- CompletionService Interface
- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview

Phaser

Comprehensive guide with executable examples to using Phaser, an advanced and sophisticated synchronization barrier construct.

Overview

The `Phaser` class is an extension of the functionality offered by `CyclicBarrier` and `CountDownLatch` classes and is more flexible in use. One stark difference is that the `Phaser` class allows the number of registered parties that synchronize on a phaser to vary over time. The `Phaser` can be repeatedly awaited similar to a `CyclicBarrier`.

Example

Apart from specifying the number of threads/tasks to synchronize in the constructor, threads/tasks can also register with an instance of `Phaser` using the `register()` or the `bulkRegister(int)` methods. Note, that if a thread `register()`-s with an instance of `Phaser` there's no way for the thread to query the instance to determine if it registered with the instance, i.e. there's no internal book-keeping maintained by the `Phaser` instance. However, if such behavior is desired the `Phaser` class can be subclassed and the book-keeping functionality added.

The program below exercises some of the APIs exposed by `Phaser` to register threads with the barrier. Run the program and study the comments before we discuss them.

Java

main.java

Saved

```java
import java.util.concurrent.*;

class Demostration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of Phaser class and register only a single that will arrive
        // at the barrier
        Phaser phaser = new Phaser(1);

        try {
```

Run

Notice that the main thread is responsible for registering 3 parties with the `Phaser` instance after the instance has been constructed but arrives at the barrier only once, i.e. it is not necessary that the thread that invokes `register()` must also be the same thread that arrives at the barrier.

Arriving and Deregistering

Consider a scenario where we want all the spawned threads/tasks to wait until the main thread has finished initialization or performed some tasks before we want the spawned threads to proceed. We could initialize the `Phaser` with a count of one more than the number of threads we plan to spawn, and then have the main thread do the required work. Finally, the main thread arrives at and deregisters with the barrier at the same time. This releases the spawned threads that have already been waiting at the barrier and reduces the number of parties required to synchronize at the barrier by one for future. The described example appears in the program below.

Java

main.java

Saved

```java
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
```

```
    ExecutorService executorService = Executors.newFixedThreadPool(15);

    // create an instance of Phaser with 3 registered parties
    Phaser phaser = new Phaser(3);


    try {
```

Run

From the output of the program note that once the main thread is past the barrier the number of registered parties with the barrier is reduced by one than when we initially created the barrier.

Non-blocking arrival

In the previous example we used the blocking method `arriveAndAwaitAdvance()` to synchronize at the barrier. However, there may be scenarios where we want a thread/task to record arrival at a barrier but not block. For such use cases `Phaser` class provides two non-blocking methods:

- `arrive()`
- `arriveAndDeregister(()`

Here's an example program that demonstrates the use of `arrive()`. Note that the barrier is initialized with a party count of 5 and only the main thread records arrival and moves past the barrier without blocking.

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) {
        // create an instance of Phaser class and register 5 parties
        Phaser phaser = new Phaser(5);

        // main thread records arrival at the barrier
        phaser.arrive();

        System.out.println("main thread moving past the barrier \n Unarrived parties " + ph
```

Run

Phases of a Phaser

You might be wondering why the `Phaser` class is named `Phaser` and it is because an instance of the class moves from one phase to another as the registered parties record arrival and advance. The starting phase is numbered 0, when all the registered parties arrive at the barrier, the `Phaser` instance advances to the next phase which is 1. This pattern continues until the phase reaches the maximum allowed `Integer.MAX_VALUE` and then wraps to zero. The phase numbered 0 is never arrived at by synchronizing parties. The synchronization methods return the arrival phase which starts at 1. The program below prints the phases of a `Phaser` instance as the main thread records arrival and then advances to the next phase.

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) {
        // create an instance of Phaser class and register a single party
        Phaser phaser = new Phaser(1);

        // get the initial phase number
        int phase = phaser.getPhase();
        System.out.println("starting at phase : " + phase);

```

Run

We have another example demonstrating the main thread waiting until 10 phases of a `Phaser` instance complete and then advancing forward. You can consider a scenario where we want the main thread to proceed forward after worker threads have synchronized a certain number of times over a barrier. The example with code comments appears below:

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(15);

```

```
    Phaser phaser = new Phaser(2);

    try {
        // register main thread with the phaser
        int arrivalPhase = phaser.register();
```

Run

Awaiting phase advance

The `Phaser` class provides a thread/task an opportunity to await the current phase a `Phaser` instance is in via the method `awaitAdvance(int)`. The method takes in the phase argument which a thread intends to wait out. However, the invocation returns immediately if the phase the `Phaser` instance is currently in, is different from the passed-in argument to the method. Consider the program below, where the main thread attempts to await the the very first phase numbered 0. We spawn another thread which sleeps for 5 seconds to simulate work and then records arrival at the barrier. Once it does so, the main thread proceeds forward.

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of Phaser class
        Phaser phaser = new Phaser(1);

        try {
```

Run

In the program above, if you change the phase to a higher number e.g. 5 on line#33, the main thread would immediately return from the `awaitAdvance(int)` invocation and the print statement on line#34 will output phase number as 0 instead of 1.

Another example demonstrating the use of `awaitAdvance(int)` appears below. Two parties register with the phaser. The main thread spawns a thread and then `arriveAndDeregister()`-s at the barrier. The call `arriveAndDeregister()` is a non-blocking call and the main thread proceeds forward. Say if we want the main thread to wait for the other worker thread to complete its work and then proceed we can do so by invoking `awaitAdvance(0)` to block the main thread to wait for phase 0 to advance.

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        // create an executor service
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        // create an instance of the Phaser class
        Phaser phaser = new Phaser(2);

        try {
```

Run

Termination

A `Phaser` instance can eventually transition to a terminated state. Once terminated, attempts to register with the instance have no effect and all synchronization methods return immediately with a negative value. One of the ways a `Phaser` can reach a terminal state is when the number of registered parties falls to zero. We can also determine if a `Phaser` is in terminal state by invoking the `isTerminated()` method. The program below demonstrates a `Phaser` instance reaching a terminal state.

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) {
        // initialize a `Phaser` with a single party
        Phaser phaser = new Phaser(1);

        // arrive and deregister so that the registered parties count falls to zero
        phaser.arriveAndDeregister();

        // print the termination status
        System.out.println(phaser.isTerminated());
```

Run

Bear in mind that initializing a `Phaser` with zero registered parties doesn't cause the instance to be in terminated state. The registered party count has to fall from a non-zero count to zero for a `Phaser` to enter into the terminal state. The program below demonstrates these nuances.

```java
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) {
      // initialize a `Phaser` with zero registered parties
       Phaser phaser = new Phaser(0);

       // print the termination status
       System.out.println(phaser.isTerminated());

       // main thread registers with the phaser
```

We can also force terminate a `Phaser` instance using the method `forceTermination()` If so, the waiting threads are abruptly terminated and allowed to proceed past the barrier.

```java
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {
       // create an executor service
       ExecutorService executorService = Executors.newFixedThreadPool(15);

       // register four parties with phaser, when in fact we'll only have
       // threads synchronize at the barrier to cause them to block
       Phaser phaser = new Phaser(4);
       System.out.println("Is phaser terminated " + phaser.isTerminated());
```

`onAdvance()`

When a `Phaser` enters the terminal state is determined by a method `onAdvance()` that can also be overridden in derived classes. By default this method returns true when deregistrations cause the number of registered parties to fall to zero. In case, if we never want a `Phaser` to enter the terminal state, we can override the `onAdvance()` method to always return false.

Furthermore, the `onAdvance()` method can be overridden by subclasses to optionally perform some action when the `Phaser` proceeds to the next phase. The `onAdvance()` is executed by one of the threads/parties triggering the phase advance, i.e. the action desired on advancing to the next phase is only performed by one of the threads. Overriding this method is similar to, but more flexible than, providing a barrier action to a `CyclicBarrier`.

To demonstrate the utility of `onAdvance()` method, we'll create a class `MyPhaser` that extends `Phaser` but terminates after 5 iterations or phases, i.e. threads or registered parties are expected to synchronize at the barrier only five times. We'll also print a log statement in the `onAdvance()` method. The code listing appears below with comments:

```java
import java.util.concurrent.*;

class Demonstration {

    static class MyPhaser extends Phaser {

        public MyPhaser(int registeredParties) {
            super(registeredParties);
        }

        @Override
```

The output of the above program shows that one of the threads executes the `onAdvance()` method and threads/tasks advance from one phase to another when repeatedly synchronizing on the barrier. The interesting aspect to note is that when we ask for the terminal status at the end of the program, it is reported as true and two of the threads print negative phase numbers as return values from the `arriveAndAwaitAdvance()` method. The return values indicate the next phase number and a negative value implies that the phaser is

now in terminal state. The sixth attempt to synchronize on the barrier will fail. You can change the limit to 6 on line#39 in the `for` loop to observe the behavior of the barrier in terminal state. The invocations to `arriveAndAwaitAdvance()` methods return immediately with a negative value.

The above program can be rewritten without a separate class as shown in the widget below. Each thread/task checks for the phaser's terminal condition in a while loop and performs some action repeatedly. This is the idiomatic use of overriding the `onAdvance()` method and its default functionality.

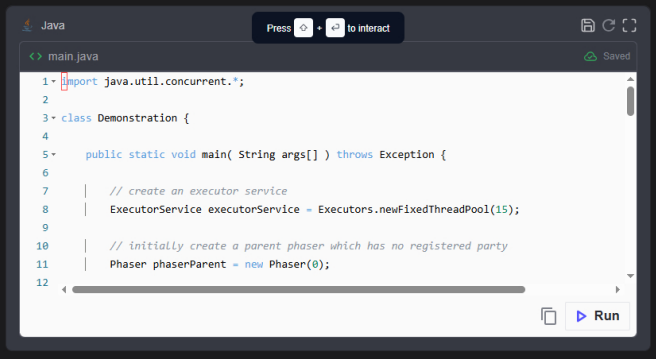
Tiering

The `Phaser` allows registration of a maximum number of 65,535 parties. For scenarios, where we want to register a greater number of parties than allowed, we can construct *tiered phasers* to accommodate an arbitrarily large set of participants. Tiering refers to arranging `Phaser` instances as a tree and establishing child-parent relationships between them.

Apart from circumventing the limitation on the number of possible registered parties, tiered phasers can reduce the heavy synchronization contention experienced by `Phaser`-s with large number of parties. Participants can be divided among group of sub-phasers that share a common parent. Doing so may greatly increase throughput even though it incurs greater per-operation overhead.

In a tree of tiered phasers, registration and deregistration of child phasers with their parent phaser are managed automatically. Whenever the number of registered parties of a child phaser becomes non-zero the child phaser is registered with its parent and vice versa, i.e. the child phaser is deregistered with its parent in case the registered parties for the child become zero.

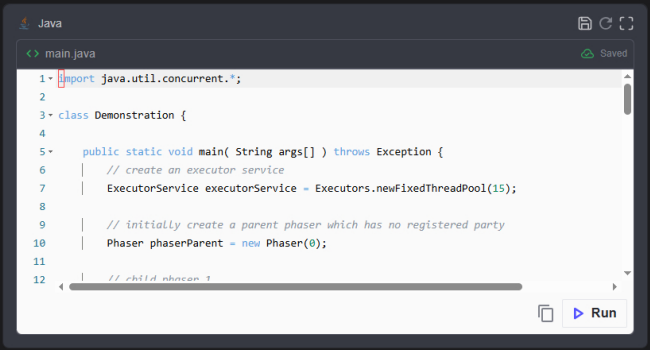
The widget below illustrates a simple program showing the working of parent and child phasers:



If you run the above program, you'll see that the number of registered parties for the `parentPhaser` is output to be 4. We create a parent-child relationship between the `parentPhaser` and the other three phasers by passing in the `parentPhaser` as the argument for the parent in the `Phase` constructor. It is important to note that from the parent phaser's perspective the number of registered parties is 4 which includes the main thread and the other three phasers. *The registered parties for each individual child phaser don't factor into the count of registered parties for the parent phaser.* In fact, the parent phaser moves to the next phase when all the child phases move to the next phase and the main thread records arrival with the parent phaser. However, this also implies that any thread/task that invokes `arriveAndAwaitAdvance()` on a child phase will not move ahead until the root phase moves to the next phase. Awaiting advance on any child phaser amounts to awaiting advance on all the children of the root phaser.

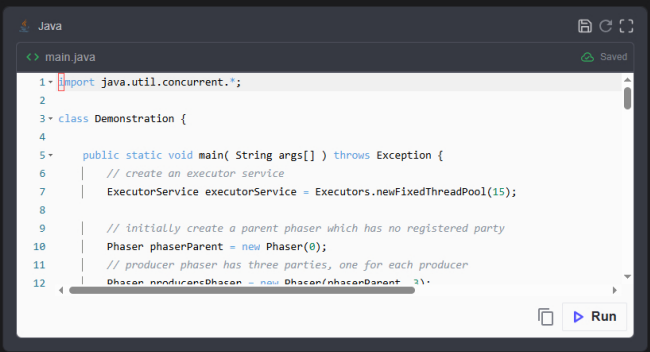
As an exercise if you comment out lines 33 and 34 in the above widget and re-run the program, you'll observe the program will hang and execution will time out. The other thread, even though they arrive at their respective child phasers but don't move past the child barrier since the root phaser is still waiting for an unarrived party.

Another caveat when working with tiered phasers is to be cognizant that if the parent phaser advances its phase then all the child phasers do too, even if the required number of parties have not arrived at the child phaser. This is illustrated by the following program, where threads block on child phasers proceed past the barrier when the main thread artificially causes a phase advance for the parent phaser.



An involved example

Let's consider a hypothetical example of producer and consumer threads. Suppose our program flow spawns a bunch of producer and consumer threads and we want to structure the program such that the producer threads run first, the consumer threads second and finally the main thread exits after the two sets of threads/tasks are done. The program along with detailed comments appears in the widget below:



Conclusion

In summary, `Phaser` is an advanced and sophisticated barrier construct that can be used in complex synchronization scenarios, however, beware its use can also introduce subtle bugs that may be hard to find. For simple use cases go with the simpler `CyclicBarrier` or `CountDownLatch`.

← Back lesson

Exchanger

Mark As Completed Next →

IllegalMonitorStateException