17. Java.util.concurrent.CyclicBarrier in Java

CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

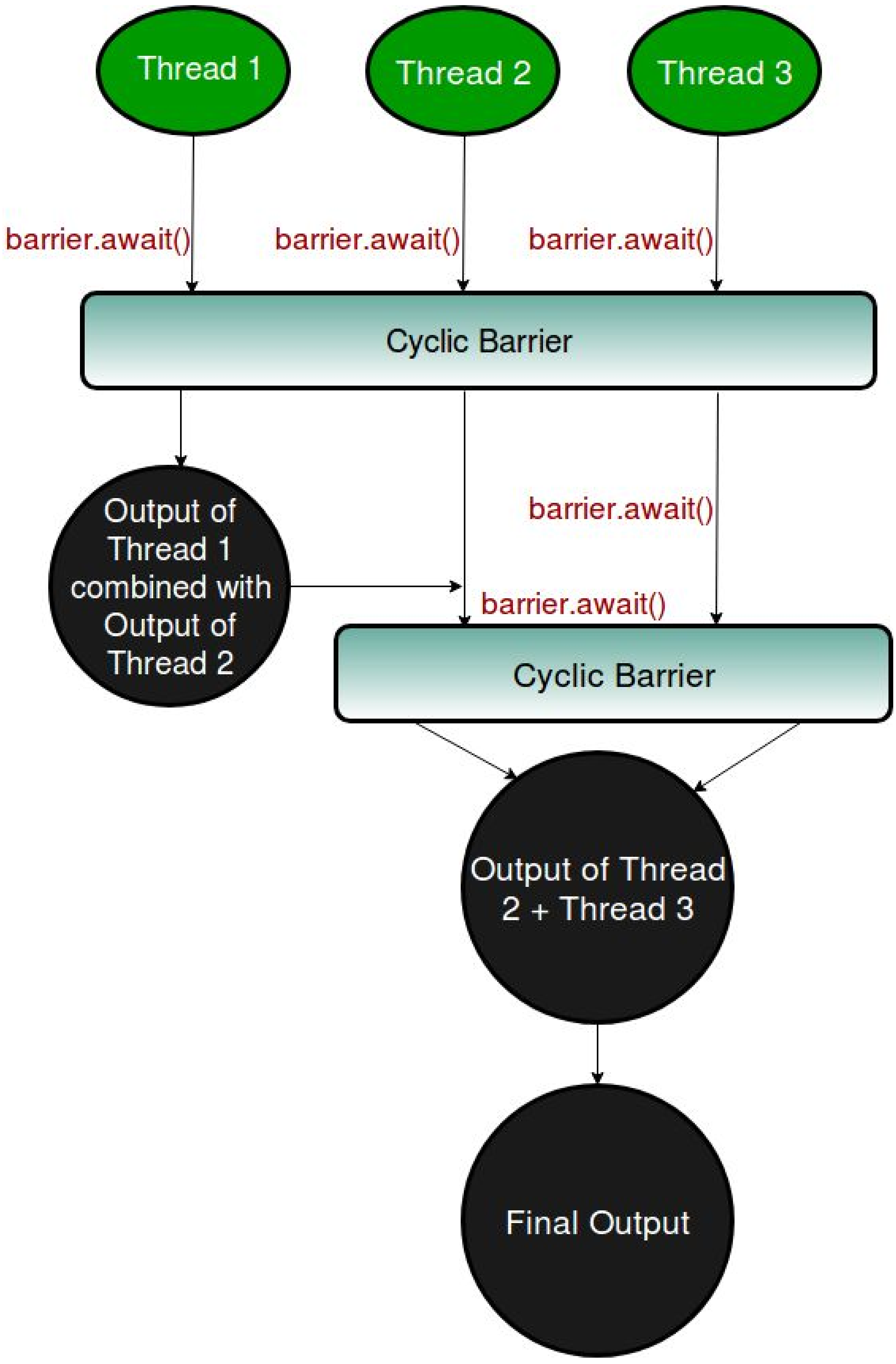
Working of CyclicBarrier

CyclicBarriers are defined in java.util.concurrent package. First a new instance of a CyclicBarriers is created specifying the number of threads that the barriers should wait upon.

```
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);
```

Each and every thread does some computation and after completing it’s execution, calls await() methods as shown:

```
public void run()
{
    // thread does the computation
    newBarrier.await();
}
```



Once the number of threads that called await() equals **numberOfThreads**, the barrier then gives a way for the waiting threads. The CyclicBarrier can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier.

```
Runnable action = ...
//action to be performed when all threads reach the barrier;
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads, action);
```

Important Methods of CyclicBarrier:

1. **getParties:** Returns the number of parties required to trip this barrier.

   Syntax:

   `public int getParties()`

   Returns:

   the number of parties required to trip this barrier
2. **reset:** Resets the barrier to its initial state.

   Syntax:

   `public void reset()`

   Returns:

   void but resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.
3. **isBroken:** Queries if this barrier is in a broken state.

   Syntax:

   `public boolean isBroken()`

   Returns:

   true if one or more parties broke out of this barrier due to interruption or timeout since construction or the last reset, or a barrier action failed due to an exception; false otherwise.
4. **getNumberWaiting:** Returns the number of parties currently waiting at the barrier.

   Syntax:

   `public int getNumberWaiting()`

   Returns:

   the number of parties currently blocked in await()

5. **await:** Waits until all parties have invoked await on this barrier.

   Syntax:

   `public int await() throws InterruptedException, BrokenBarrierException`

   Returns:

   the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive.

6. **await:** Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.

```
public int await(long timeout, TimeUnit unit) throws
InterruptedException,BrokenBarrierException, TimeoutException
```

```
//JAVA program to demonstrate execution on Cyclic Barrier

import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

class Computation1 implements Runnable
{
	public static int product = 0;
	public void run()
	{
		product = 2 * 3;
		try
		{
			Tester.newBarrier.await();
		}
		catch (InterruptedException | BrokenBarrierException e)
		{
			e.printStackTrace();
		}
	}
}
```

```
class Computation2 implements Runnable
{
        public static int sum = 0;
        public void run()
        {
                // check if newBarrier is broken or not
                System.out.println("Is the barrier broken? - " +
Tester.newBarrier.isBroken());
                sum = 10 + 20;
                try
                {
                        Tester.newBarrier.await(3000, TimeUnit.MILLISECONDS);

                        // number of parties waiting at the barrier
                        System.out.println("Number of parties waiting at the barrier "+
                        "at this point = " + Tester.newBarrier.getNumberWaiting());
                }
                catch (InterruptedException | BrokenBarrierException e)
                {
                        e.printStackTrace();
                }
                catch (TimeoutException e)
                {
                        e.printStackTrace();
                }
        }
}


public class Tester implements Runnable
{
        public static CyclicBarrier newBarrier = new CyclicBarrier(3);

        public static void main(String[] args)
        {
                // parent thread
                Tester test = new Tester();

                Thread t1 = new Thread(test);
                t1.start();
        }
```

```
	public void run()
	{
		System.out.println("Number of parties required to trip the barrier = "+
		newBarrier.getParties());
		System.out.println("Sum of product and sum = " +
(Computation1.product +
		Computation2.sum));

		// objects on which the child thread has to run
		Computation1 comp1 = new Computation1();
		Computation2 comp2 = new Computation2();

		// creation of child thread
		Thread t1 = new Thread(comp1);
		Thread t2 = new Thread(comp2);

		// moving child thread to runnable state
		t1.start();
		t2.start();

		try
		{
			Tester.newBarrier.await();
		}
		catch (InterruptedException | BrokenBarrierException e)
		{
			e.printStackTrace();
		}

		// barrier breaks as the number of thread waiting for the barrier
		// at this point = 3
		System.out.println("Sum of product and sum = " +
(Computation1.product +
		Computation2.sum));

		// Resetting the newBarrier
		newBarrier.reset();
		System.out.println("Barrier reset successful");
	}
}
```