

Anything prior to writing *true* to the *ready* variable is visible to anything after reading the *ready* variable. Therefore, the *number* variable piggybacks on the memory visibility enforced by the ready variable. Simply put, **even though it's not a volatile variable, it's exhibiting a volatile behavior.**

Using these semantics, we can define only a few variables in our class as volatile and optimize the visibility guarantee.

7. Conclusion

In this chapter, we explored the volatile keyword, including its capabilities and the improvements made to it starting with Java 5.

As always, the code examples can be found on GitHub.

7. A Guide to the Java ExecutorService

1. Overview

ExecutorService is a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, *ExecutorService* automatically provides a pool of threads and an API for assigning tasks to it.

2.1. Factory Methods of the *Executors Class*

The easiest way to create *ExecutorService* is to use one of the factory methods of the *Executors* class.

For example, the following line of code will create a thread pool with 10 threads:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

There are several other factory methods to create a predefined *ExecutorService* that meets specific use cases. To find the best method for your needs, consult Oracle's official documentation.

2.2. Directly Create an *ExecutorService*

Because *ExecutorService* is an interface, an instance of any of its implementations can be used. There are several implementations to choose from in the java.util.concurrent package, or you can create your own.

For example, the *ThreadPoolExecutor* class has a few constructors that we can use to configure an executor service and its internal pool:

```
ExecutorService executorService =
  new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
  new LinkedBlockingQueue<Runnable>());
```

You may notice that the code above is very similar to the source code of the factory method *newSingleThreadExecutor()*. For most cases, a detailed manual configuration isn't necessary.

ExecutorService can execute *Runnable* and *Callable* tasks. To keep things simple in this chapter, two primitive tasks will be used. Notice that we use lambda expressions here instead of anonymous inner classes:

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};

List<Callable<String>> callableTasks = new ArrayList<>();
callableTasks.add(callableTask);
callableTasks.add(callableTask);
callableTasks.add(callableTask);
```

We can assign tasks to the *ExecutorService* using several methods, including *execute()*, which is inherited from the *Executor* interface, and also *submit()*, *invokeAny()*, and *invokeAll()*.

The ***execute()*** method is *void* and doesn't give any possibility to get the result of a task's execution or to check the task's status (is it running):

```
executorService.execute(runnableTask);
```

submit() submits a *Callable* or *Runnable* task to an *ExecutorService* and returns a result of type *Future*:

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

invokeAny() assigns a collection of tasks to an *ExecutorService*, causing each to run, and returns the result of successful execution of one task (if there was a successful execution):

```
String result = executorService.invokeAny(callableTasks);
```

invokeAll() assigns a collection of tasks to an *ExecutorService*, causing each to run, and returns the result of all task executions in the form of a list of objects of type Future:

```
List<Future<String>> futures = executorService.
invokeAll(callableTasks);
```

Before going further, we need to discuss two more items: shutting down an *ExecutorService* and dealing with *Future* return types.

In general, the *ExecutorService* won't be automatically destroyed when there's no task to process. It will stay alive and for new work to be done.

In some cases, this is very helpful, such as when an app needs to process tasks that appear on an irregular basis or the task quantity isn't known at compile time.

On the other hand, an app could reach its end but not be stopped because a waiting *ExecutorService* will cause the JVM to keep running.

To properly shut down an *ExecutorService*, we have the *shutdown()* and *shutdownNow()* APIs.

The ***shutdown()*** method doesn't cause immediate destruction of the *ExecutorService*. It will make the *ExecutorService* stop accepting new tasks and shut down after all running threads finish their current work:

```
String result = executorService.invokeAny(callableTasks);
```

The s***hutdownNow()*** method tries to destroy the *ExecutorService* immediately, but it doesn't guarantee that all the running threads will be stopped at the same time:

```
List<Runnable> notExecutedTasks = executorService.shutDownNow();
```

This method returns a list of tasks that are waiting to be processed. It's up to the developer to decide what to do with these tasks.

One good way to shut down the *ExecutorService* (which is also recommended by Oracle) is to use both of these methods combined with the ***aTermination()*** method:

```
executorService.shutdown();
try {
    if (!executorService.aTermination(800, TimeUnit.MILLISECONDS))
{
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
}
```

With this approach, the *ExecutorService* will first stop taking new tasks and then up to a specified time for all tasks to be completed. If that time expires, the execution is stopped immediately.

The *submit()* and *invokeAll()* methods return an object or a collection of objects of type *Future*, which allows us to get the result of a task's execution or to check the task's status (is it running).

The *Future* interface provides a special blocking method, *get()*, which returns an actual result of the *Callable* task's execution, or *null* in the case of a *Runnable* task:

```java
Future<String> future = executorService.submit(callableTask);
String result = null;
try {
    result = future.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

Calling the *get()* method while the task is still running will cause execution to block until the task is properly executed and the result is available.

An application's performance can degrade with very long blocking caused by the *get()* method. If the resulting data isn't crucial, it's possible to avoid such a problem by using timeouts:

```java
String result = future.get(200, TimeUnit.MILLISECONDS);
```

If the execution period is longer than specified (in this case, 200 milliseconds), a *TimeoutException* will be thrown.
We can use the *isDone()* method to check whether the assigned task is already processed.

The *Future* interface also allows for cancelling task execution with the *cancel()* method and checking the cancellation with the *isCancelled()* method:

```java
boolean canceled = future.cancel(true);
boolean isCancelled = future.isCancelled();
```

The *ScheduledExecutorService* runs tasks after some predefined delay and/or periodically.

Once again, the best way to instantiate a *ScheduledExecutorService* is to use the factory methods of the *Executors* class.

For this section, we'll use a *ScheduledExecutorService* with one thread:

```
ScheduledExecutorService executorService = Executors
  .newSingleThreadScheduledExecutor();
```

To schedule a single task's execution after a fixed delay, we can use the *scheduled()* method of the *ScheduledExecutorService*.

Two *scheduled()* methods allow us to execute *Runnable* or *Callable* tasks:

```
Future<String> resultFuture =
  executorService.schedule(callableTask, 1, TimeUnit.SECONDS);
```

The *scheduleAtFixedRate()* method lets us run a task periodically after a fixed delay. The code above delays for one second before executing *callableTask*.

The following block of code will run a task after an initial delay of 100 milliseconds; after that, it will run the same task every 450 milliseconds:

```
executorService.scheduleAtFixedRate(runnableTask, 100, 450,
TimeUnit.MILLISECONDS);
```

If the processor needs more time to run an assigned task than the period parameter of the *scheduleAtFixedRate()* method, the *ScheduledExecutorService* will until the current task is completed before starting the next.

If it's necessary to have a fixed length delay between iterations of the task, *scheduleWithFixedDelay()* should be used.

For example, the following code will guarantee a 150-millisecond pause between the end of the current execution and the start of another one:

```
executorService.scheduleWithFixedDelay(task, 100, 150, TimeUnit.
MILLISECONDS);
```

According to the *scheduleAtFixedRate()* and *scheduleWithFixedDelay()* method contracts, the period execution of the task will end at the termination of the *ExecutorService* or if an exception is thrown during task execution.

After the release of Java 7, many developers decided to replace the *ExecutorService* framework with the fork/join framework.

However, this isn't always the right decision. Despite the simplicity and frequent performance gains associated with fork/join, it reduces developer control over concurrent execution.

ExecutorService allows the developer to control the number of generated threads and the granularity of tasks that separate threads should run. The best use case for *ExecutorService* is processing independent tasks, such as transactions or requests, according to the scheme "one thread for one task."

In contrast, according to Oracle's documentation, fork/join was designed to recursively speed up work that can be broken into smaller pieces.

8. Conclusion

Despite the relative simplicity of *ExecutorService*, there are a few common pitfalls.

Let's summarize them:

Keeping an unused *ExecutorService* alive: See the detailed explanation in Section 4 on how to shut down an *ExecutorService*.

Wrong thread-pool capacity while using fixed length thread pool: It's very important to determine how many threads the application will need to run tasks efficiently. A thread pool that is too large will cause unnecessary overhead just to create threads that will mostly be in the waiting mode. Too few can make an application seem unresponsive because of long waiting periods for tasks in the queue.

Calling a *Future's get()* method after task cancellation: Attempting to get the result of an already canceled task triggers a *CancellationException*.

Unexpectedly long blocking with *Future's get()* method: We should use timeouts to avoid unexpected s.

As always, the code for this chapter is available in the GitHub repository.

8. Guide To CompletableFuture

1. Introduction

This chapter is a guide to the functionality and use cases of the *CompletableFuture* class that was introduced as a Java 8 Concurrency API improvement.

Asynchronous computation can be difficult to understand. Usually, we want to think of any computation as a series of steps. Still, in the case of asynchronous computation, **actions represented as callbacks tend to be either scattered across the code or deeply nested inside each other.** Things get even more complicated when we need to handle errors that might occur during one of the steps.

The *Future* interface was added in Java 5 to serve as a result of an asynchronous computation, but it didn't have any methods to combine these computations or handle possible errors.

Java 8 introduced the *CompletableFuture* class. Along with the *Future* interface, it also implemented the *CompletionStage* interface. This interface defines the contract for an asynchronous computation step that we can combine with other steps.

CompletableFuture is simultaneously both a building block and a framework, with about 50 different methods for composing, combining, and executing asynchronous computation steps and **handling errors**.
Such a large API can be overwhelming, but these mostly fall into several clear and distinct use cases.

First, the *CompletableFuture* class implements the *Future* interface to **use it as a *Future* implementation, but with additional completion logic.**

For example, we can create an instance of this class with a no-arg constructor to represent some future result, hand it out to the consumers, and complete it at some time in the future using the *complete* method. The consumers may use the *get* method to block the current thread until this result is provided.

In the example below, we have a method that creates a *CompletableFuture* instance, then spins off some computation in another thread and returns the *Future* immediately.

When the computation is done, the method completes the *Future* by providing the result to the *complete* method:

```
public Future<String> calculateAsync() throws InterruptedException
{
            CompletableFuture<String>   completableFuture   =   new
CompletableFuture<>();

    Executors.newCachedThreadPool().submit(() -> {
        Thread.sleep(500);
        completableFuture.complete("Hello");
        return null;
    });

    return completableFuture;
}
```

To spin off the computation, we use the *Executor* API. This method of creating and completing a *CompletableFuture* can be used with any concurrency mechanism or API, including raw threads.

Notice that **the *calculateAsync* method returns a *Future* instance.**

We simply call the method, receive the *Future* instance, and call the *get* method on it when we're ready to block for the result.

Also, we can observe that the *get* method throws some checked exceptions, namely *ExecutionException* (encapsulating an exception that occurred during a computation) and *InterruptedException* (an exception signifying that a thread was interrupted either before or during an activity):

```
Future<String> completableFuture = calculateAsync();

// ...

String result = completableFuture.get();
assertEquals("Hello", result);
```

If we already know the result of a computation, we can use the static *completedFuture* method with an argument that represents the result of this computation. Consequently, the *get* method of the *Future* will never block, immediately returning this result instead:

```
Future<String> completableFuture =
  CompletableFuture.completedFuture("Hello");

// ...

String result = completableFuture.get();
assertEquals("Hello", result);
```

As an alternative scenario, we may want to **cancel the execution of a *Future*.**

The code above allows us to pick any mechanism of concurrent execution, but what if we want to skip this boilerplate and execute some code asynchronously?

Static methods *runAsync* and *supplyAsync* allow us to create a *CompletableFuture* instance out of *Runnable* and *Supplier* functional types correspondingly.

Runnable and *Supplier* are functional interfaces that allow passing their instances as lambda expressions thanks to the new Java 8 feature.

The *Runnable* interface is the same old interface used in threads and doesn't allow us to return a value.

The *Supplier* interface is a generic functional interface with a single method with no arguments and returns a value of a parameterized type.

This allows us to **provide an instance of the *Supplier* as a lambda expression that does the calculation and returns the result**. It's as simple as:

```
CompletableFuture<String> future
  = CompletableFuture.supplyAsync(() -> "Hello");

// ...

assertEquals("Hello", future.get());
```

The most generic way to process the result of a computation is to feed it to a function. The *thenApply* method does exactly that; it accepts a *Function* instance, uses it to process the result, and returns a *Future* that holds a value returned by a function:

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future = completableFuture
  .thenApply(s -> s + " World");

assertEquals("Hello World", future.get());
```

If we don't need to return a value down the *Future* chain, we can use an instance of the *Consumer* functional interface. Its single method takes a parameter and returns void.

There's a method for this use case in the *CompletableFuture*. The *thenAccept* method receives a *Consumer* and passes it the result of the computation. Then the final *future.get()* call returns an instance of the *Void* type:

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<Void> future = completableFuture
   .thenAccept(s -> System.out.println("Computation returned: " +
s));

future.get();
```

Finally, if we neither need the value of the computation nor want to return some value at the end of the chain, then we can pass a *Runnable* lambda to the *thenRun* method. In the following example, we simply print a line in the console after calling the *future.get()*:

```java
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> “Hello”);

CompletableFuture<Void> future = completableFuture
  .thenRun(() -> System.out.println(“Computation finished.”));

future.get();
```

The best part of the *CompletableFuture* API is the **ability to combine *CompletableFuture* instances in a chain of computation steps.**
The result of this chaining is itself a CompletableFuture that allows further chaining and combining. This approach is ubiquitous in functional languages and is often called a monadic design pattern.

In the following example, we use the *thenCompose* method to chain two *Futures* sequentially.

Notice that this method takes a function that returns a *CompletableFuture* instance. The argument of this function is the result of the previous computation step. This allows us to use this value inside the next *CompletableFuture's* lambda:

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello")
      .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + "
World"));

assertEquals("Hello World", completableFuture.get());
```

The *thenCompose* method, together with *thenApply*, implements the basic building blocks of the monadic pattern. They closely relate to the *map* and flatMap methods of the *Stream* and *Optional* classes, also available in Java 8.

Both methods receive a function and apply it to the computation result, but the *thenCompose (flatMap)* method **receives a function that returns another object of the same type**. This functional structure allows us to compose the instances of these classes as building blocks.

If we want to execute two independent *Futures* and do something with their results, we can use the *thenCombine* method that accepts a *Future* and a *Function* with two arguments to process both results:

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello")
    .thenCombine(CompletableFuture.supplyAsync(
      () -> " World"), (s1, s2) -> s1 + s2));

assertEquals("Hello World", completableFuture.get());
```

A simpler case is when we want to do something with two *Futures*' results but don't need to pass any resulting value down a *Future* chain. The *thenAcceptBoth* method is there to help:

```
CompletableFuture future = CompletableFuture.supplyAsync(() ->
"Hello")
  .thenAcceptBoth(CompletableFuture.supplyAsync(() -> " World"),
    (s1, s2) -> System.out.println(s1 + s2));
```

In our previous sections, we've shown examples of *thenApply()* and *thenCompose()*. Both APIs help chain different *CompletableFuture* calls, but the usage of these two functions is different.

7.1. *thenApply()*

We can use this method to work with the result of the previous call. However, a key point to remember is that the return type will be a combination of all the calls.

So, this method is useful when we want to transform the result of a *CompletableFuture* call:

```
CompletableFuture<Integer> finalResult = compute().thenApply(s-> s
+ 1);
```

7.2. *thenCompose()*

The *thenCompose()* is similar to *thenApply()* in that both return a new CompletionStage. However, ***thenCompose()* uses the previous stage as the argument.** It will flatten and return a *Future* with the result directly, rather than a nested future as we observed in *thenApply()*:

```
CompletableFuture<Integer> computeAnother(Integer i){
    return CompletableFuture.supplyAsync(() -> 10 + i);
}
CompletableFuture<Integer> finalResult = compute().
thenCompose(this::computeAnother);
```

So if the idea is to chain *CompletableFuture* methods, then it's better to use *thenCompose()*.
Also, note that the difference between these two methods is analogous to the difference between *map()* and *flatMap()*.

When we need to execute multiple *Futures* in parallel, we usually want to for all of them to execute and then process their combined results.

The *CompletableFuture.allOf* static method allows us to for the completion of all the *Futures* provided as a var-arg:

```java
CompletableFuture<String> future1
  = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future2
  = CompletableFuture.supplyAsync(() -> "Beautiful");
CompletableFuture<String> future3
  = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<Void> combinedFuture
  = CompletableFuture.allOf(future1, future2, future3);
// ...

combinedFuture.get();

assertTrue(future1.isDone());
assertTrue(future2.isDone());
assertTrue(future3.isDone());
```

Notice that the return type of the *CompletableFuture.allOf()* is a *CompletableFuture<Void>*. The limitation of this method is that it doesn't return the combined results of all *Futures*. Instead, we have to get results from *Futures* manually. Fortunately, the *CompletableFuture.join()* method and Java 8 Streams API make it simple:

```java
String combined = Stream.of(future1, future2, future3)
  .map(CompletableFuture::join)
  .collect(Collectors.joining(" "));

assertEquals("Hello Beautiful World", combined);
```

The *CompletableFuture.join()* method is similar to the *get* method but throws an unchecked exception in case the *Future* doesn't complete normally. This makes it possible to use it as a method reference in the *Stream.map()* method.

For error handling in a chain of asynchronous computation steps, we have to adapt the *throw/catch* idiom in a similar fashion.

Instead of catching an exception in a syntactic block, the *CompletableFuture* class allows us to *handle* it in a special *handle* method. This method receives two parameters: a result of a computation (if it finished successfully) and the exception thrown (if some computation step wasn't completed normally).

In the following example, we'll use the *handle* method to provide a default value when the asynchronous computation of a greeting was finished with an error because no name was provided:

```
String name = null;

// ...

CompletableFuture<String> completableFuture
  =  CompletableFuture.supplyAsync(() -> {
      if (name == null) {
          throw new RuntimeException("Computation error!");
      }
      return "Hello, " + name;
  }).handle((s, t) -> s != null ? s : "Hello, Stranger!");

assertEquals("Hello, Stranger!", completableFuture.get());
```

As an alternative scenario, suppose we want to manually complete the *Future* with a value, as in the first example, but can also complete it with an exception.

The *completeExceptionally* method is intended for just that. The *completableFuture.get()* method in the following example throws an *ExecutionException* with a *RuntimeException* as its cause:

```java
CompletableFuture<String> completableFuture = new
CompletableFuture<>();

// ...

completableFuture.completeExceptionally(
  new RuntimeException(“Calculation failed!”));

// ...

completableFuture.get(); // ExecutionException
```

In the example above, we could have handled the exception asynchronously with the handle method, but with the *get* method, we can use the more typical approach of synchronous exception processing.

Most methods of the fluent API in the *CompletableFuture* class have two additional variants with the *Async* postfix. These methods are usually intended for **running a corresponding execution step in another thread**.

The methods without the *Async* postfix run the next execution stage using a calling thread. In contrast, the *Async* method without the *Executor* argument runs a step using the common *fork/join* pool implementation of *Executor*, accessed with the *ForkJoinPool.commonPool()*, as long as parallelism > 1.

Finally, the *Async* method with an *Executor* argument runs a step using the passed *Executor*.

Here's a modified example that processes the result of a computation with a *Function* instance. The only visible difference is the *thenApplyAsync* method, but under the hood, the application of a function is wrapped into a *ForkJoinTask* instance (for more information on the fork/join framework, see the article "Guide to the *Fork/Join* Framework in Java"). This allows us to parallelize our computation even more and use system resources more efficiently:

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future = completableFuture
  .thenApplyAsync(s -> s + " World");

assertEquals("Hello World", future.get());
```

Java 9 enhances the CompletableFuture API with the following changes:

- New factory methods added
- Support for delays and timeouts
- Improved support for subclassing

Also included are new instance APIs:

- *Executor defaultExecutor()*
- *CompletableFuture<U> newIncompleteFuture()*
- *CompletableFuture<T> copy()*
- *CompletionStage<T> minimalCompletionStage()*
- *CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)*
- *CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)*
- *CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)*
- *CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)*

We also now have a few static utility methods:

- *Executor delayedExecutor(long delay, TimeUnit unit, Executor executor)*
- *Executor delayedExecutor(long delay, TimeUnit unit)*
- *<U> CompletionStage<U> completedStage(U value)*
- *<U> CompletionStage<U> failedStage(Throwable ex)*
- *<U> CompletableFuture<U> failedFuture(Throwable ex)*

Finally, to address timeout, Java 9 has introduced two more new functions:

- *orTimeout()*
- *completeOnTimeout()*

Here's the detailed article for further reading: Java 9 CompletableFuture API Improvements.

12. Conclusion

In this chapter, we described the methods and typical use cases of the *CompletableFuture* class.

The source code for the chapter is available over on GitHub.