(i) [BlockingQueue]

BlockingQueue is an interface that supports queue data structure. BlockingQueue prevents race condition and supports synchronization. For example, when retrieving an element from the empty queue, this action will be blocked and wait until at least one element is added. Also, when adding an element to the queue which is full, this action will be blocked and wait until at least one element is retrieved from the queue.

[ArrayBlockingQueue]

ArrayBlockingQueue is a class that implements the BlockingQueue interface. ArrayBlockingQueue cannot have infinite elements. In other words, you must set the limitation when creating an instance of ArrayBlockingQueue. This queue prevents race condition and supports synchronization.

[ex1.java]

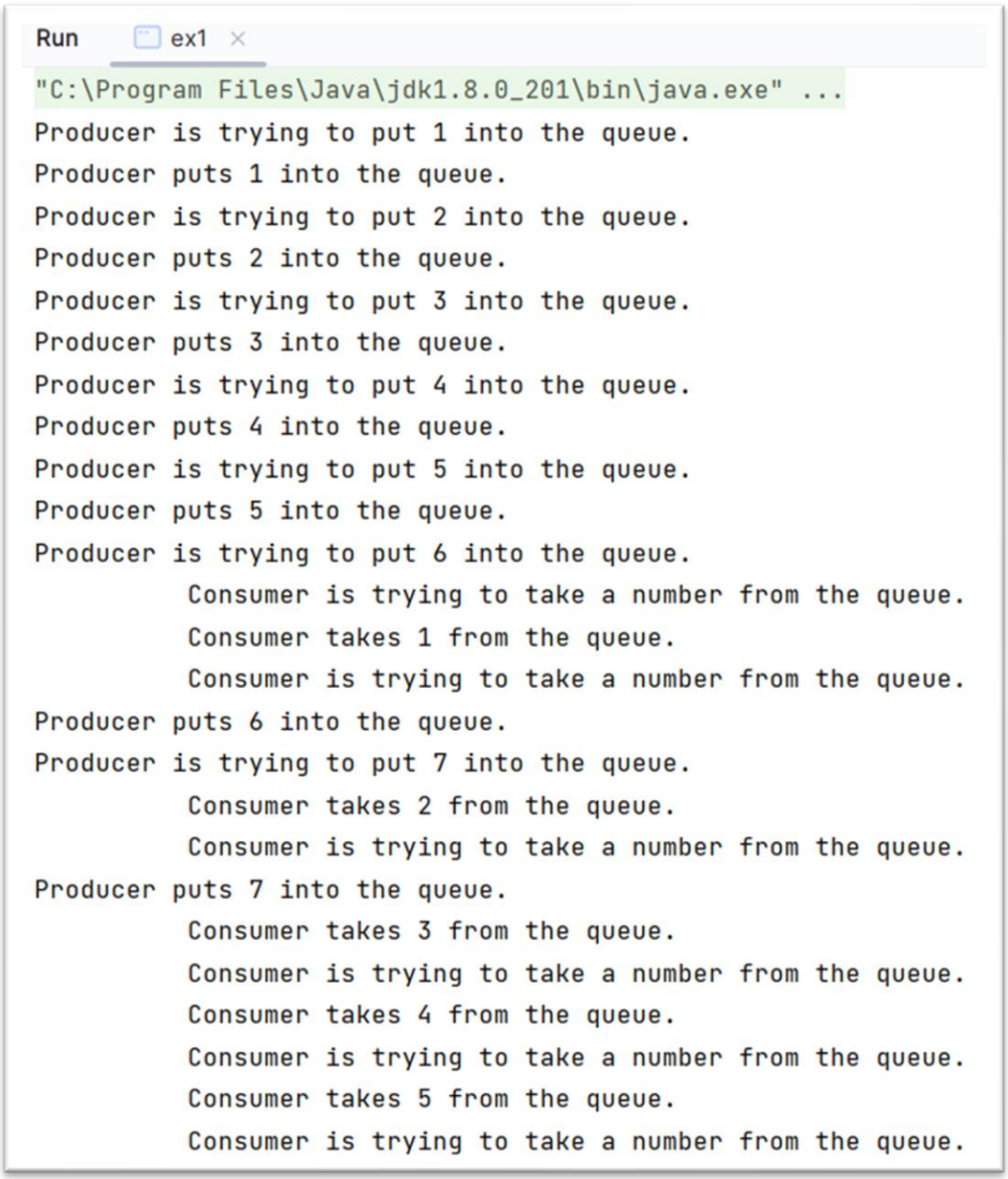
```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public final class ex1 {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);
        new Producer(queue);
        new Consumer(queue);
    }
}

final class Producer extends Thread {
    private final BlockingQueue<Integer> queue;

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
        start();
    }

    private void beforePutting(int number) {
        System.out.println("Producer is trying to put " + number + " into
the queue.");
    }

    private void afterPutting(int number) {
        System.out.println("Producer puts " + number + " into the queue.");
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            try {
                beforePutting(i + 1);
                queue.put(i + 1);
```

```
                afterPutting(i + 1);
            } catch (Exception ignored) {}
        }
    }
}

final class Consumer extends Thread {
    private final BlockingQueue<Integer> queue;

    public Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
        start();
    }

    private void beforeTaking() {
        System.out.println("        Consumer is trying to take a number from
the queue.");
    }

    private void afterTaking(int number) {
        System.out.println("        Consumer takes " + number + " from the
queue.");
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            try {
                beforeTaking();
                int number = queue.take();
                afterTaking(number);
            } catch (Exception ignored) {}
        }
    }
}
```

[explanation]

This source code has producer thread and consumer thread. The producer thread plays a role of putting a number into a queue. And the consumer thread plays a role of taking a number from the queue. The BlockingQueue prevents a race condition.

[result]


Run
ex1
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Producer is trying to put 1 into the queue.
Producer puts 1 into the queue.
Producer is trying to put 2 into the queue.
Producer puts 2 into the queue.
Producer is trying to put 3 into the queue.
Producer puts 3 into the queue.
Producer is trying to put 4 into the queue.
Producer puts 4 into the queue.
Producer is trying to put 5 into the queue.
Producer puts 5 into the queue.
Producer is trying to put 6 into the queue.
Consumer is trying to take a number from the queue.
Consumer takes 1 from the queue.
Consumer is trying to take a number from the queue.
Producer puts 6 into the queue.
Producer is trying to put 7 into the queue.
Consumer takes 2 from the queue.
Consumer is trying to take a number from the queue.
Producer puts 7 into the queue.
Consumer takes 3 from the queue.
Consumer is trying to take a number from the queue.
Consumer takes 4 from the queue.
Consumer is trying to take a number from the queue.
Consumer takes 5 from the queue.
Consumer is trying to take a number from the queue.

Run ex1

```
Producer is trying to put 8 into the queue.
Producer puts 8 into the queue.
          Consumer takes 6 from the queue.
          Consumer is trying to take a number from the queue.
Producer is trying to put 9 into the queue.
          Consumer takes 7 from the queue.
          Consumer is trying to take a number from the queue.
Producer puts 9 into the queue.
Producer is trying to put 10 into the queue.
          Consumer takes 8 from the queue.
          Consumer is trying to take a number from the queue.
Producer puts 10 into the queue.
          Consumer takes 9 from the queue.
          Consumer is trying to take a number from the queue.
Producer is trying to put 11 into the queue.
          Consumer takes 10 from the queue.
          Consumer is trying to take a number from the queue.
          Consumer takes 11 from the queue.
          Consumer is trying to take a number from the queue.
Producer puts 11 into the queue.
Producer is trying to put 12 into the queue.
Producer puts 12 into the queue.
Producer is trying to put 13 into the queue.
Producer puts 13 into the queue.
Producer is trying to put 14 into the queue.
          Consumer takes 12 from the queue.
          Consumer is trying to take a number from the queue.
```

Run ex1

```
Producer puts 14 into the queue.
Producer is trying to put 15 into the queue.
Producer puts 15 into the queue.
Producer is trying to put 16 into the queue.
          Consumer takes 13 from the queue.
          Consumer is trying to take a number from the queue.
Producer puts 16 into the queue.
          Consumer takes 14 from the queue.
          Consumer is trying to take a number from the queue.
Producer is trying to put 17 into the queue.
          Consumer takes 15 from the queue.
          Consumer is trying to take a number from the queue.
          Consumer takes 16 from the queue.
          Consumer is trying to take a number from the queue.
          Consumer takes 17 from the queue.
          Consumer is trying to take a number from the queue.
Producer puts 17 into the queue.
Producer is trying to put 18 into the queue.
Producer puts 18 into the queue.
Producer is trying to put 19 into the queue.
Producer puts 19 into the queue.
Producer is trying to put 20 into the queue.
Producer puts 20 into the queue.
          Consumer takes 18 from the queue.
          Consumer is trying to take a number from the queue.
          Consumer takes 19 from the queue.
          Consumer is trying to take a number from the queue.
          Consumer takes 20 from the queue.
```

(ii) [ReadWriteLock]

ReadWriteLock is an interface that supports synchronization. When multiple threads try to read shared resources, then all these actions will be allowed simultaneously, But, when multiple threads try to writes to shared resources, then only one thread can access shared resources and the other threads will be blocked and wait until a thread that has a lock releases the lock.

[ex2.java]

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

public class ex2 {
    public static void main(String[] args) {
        ArrayList<Integer> arrayList = new ArrayList<>();
        ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
        Writer[] writers = new Writer[3];

        for (int i = 0; i < writers.length; i++) {
            writers[i] = new Writer(arrayList, readWriteLock, i * 100);
        }

        for (Writer writer : writers) {
            try {
                writer.join();
            } catch (Exception ignored) {}
        }

        for (int i = 0; i < 3; i++) {
            new Reader(arrayList, readWriteLock, ("Thread-" + i));
        }
    }
}

final class Writer extends Thread {
    private final ArrayList<Integer> arrayList;
    private final ReadWriteLock readWriteLock;
    private final int num;


    public Writer(ArrayList<Integer> arrayList, ReadWriteLock
readWriteLock, int num) {
        this.arrayList = arrayList;
        this.readWriteLock = readWriteLock;
        this.num = num;
        start();
    }

    @Override
    public void run() {
        readWriteLock.writeLock().lock();
```

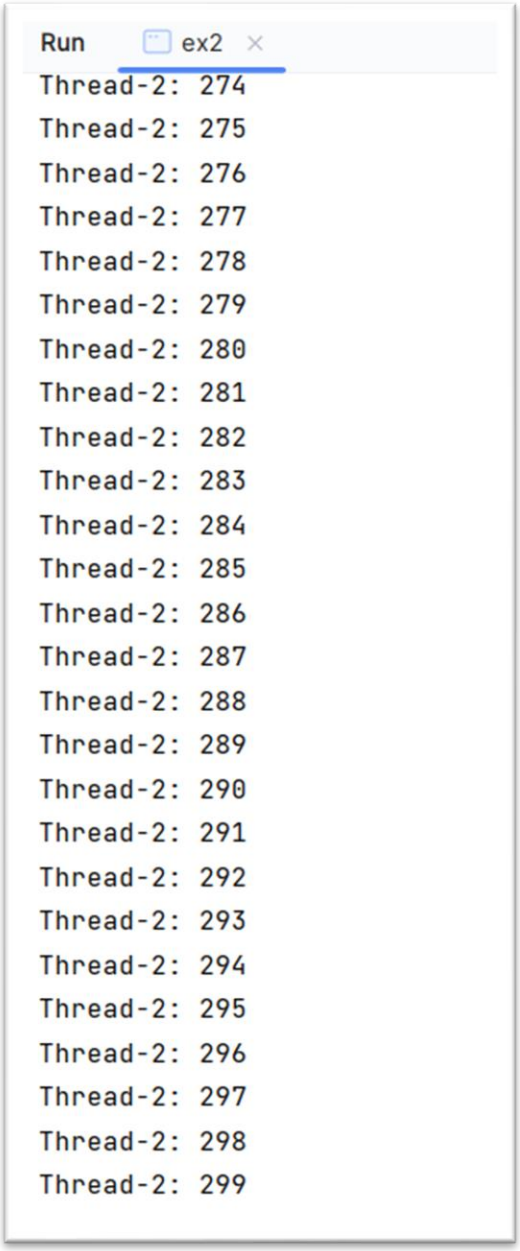
```
        for (int i = num; i < num + 100; i++) {
            arrayList.add(i);
        }

        readWriteLock.writeLock().unlock();
    }
}

final class Reader extends Thread {
    ArrayList<Integer> arrayList;
    ReadWriteLock readWriteLock;

    public Reader(ArrayList<Integer> arrayList, ReadWriteLock
readWriteLock, String name) {
        this.arrayList = arrayList;
        this.readWriteLock = readWriteLock;
        setName(name);
        start();
    }

    @Override
    public void run() {
        readWriteLock.readLock().lock();

        for (Integer integer : arrayList) {
            System.out.println(getName() + ": " + integer);
        }

        readWriteLock.readLock().unlock();
    }
}
```

[explanation]

This source code has 3 writer threads and 3 reader threads. The writer thread plays a role of adding an element into arraylist. And the reader thread plays a role of iterating the arraylist. When writers try to add an element into shared arraylist, the ReadWriteLock prevents a race condition. However, when readers try to iterate the shared arraylist, the ReadWriteLock allows the readers to iterate the arraylist simultaneously.

[result]

(Middle results skipped)


Run
ex2
Thread-2: 274
Thread-2: 275
Thread-2: 276
Thread-2: 277
Thread-2: 278
Thread-2: 279
Thread-2: 280
Thread-2: 281
Thread-2: 282
Thread-2: 283
Thread-2: 284
Thread-2: 285
Thread-2: 286
Thread-2: 287
Thread-2: 288
Thread-2: 289
Thread-2: 290
Thread-2: 291
Thread-2: 292
Thread-2: 293
Thread-2: 294
Thread-2: 295
Thread-2: 296
Thread-2: 297
Thread-2: 298
Thread-2: 299

(iii) [AtomicInteger]

AtomicInteger is a class that represents a **int** variable. This class supports synchronization and ensures that when a thread read an atomicinteger variable or write to the variable, this class prevents other threads from accessing the variable.

[ex3.java]

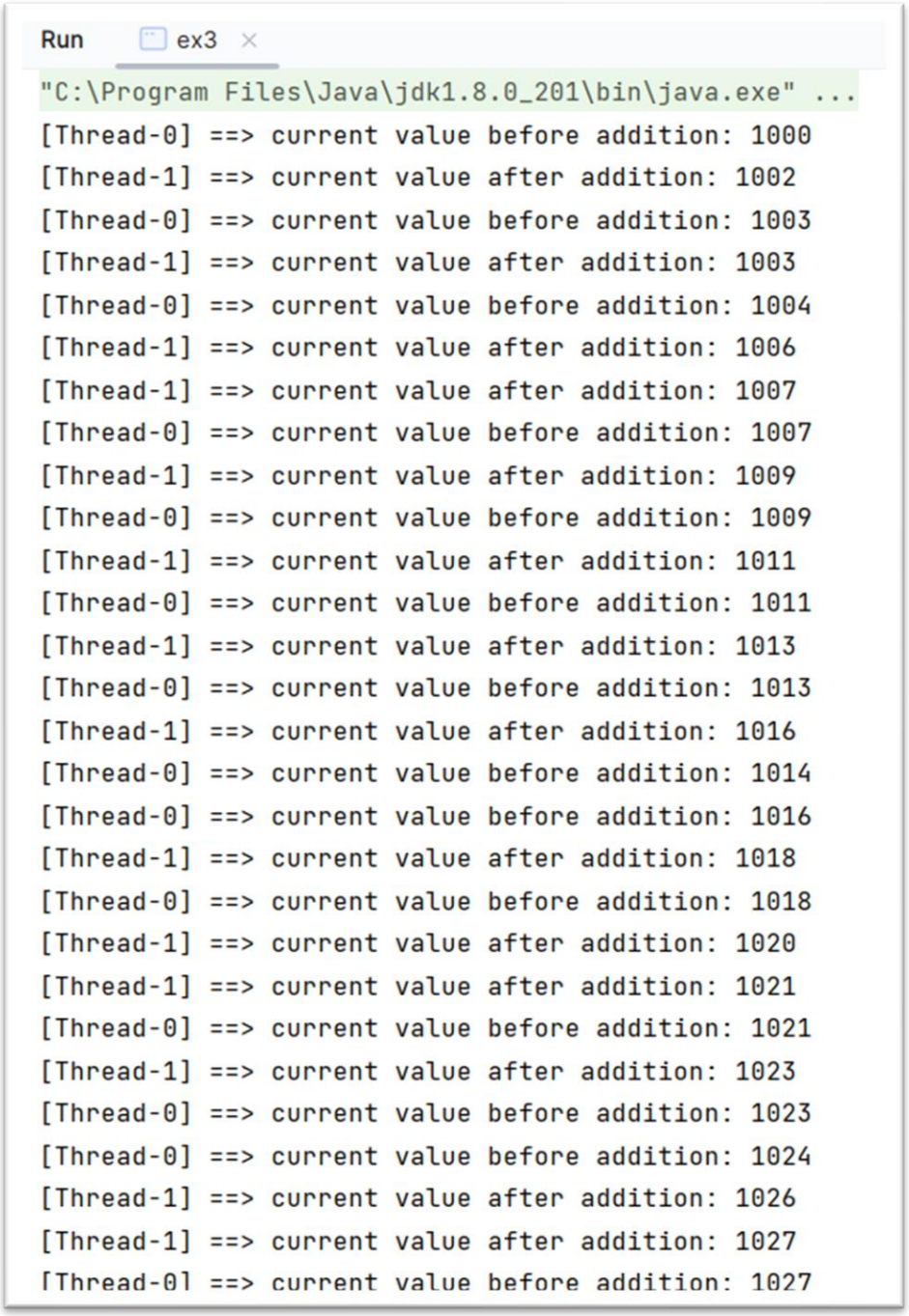
```
import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger();
        atomicInteger.set(1000);

        NumProducer1 numProducer1 = new NumProducer1(atomicInteger);
        NumProducer2 numProducer2 = new NumProducer2(atomicInteger);

        try {
            numProducer1.join();
            numProducer2.join();
        } catch (Exception ignored) {}

        System.out.println("Result: " + atomicInteger.get());
    }
}

final class NumProducer1 extends Thread {
    private final AtomicInteger atomicInteger;

    public NumProducer1(AtomicInteger atomicInteger) {
        this.atomicInteger = atomicInteger;
        start();
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                Thread.sleep(100);
                System.out.println("[" + getName() + "] ==> current value
before addition: " + atomicInteger.getAndAdd(1));
            } catch (Exception ignored) {}
        }
    }
}

final class NumProducer2 extends Thread {
    private final AtomicInteger atomicInteger;

    public NumProducer2(AtomicInteger atomicInteger) {
        this.atomicInteger = atomicInteger;
        start();
    }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                Thread.sleep(100);
                System.out.println("[" + getName() + "] ==> current value
after addition: " + atomicInteger.addAndGet(1));
            } catch (Exception ignored) {}
        }
    }
}
```

[explanation]

This source code two producer threads. One uses getAndAdd( ) method, and the other uses addAndGet( ) method. The initial value of atomicinteger is determined by set(1000) method. And each threads add 100 to atomicinteger. So the final result is 1200. When the two producer add 100 to atomicinteger, getAndAdd( ) method and addAndGet( ) method prevent a race condition.

[result]



(Middle results skipped)

Run ex3

```
[Thread-0] ==> current value before addition: 1174
[Thread-1] ==> current value after addition: 1178
[Thread-0] ==> current value before addition: 1176
[Thread-0] ==> current value before addition: 1178
[Thread-1] ==> current value after addition: 1180
[Thread-1] ==> current value after addition: 1181
[Thread-0] ==> current value before addition: 1181
[Thread-1] ==> current value after addition: 1183
[Thread-0] ==> current value before addition: 1183
[Thread-0] ==> current value before addition: 1184
[Thread-1] ==> current value after addition: 1186
[Thread-0] ==> current value before addition: 1186
[Thread-1] ==> current value after addition: 1188
[Thread-0] ==> current value before addition: 1189
[Thread-1] ==> current value after addition: 1189
[Thread-1] ==> current value after addition: 1191
[Thread-0] ==> current value before addition: 1191
[Thread-0] ==> current value before addition: 1193
[Thread-1] ==> current value after addition: 1193
[Thread-0] ==> current value before addition: 1194
[Thread-1] ==> current value after addition: 1196
[Thread-1] ==> current value after addition: 1197
[Thread-0] ==> current value before addition: 1197
[Thread-1] ==> current value after addition: 1199
[Thread-0] ==> current value before addition: 1199
Result: 1200
```

(iv) [CyclicBarrier]

CyclicBarrier is a class that represents a barrier and supports synchronization. This class is used for blocking multiple threads until the specific number of threads reach the barrier. When creating an instance of CyclicBarrier, you must specify how many threads should wait at this barrier. When the specified number of threads reach the barrier, then all the threads waiting at this barrier can proceed.

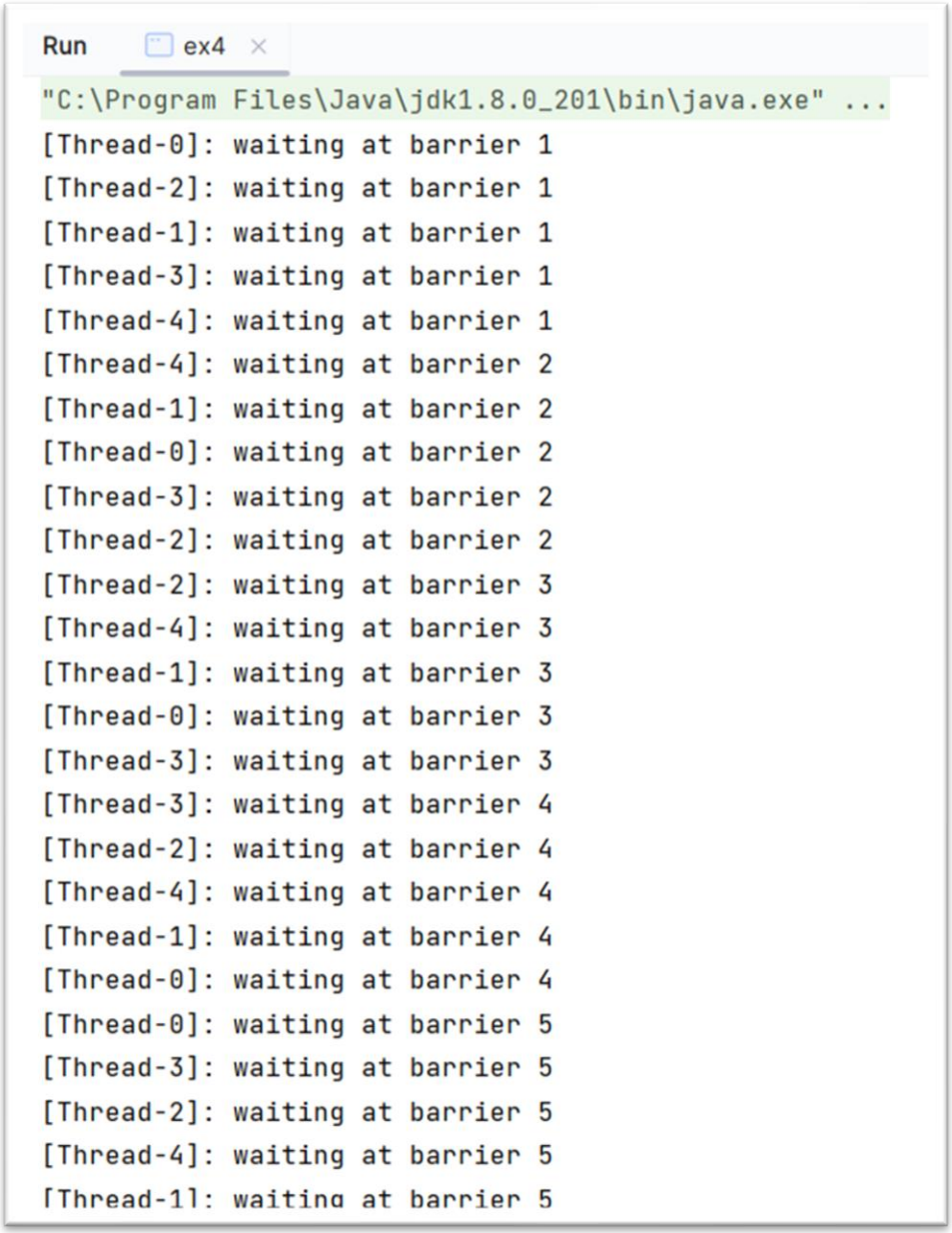
[ex4.java]

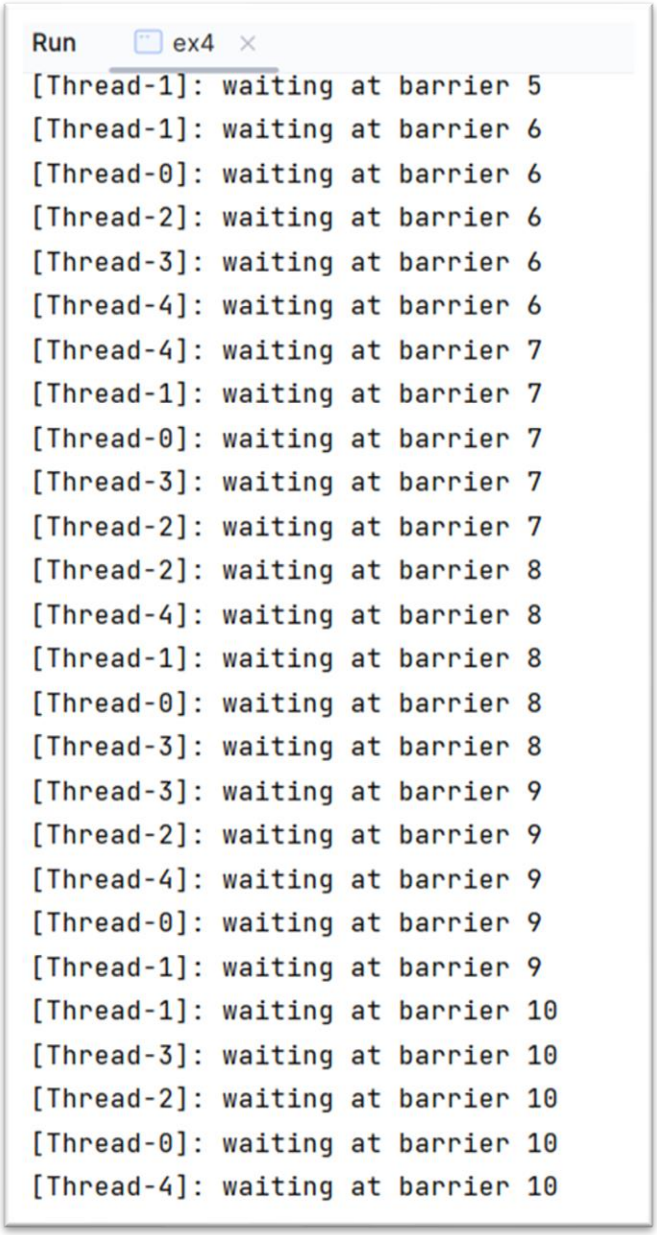
```
import java.util.concurrent.CyclicBarrier;

public class ex4 {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(5);
        MyThread[] myThreads = new MyThread[5];

        for (int i = 0; i < myThreads.length; i++) {
            myThreads[i] = new MyThread(cyclicBarrier);
        }
    }
}

final class MyThread extends Thread {
    CyclicBarrier cyclicBarrier;

    public MyThread(CyclicBarrier cyclicBarrier) {
        this.cyclicBarrier = cyclicBarrier;
        start();
    }

    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("[" + getName() + "]: waiting at barrier " +
i);
            try {
                this.cyclicBarrier.await();
            } catch (Exception ignored) {}
        }
    }
}
```

[explanation]

This source code has 5 threads. Each threads print a text. When each thread prints a text, each thread should wait until others print a text. The CyclicBarrier plays a role of making each threads wait the other threads.

[result]


Run
ex4
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
[Thread-0]: waiting at barrier 1
[Thread-2]: waiting at barrier 1
[Thread-1]: waiting at barrier 1
[Thread-3]: waiting at barrier 1
[Thread-4]: waiting at barrier 1
[Thread-4]: waiting at barrier 2
[Thread-1]: waiting at barrier 2
[Thread-0]: waiting at barrier 2
[Thread-3]: waiting at barrier 2
[Thread-2]: waiting at barrier 2
[Thread-2]: waiting at barrier 3
[Thread-4]: waiting at barrier 3
[Thread-1]: waiting at barrier 3
[Thread-0]: waiting at barrier 3
[Thread-3]: waiting at barrier 3
[Thread-3]: waiting at barrier 4
[Thread-2]: waiting at barrier 4
[Thread-4]: waiting at barrier 4
[Thread-1]: waiting at barrier 4
[Thread-0]: waiting at barrier 4
[Thread-0]: waiting at barrier 5
[Thread-3]: waiting at barrier 5
[Thread-2]: waiting at barrier 5
[Thread-4]: waiting at barrier 5
[Thread-1]: waiting at barrier 5


Run
ex4
[Thread-1]: waiting at barrier 5
[Thread-1]: waiting at barrier 6
[Thread-0]: waiting at barrier 6
[Thread-2]: waiting at barrier 6
[Thread-3]: waiting at barrier 6
[Thread-4]: waiting at barrier 6
[Thread-4]: waiting at barrier 7
[Thread-1]: waiting at barrier 7
[Thread-0]: waiting at barrier 7
[Thread-3]: waiting at barrier 7
[Thread-2]: waiting at barrier 7
[Thread-2]: waiting at barrier 8
[Thread-4]: waiting at barrier 8
[Thread-1]: waiting at barrier 8
[Thread-0]: waiting at barrier 8
[Thread-3]: waiting at barrier 8
[Thread-3]: waiting at barrier 9
[Thread-2]: waiting at barrier 9
[Thread-4]: waiting at barrier 9
[Thread-0]: waiting at barrier 9
[Thread-1]: waiting at barrier 9
[Thread-1]: waiting at barrier 10
[Thread-3]: waiting at barrier 10
[Thread-2]: waiting at barrier 10
[Thread-0]: waiting at barrier 10
[Thread-4]: waiting at barrier 10