Java Concurrency Utilities Report

Introduction

In this report, we will explore various classes from the `java.util.concurrent` package, which are fundamental for concurrent programming in Java. I will cover the following classes:

1. BlockingQueue and ArrayBlockingQueue
2. ReadWriteLock
3. AtomicInteger
4. CyclicBarrier
5. ExecutorService, Executors, Callable, Future

For each class, I will:

- Provide a brief explanation.
- Include a multithreaded code example.
- Provide sample output of the program.

1. BlockingQueue and ArrayBlockingQueue

Explanation

A `BlockingQueue` in Java, provided by the `java.util.concurrent` package, is a thread-safe queue that allows concurrent insertion and removal of elements. The primary feature of a `BlockingQueue` is that it can **block** threads when attempting to add to a full queue or remove from an empty queue. This blocking behavior is useful for implementing producer-consumer scenarios where data is shared between multiple threads.

There are several implementations of `BlockingQueue` in Java, including:

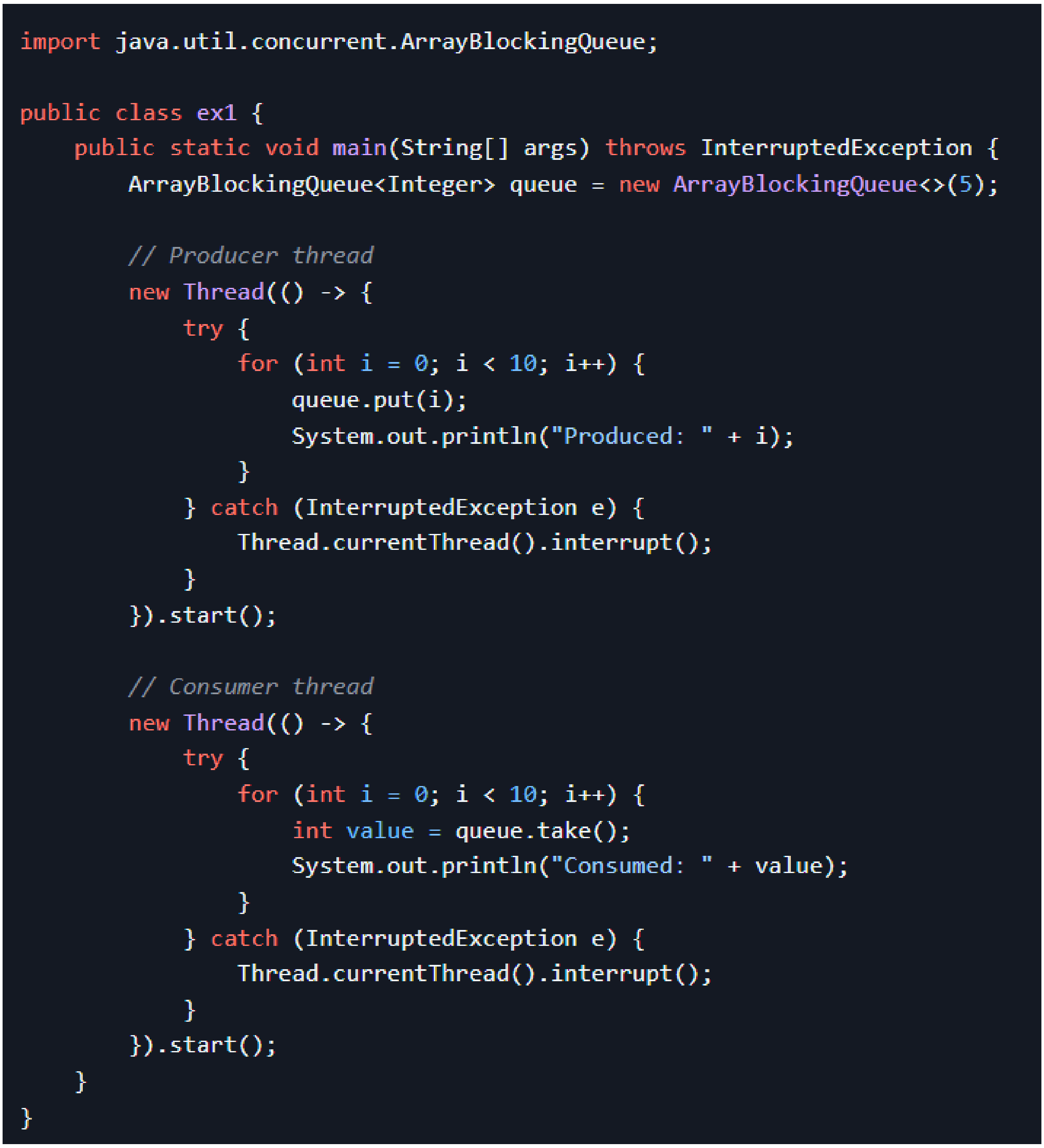
- `ArrayBlockingQueue`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `SynchronousQueue`
- `DelayQueue`
- `LinkedTransferQueue`

Among these, the `ArrayBlockingQueue` is a fixed-size, bounded queue backed by an array. It is suitable for scenarios where the maximum number of elements is known in advance. The `ArrayBlockingQueue` follows a **First-In-First-Out (FIFO)** order for processing elements.

Key Methods:

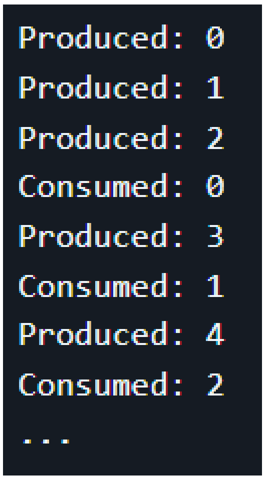
- **put()**: Adds an element to the queue, blocking if the queue is full.
- **take()**: Removes and returns an element from the queue, blocking if the queue is empty.
- **offer()**: Attempts to add an element without blocking, returns `false` if full.
- **poll()**: Retrieves and removes the head of the queue, returning `null` if empty.

Code Example (ex1.java)



```java
import java.util.concurrent.ArrayBlockingQueue;

public class ex1 {
    public static void main(String[] args) throws InterruptedException {
        ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

        // Producer thread
        new Thread(() -> {
            try {
                for (int i = 0; i < 10; i++) {
                    queue.put(i);
                    System.out.println("Produced: " + i);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        // Consumer thread
        new Thread(() -> {
            try {
                for (int i = 0; i < 10; i++) {
                    int value = queue.take();
                    System.out.println("Consumed: " + value);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }
}
```

Example Output



```
Produced: 0
Produced: 1
Produced: 2
Consumed: 0
Produced: 3
Consumed: 1
Produced: 4
Consumed: 2
...
```

2. ReadWriteLock

Explanation

The `ReadWriteLock` in Java, part of the `java.util.concurrent.locks` package, is a lock mechanism that allows **multiple threads to read** a shared resource simultaneously but **only one thread to write** at a time. This mechanism is useful in scenarios where **read operations are more frequent** than write operations, as it helps improve concurrency without risking data inconsistency.

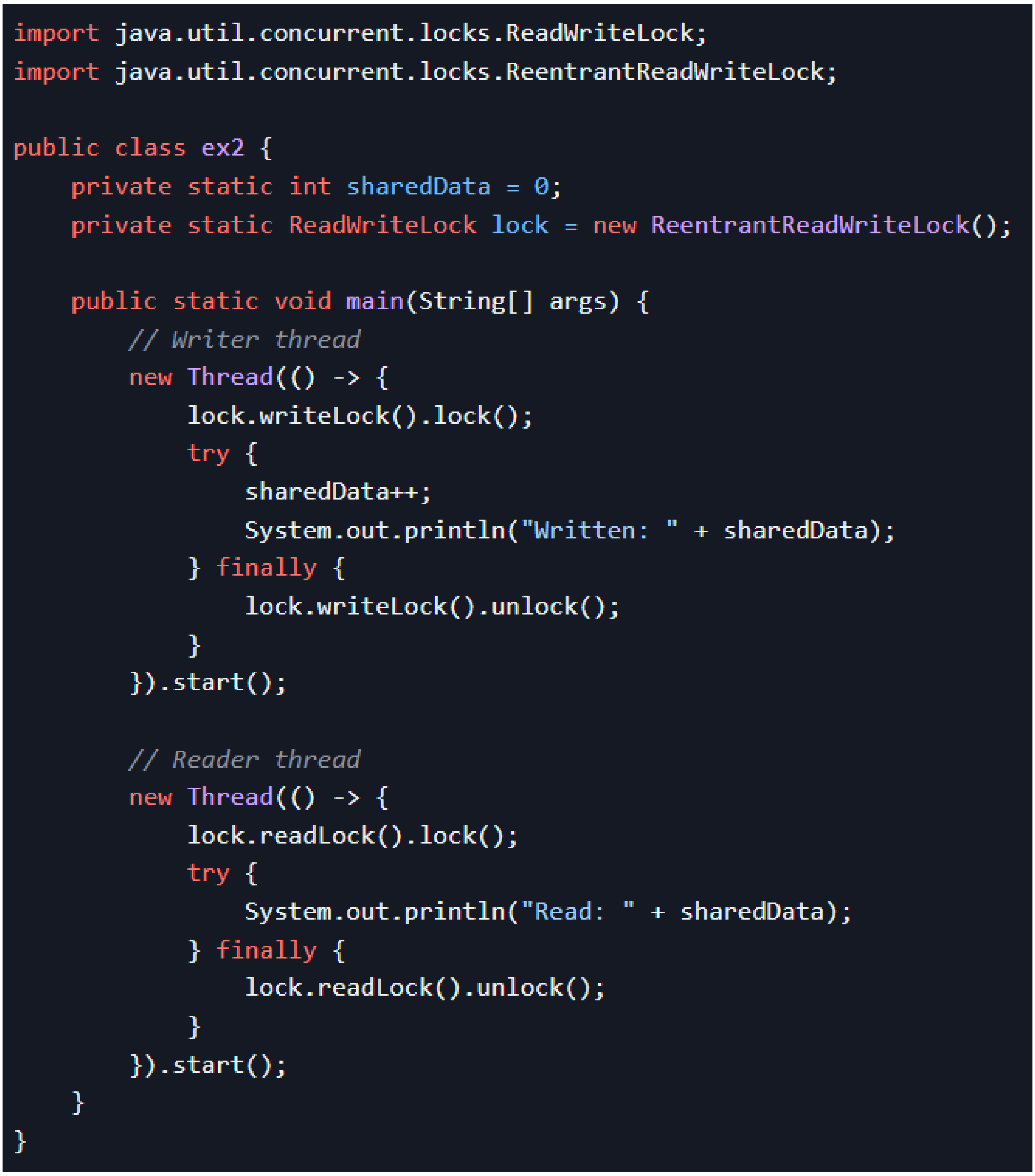
Key Concepts:

- **Read Lock:** Multiple threads can acquire the read lock as long as no thread holds the write lock. This allows concurrent reading.
- **Write Lock:** Only one thread can acquire the write lock, and it can do so only if no threads are holding the read lock. This ensures exclusive access for modification.

Typical Usage:

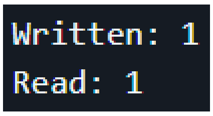
1. Use `readLock().lock()` to acquire the read lock.
2. Use `writeLock().lock()` to acquire the write lock.
3. Always pair each lock acquisition with `unlock()` to release the lock.

Code Example (ex2.java)



```java
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ex2 {
    private static int sharedData = 0;
    private static ReadWriteLock lock = new ReentrantReadWriteLock();

    public static void main(String[] args) {
        // Writer thread
        new Thread(() -> {
            lock.writeLock().lock();
            try {
                sharedData++;
                System.out.println("Written: " + sharedData);
            } finally {
                lock.writeLock().unlock();
            }
        }).start();

        // Reader thread
        new Thread(() -> {
            lock.readLock().lock();
            try {
                System.out.println("Read: " + sharedData);
            } finally {
                lock.readLock().unlock();
            }
        }).start();
    }
}
```

Example Output



```
Written: 1
Read: 1
```

3. AtomicInteger

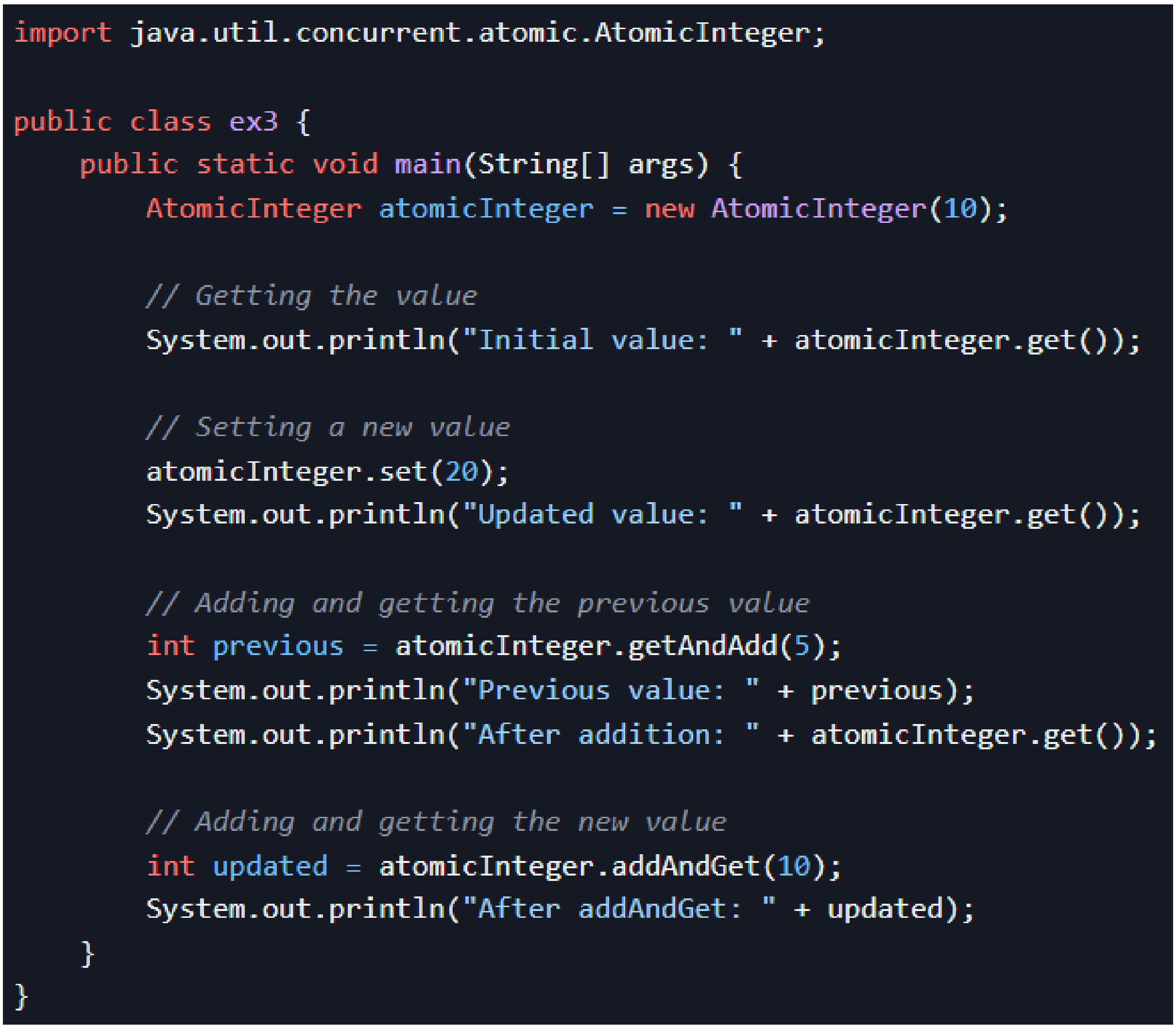
Explanation

The `AtomicInteger` class, part of the `java.util.concurrent.atomic` package, provides an atomic, thread-safe way to manipulate integer values. It offers methods to perform atomic operations without using synchronization. This class is especially useful when multiple threads need to update a shared integer variable.

Key Methods:

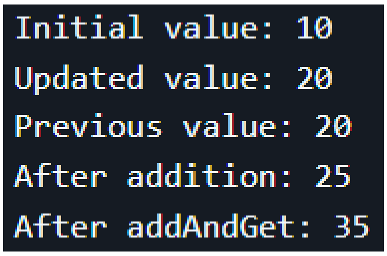
- **get()**: Retrieves the current value.
- **set(int newValue)**: Sets the value to the specified value.
- **getAndAdd(int delta)**: Atomically adds the given value and returns the previous value.
- **addAndGet(int delta)**: Atomically adds the given value and returns the updated value.

Code Example (ex3.java)



```java
import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(10);

        // Getting the value
        System.out.println("Initial value: " + atomicInteger.get());

        // Setting a new value
        atomicInteger.set(20);
        System.out.println("Updated value: " + atomicInteger.get());

        // Adding and getting the previous value
        int previous = atomicInteger.getAndAdd(5);
        System.out.println("Previous value: " + previous);
        System.out.println("After addition: " + atomicInteger.get());

        // Adding and getting the new value
        int updated = atomicInteger.addAndGet(10);
        System.out.println("After addAndGet: " + updated);
    }
}
```

Example Output



```
Initial value: 10
Updated value: 20
Previous value: 20
After addition: 25
After addAndGet: 35
```

4. CyclicBarrier

Explanation

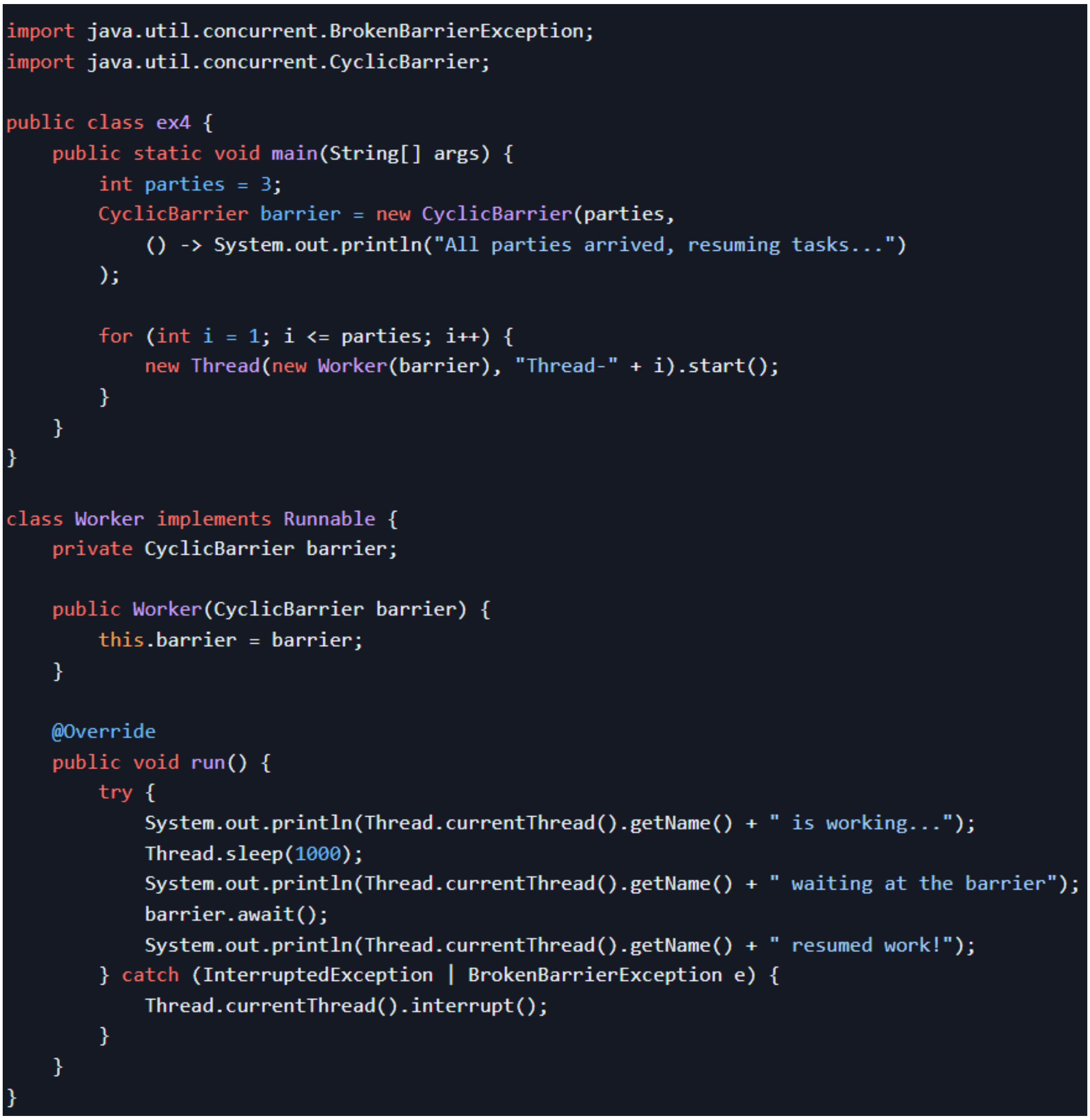
The `CyclicBarrier` class in Java, part of the `java.util.concurrent` package, is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. Once all threads have arrived at the barrier, they are released simultaneously to continue executing.

This is useful when multiple threads need to **perform some work in phases** and wait for each other between phases.

Key Concepts:

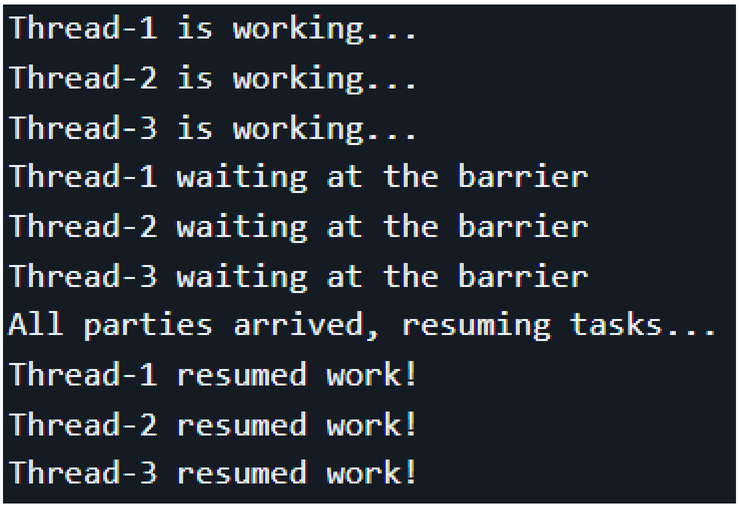
- **Barrier Point:** The point where threads wait for each other.
- **await():** A method that makes a thread wait at the barrier until all participating threads reach the barrier.
- **Action:** An optional action that can be executed once the barrier is released.

Code Example (ex4.java)



```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ex4 {
    public static void main(String[] args) {
        int parties = 3;
        CyclicBarrier barrier = new CyclicBarrier(parties,
            () -> System.out.println("All parties arrived, resuming tasks...")
        );

        for (int i = 1; i <= parties; i++) {
            new Thread(new Worker(barrier), "Thread-" + i).start();
        }
    }
}

class Worker implements Runnable {
    private CyclicBarrier barrier;

    public Worker(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is working...");
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + " waiting at the barrier");
            barrier.await();
            System.out.println(Thread.currentThread().getName() + " resumed work!");
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Example Output



```
Thread-1 is working...
Thread-2 is working...
Thread-3 is working...
Thread-1 waiting at the barrier
Thread-2 waiting at the barrier
Thread-3 waiting at the barrier
All parties arrived, resuming tasks...
Thread-1 resumed work!
Thread-2 resumed work!
Thread-3 resumed work!
```

5. ExecutorService, Executors, Callable, Future

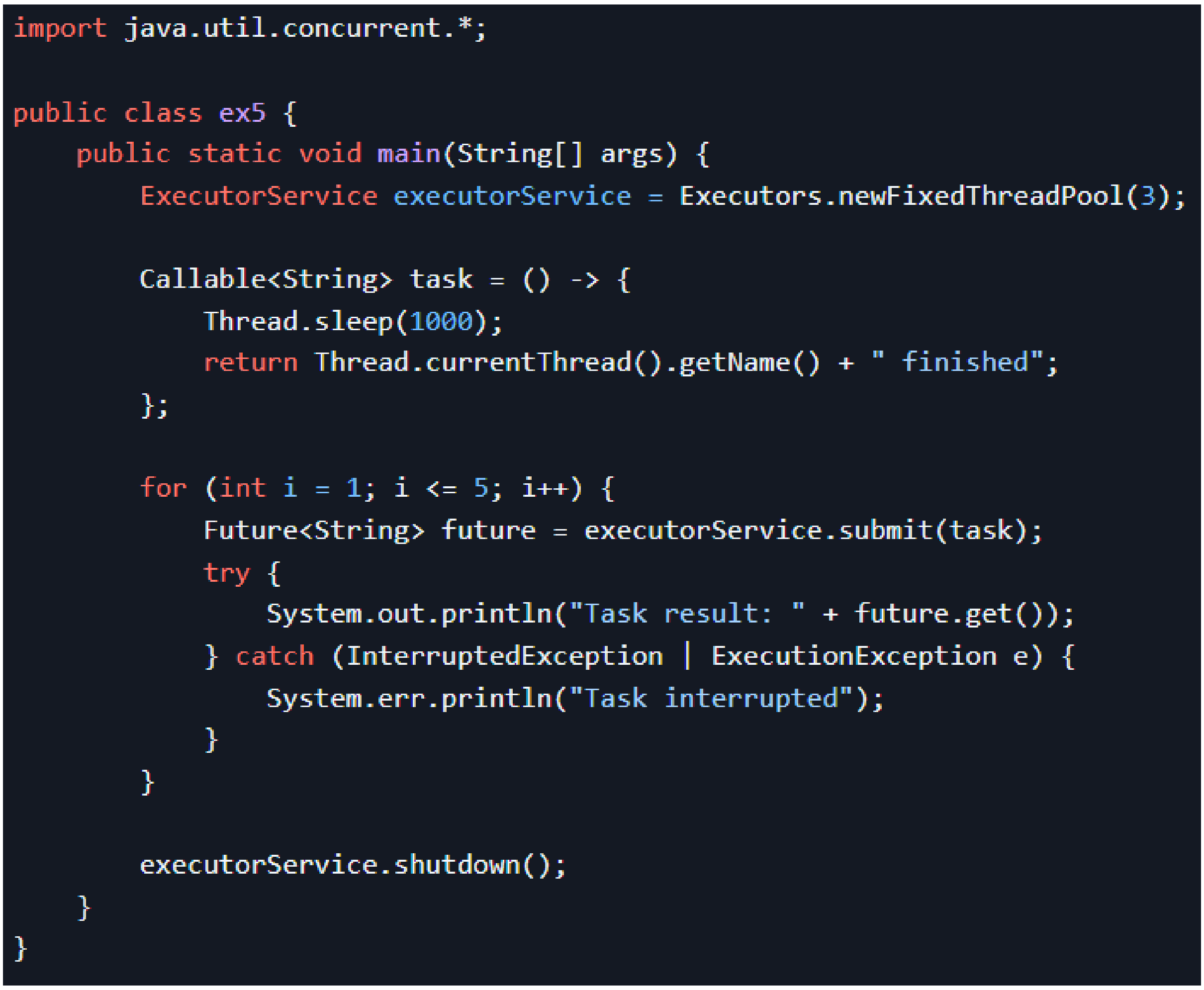
Explanation

The `ExecutorService` in Java, part of the `java.util.concurrent` package, provides a way to manage and control asynchronous task execution. It allows tasks to be executed concurrently without manually creating and managing threads. The primary advantage of `ExecutorService` is that it can efficiently manage a pool of threads, reducing the overhead of thread creation.

Key Components:

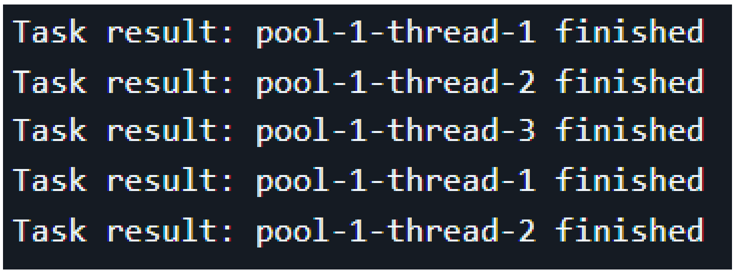
- **ExecutorService:** The core interface for asynchronous task execution.
- **Executors:** A factory class for creating various types of thread pools (e.g., fixed thread pool).
- **Callable:** Similar to `Runnable`, but can return a result and throw an exception.
- **Future:** Represents the result of an asynchronous computation.

Code Example (ex5.java)



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

public class ex5 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        Callable<String> task = () -> {
            Thread.sleep(1000);
            return Thread.currentThread().getName() + " finished";
        };

        for (int i = 1; i <= 5; i++) {
            Future<String> future = executorService.submit(task);
            try {
                System.out.println("Task result: " + future.get());
            } catch (InterruptedException | ExecutionException e) {
                System.err.println("Task interrupted");
            }
        }

        executorService.shutdown();
    }
}
```

Example Output



```
Task result: pool-1-thread-1 finished
Task result: pool-1-thread-2 finished
Task result: pool-1-thread-3 finished
Task result: pool-1-thread-1 finished
Task result: pool-1-thread-2 finished
```