

Assignment Day-18

Core Java with DS and Algorithms

Name: Joshnitha Rangolu

Task 1: Creating and Managing Threads

Write a program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number.

```
package day_18;

public class PrintNumbers implements Runnable{

    private int start;

    public PrintNumbers(int start) {

        this.start = start;

    }

    @Override

    public void run() {

        for (int i = start; i <= start + 9; i++) {

            System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);

            try {

                Thread.sleep(1000); // Sleep for 1 second

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

    public static void main(String[] args) {

        Thread thread1 = new Thread(new PrintNumbers(1));

        Thread thread2 = new Thread(new PrintNumbers(1));
```

```

thread1.setName("Thread-1");

thread2.setName("Thread-2");

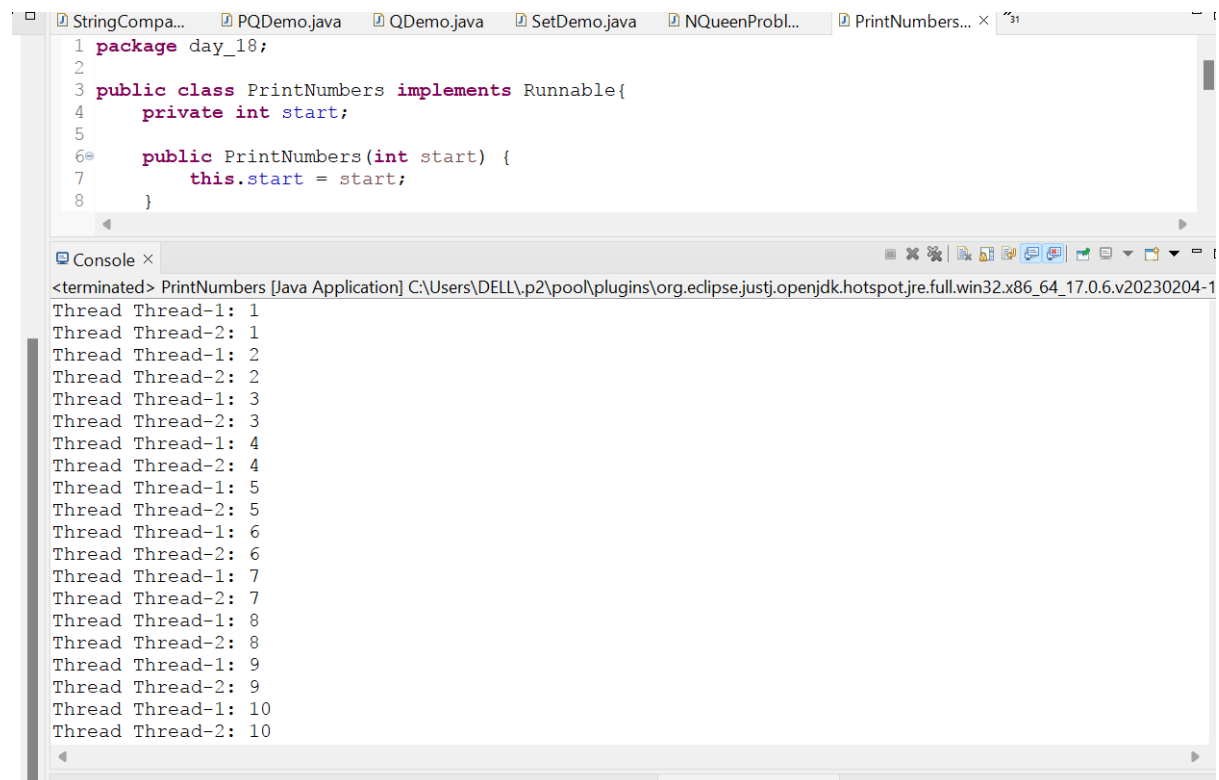
thread1.start();

thread2.start();

}

}

```



The screenshot shows the Eclipse IDE with a project named 'day_18'. The 'PrintNumbers.java' file is open, showing a class that implements the 'Runnable' interface. The class has a private integer 'start' and a 'run()' method that prints the thread name and the 'start' value. The console output shows the execution of the program, with two threads, 'Thread-1' and 'Thread-2', each printing their 'start' values from 1 to 10. The output is interleaved, showing that both threads are running concurrently.

```

1 package day_18;
2
3 public class PrintNumbers implements Runnable{
4     private int start;
5
6     public PrintNumbers(int start) {
7         this.start = start;
8     }
9
10    public void run() {
11        for (int i = 1; i <= 10; i++) {
12            System.out.println(Thread.currentThread().getName() + ": " + start);
13            try {
14                Thread.sleep(100);
15            } catch (InterruptedException e) {}
16        }
17    }
18 }

```

```

<terminated> PrintNumbers [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1
Thread Thread-1: 1
Thread Thread-2: 1
Thread Thread-1: 2
Thread Thread-2: 2
Thread Thread-1: 3
Thread Thread-2: 3
Thread Thread-1: 4
Thread Thread-2: 4
Thread Thread-1: 5
Thread Thread-2: 5
Thread Thread-1: 6
Thread Thread-2: 6
Thread Thread-1: 7
Thread Thread-2: 7
Thread Thread-1: 8
Thread Thread-2: 8
Thread Thread-1: 9
Thread Thread-2: 9
Thread Thread-1: 10
Thread Thread-2: 10

```

Task 2: States and Transitions

Create a Java class that simulates a thread going through different lifecycle states: NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCKED, and TERMINATED. Use methods like `sleep()`, `wait()`, `notify()`, and `join()` to demonstrate these states..

```

package day_18;

public class ThreadStateSimulator implements Runnable{

    private final Object lock = new Object();

    @Override

    public void run() {

```

```
System.out.println("State: NEW - Thread created");

System.out.println("State: RUNNABLE - Thread started");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (lock) {

System.out.println("State: WAITING - Waiting on lock");

try {

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("State: TIMED_WAITING - Waiting for 2 seconds");

try {

synchronized (lock) {

lock.wait(2000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

try {

System.out.println("State: BLOCKED - Waiting to call notify()");

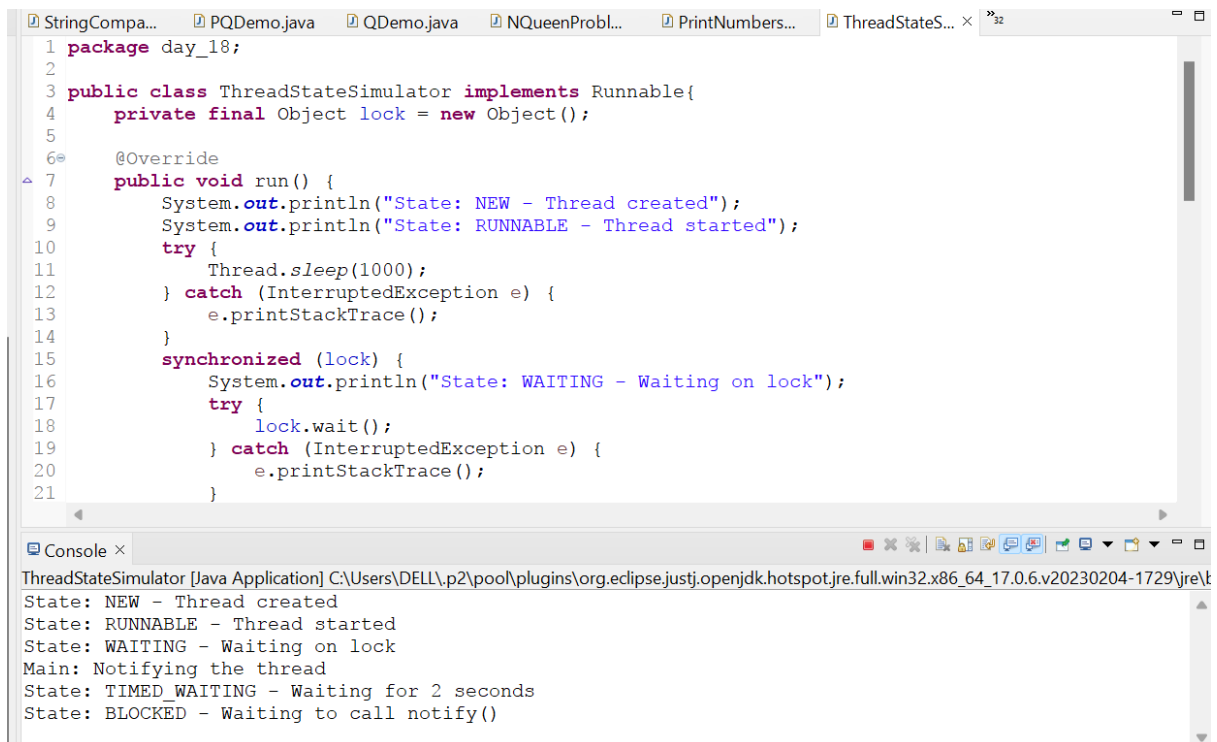
synchronized (lock) {

lock.wait();

}

}
```

```
} catch (InterruptedException e) {  
  
    e.printStackTrace();  
  
}  
  
System.out.println("State: RUNNABLE - More work after unblock");  
  
try {  
  
    Thread.sleep(500);  
  
} catch (InterruptedException e) {  
  
    e.printStackTrace();  
  
}  
  
System.out.println("State: TERMINATED - Thread completed");  
  
}  
  
public static void main(String[] args) throws InterruptedException {  
  
    ThreadStateSimulator thread = new ThreadStateSimulator();  
  
    Thread simulatorThread = new Thread(thread);  
  
    simulatorThread.start();  
  
    Thread.sleep(1500);  
  
    synchronized (thread.lock) {  
  
        System.out.println("Main: Notifying the thread");  
  
        thread.lock.notify();  
  
    }  
  
    simulatorThread.join();  
  
    System.out.println("Main: Thread joined (terminated)");  
  
}  
  
}
```



The screenshot shows the Eclipse IDE with a Java project. The editor displays the source code for `ThreadStateSimulator`, which implements the `Runnable` interface. The code uses `lock` for synchronization and `wait()` for thread communication. The console window at the bottom shows the execution output, including thread state transitions from `NEW` to `RUNNABLE`, `WAITING`, `TIMED_WAITING`, and `BLOCKED`.

```
1 package day_18;
2
3 public class ThreadStateSimulator implements Runnable{
4     private final Object lock = new Object();
5
6     @Override
7     public void run() {
8         System.out.println("State: NEW - Thread created");
9         System.out.println("State: RUNNABLE - Thread started");
10        try {
11            Thread.sleep(1000);
12        } catch (InterruptedException e) {
13            e.printStackTrace();
14        }
15        synchronized (lock) {
16            System.out.println("State: WAITING - Waiting on lock");
17            try {
18                lock.wait();
19            } catch (InterruptedException e) {
20                e.printStackTrace();
21            }
22        }
23    }
24 }
```

Console Output:

```
ThreadStateSimulator [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\k
State: NEW - Thread created
State: RUNNABLE - Thread started
State: WAITING - Waiting on lock
Main: Notifying the thread
State: TIMED_WAITING - Waiting for 2 seconds
State: BLOCKED - Waiting to call notify()
```

Task 3: Synchronization and Inter-thread Communication

Implement a producer-consumer problem using `wait()` and `notify()` methods to handle the correct processing sequence between threads.

```
package day_18;

public class ProducerConsumer {

    private static final int BUFFER_SIZE = 5;

    private int[] buffer = new int[BUFFER_SIZE];

    private int in = 0, out = 0, count = 0;

    public synchronized void produce(int item) throws InterruptedException {

        while (count == BUFFER_SIZE) {

            wait();

        }

        buffer[in] = item;

        in = (in + 1) % BUFFER_SIZE;

        count++;
    }
}
```

```

notify();

System.out.println("Produced: " + item);

}

public synchronized int consume() throws InterruptedException {

while (count == 0) {

wait();

}

int item = buffer[out];

out = (out + 1) % BUFFER_SIZE;

count--;

notify();

System.out.println("Consumed: " + item);

return item;

}

public static void main(String[] args) throws InterruptedException {

ProducerConsumer pc = new ProducerConsumer();

Thread producer = new Thread(() -> {

for (int i = 1; i <= 10; i++) {

try {

pc.produce(i);

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

Thread consumer = new Thread(() -> {

for (int i = 1; i <= 10; i++) {

```

```

try {

pc.consume();

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

producer.start();

consumer.start();

producer.join();

consumer.join();

}

}

```

The screenshot shows the Eclipse IDE interface. The top part displays the source code of a Java class named `ProducerConsumer` in the file `ProducerCons...`. The code defines a package `day_18`, a public class `ProducerConsumer`, and its internal state: a static final `BUFFER_SIZE` of 5, a private `int[] buffer` of size `BUFFER_SIZE`, and private `int` variables `in`, `out`, and `count`, all initialized to 0.

The bottom part of the screenshot shows the `Console` window. It displays the output of the application, which consists of alternating lines of "Produced: X" and "Consumed: X" for X ranging from 1 to 10. The output ends with the message `<terminated>`. The console title is `Console ×` and the application path is `C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v202302`.

Task 4: Synchronized Blocks and Methods

Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

```
package day_18;

public class BankAccount {

    private int balance = 0;

    public synchronized void deposit(int amount) {

        balance += amount;

        System.out.println("Deposited: " + amount + ", New Balance: " + balance);

    }

    public synchronized void withdraw(int amount) throws InterruptedException {

        if (balance < amount) {

            System.out.println("Insufficient balance. Waiting for deposit...");

            wait();

        }

        balance -= amount;

        System.out.println("Withdrew: " + amount + ", New Balance: " + balance);

    }

    public static void main(String[] args) throws InterruptedException {

        BankAccount account = new BankAccount();

        Thread depositor = new Thread(() -> {

            for (int i = 0; i < 10; i++) {

                account.deposit(100);

                try {

                    Thread.sleep(1000);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        });

    }

}
```



```
}

});

Thread withdrawer = new Thread(() -> {

    for (int i = 0; i < 10; i++) {

        try {

            account.withdraw(50);

            Thread.sleep(500);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

});

depositor.start();

withdrawer.start();

depositor.join();

withdrawer.join();

}

}
```

The screenshot shows an Eclipse IDE with several tabs open. The active tab is `*BankAccount...`, which contains the following Java code:

```
1 package day_18;
2
3 public class BankAccount {
4     private int balance = 0;
5
6     public synchronized void deposit(int amount) {
7         balance += amount;
8         System.out.println("Deposited: " + amount + ", New Balance: " + balance);
9     }
10
11     public synchronized void withdraw(int amount) {
12         if (balance >= amount) {
13             balance -= amount;
14             System.out.println("Withdrew: " + amount + ", New Balance: " + balance);
15         } else {
16             System.out.println("Insufficient funds for withdrawal of " + amount);
17         }
18     }
19 }
20
```

Below the code editor is the `Console` window, which displays the output of the program. The output shows a sequence of deposit and withdrawal operations, with the balance being updated accordingly. The console output is as follows:

```
<terminated> BankAccount [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-
Deposited: 100, New Balance: 100
Withdrew: 50, New Balance: 50
Withdrew: 50, New Balance: 0
Deposited: 100, New Balance: 100
Withdrew: 50, New Balance: 50
Withdrew: 50, New Balance: 0
Deposited: 100, New Balance: 100
Withdrew: 50, New Balance: 50
Withdrew: 50, New Balance: 0
Deposited: 100, New Balance: 100
Withdrew: 50, New Balance: 50
Withdrew: 50, New Balance: 0
Deposited: 100, New Balance: 100
Withdrew: 50, New Balance: 50
Withdrew: 50, New Balance: 0
Deposited: 100, New Balance: 100
Deposited: 100, New Balance: 200
Deposited: 100, New Balance: 300
Deposited: 100, New Balance: 400
Deposited: 100, New Balance: 500
```

Task 5: Thread Pools and Concurrency Utilities

Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

```
package day_18;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class FixedThreadPoolDemo {

    public static void main(String[] args) throws InterruptedException {

        int poolSize = 3;

        ExecutorService executor = Executors.newFixedThreadPool(poolSize);

        Runnable task = () -> {

            long startTime = System.currentTimeMillis();

            System.out.println("Thread " + Thread.currentThread().getName() + " started calculation");
        }
    }
}
```

```

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

long endTime = System.currentTimeMillis();

System.out.println("Thread " + Thread.currentThread().getName() + "
finished calculation in " + (endTime - startTime) + " ms");

};

for (int i = 0; i < 10; i++) {

executor.submit(task);

}

executor.shutdown();

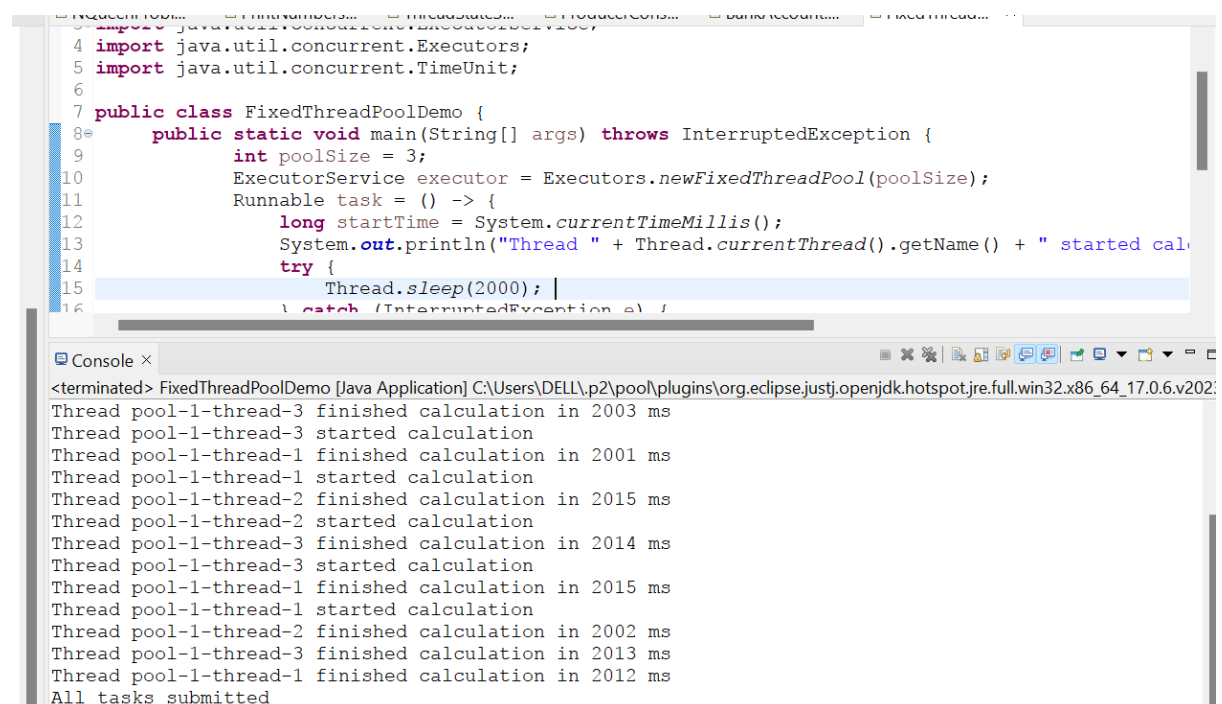
executor.awaitTermination(1, TimeUnit.MINUTES);

System.out.println("All tasks submitted");

}

}

```



The screenshot shows the Eclipse IDE with a Java project. The editor displays the source code for `FixedThreadPoolDemo.java`, which imports `java.util.concurrent.Executors` and `java.util.concurrent.TimeUnit`. The `main` method creates a `FixedThreadPool` with a size of 3, then submits 10 `Runnable` tasks. Each task sleeps for 2000 milliseconds before printing its thread name and the time taken for the calculation. The console output shows the execution of these tasks, with messages like "Thread pool-1-thread-3 finished calculation in 2003 ms" and "Thread pool-1-thread-3 started calculation". The output is interleaved, reflecting the concurrent execution of the tasks. The console also shows the final message "All tasks submitted".

```

4 import java.util.concurrent.Executors;
5 import java.util.concurrent.TimeUnit;
6
7 public class FixedThreadPoolDemo {
8     public static void main(String[] args) throws InterruptedException {
9         int poolSize = 3;
10        ExecutorService executor = Executors.newFixedThreadPool(poolSize);
11        Runnable task = () -> {
12            long startTime = System.currentTimeMillis();
13            System.out.println("Thread " + Thread.currentThread().getName() + " started calculation");
14            try {
15                Thread.sleep(2000);
16            } catch (InterruptedException e) {
17            }
18        };
19        for (int i = 0; i < 10; i++) {
20            executor.submit(task);
21        }
22        executor.shutdown();
23        executor.awaitTermination(1, TimeUnit.MINUTES);
24        System.out.println("All tasks submitted");
25    }
26 }

```

```

<terminated> FixedThreadPoolDemo [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v2023-01-17\jre\bin\java.exe
Thread pool-1-thread-3 finished calculation in 2003 ms
Thread pool-1-thread-3 started calculation
Thread pool-1-thread-1 finished calculation in 2001 ms
Thread pool-1-thread-1 started calculation
Thread pool-1-thread-2 finished calculation in 2015 ms
Thread pool-1-thread-2 started calculation
Thread pool-1-thread-3 finished calculation in 2014 ms
Thread pool-1-thread-3 started calculation
Thread pool-1-thread-1 finished calculation in 2015 ms
Thread pool-1-thread-1 started calculation
Thread pool-1-thread-2 finished calculation in 2002 ms
Thread pool-1-thread-3 finished calculation in 2013 ms
Thread pool-1-thread-1 finished calculation in 2012 ms
All tasks submitted

```

Task 6: Executors, Concurrent Collections, CompletableFuture

Use an `ExecutorService` to parallelize a task that calculates prime numbers up to a given number and then use `CompletableFuture` to write the results to a file asynchronously.

```
package day_18;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class PrimeNumberParallel {

    private static final int MAX_NUMBER = 1000;

    public static boolean isPrime(int num) {

        if (num <= 1) return false;

        if (num <= 3) return true;

        if (num % 2 == 0 || num % 3 == 0) return false;

        for (int i = 5; i * i <= num; i += 6) {

            if (num % i == 0 || num % (i + 2) == 0) {

                return false;

            }

        }

        return true;

    }

    public static void main(String[] args) throws IOException,
        InterruptedException {

        int numThreads = Runtime.getRuntime().availableProcessors();

        ExecutorService executor = Executors.newFixedThreadPool(numThreads);

        ArrayList<Integer> primes = new ArrayList<>();
```

```

CompletableFuture<Void> task = CompletableFuture.runAsync(() -> {

    for (int i = 2; i <= MAX_NUMBER; i++) {

        if (isPrime(i)) {

            primes.add(i);

        }

    }

    }, executor);

task.thenAcceptAsync(unused -> {

    try (FileWriter writer = new FileWriter("primes.txt")) {

        for (int prime : primes) {

            writer.write(prime + "\n");

        }

        writer.flush();

        System.out.println("Primes written to primes.txt");

    } catch (IOException e) {

        e.printStackTrace();

    }

    }, executor);

executor.shutdown();

executor.awaitTermination(1, TimeUnit.MINUTES);

System.out.println("Prime number calculation finished");

}

}

```



```
counter.increment();

}

});

Thread decrementer = new Thread(() -> {

for (int i = 0; i < 1000; i++) {

counter.decrement();

}

});

incrementer1.start();

incrementer2.start();

decrementer.start();

incrementer1.join();

incrementer2.join();

decrementer.join();

System.out.println("Final count: " + counter.getCount());

}

}

class Counter {

private final AtomicInteger count = new AtomicInteger(0);

public void increment() {

count.incrementAndGet();

}

public void decrement() {

count.decrementAndGet();

}

public int getCount() {

return count.get();

}

}
```

```

}

class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

}

```

The screenshot shows an IDE with a Java file named `ThreadSafeCounter.java`. The code defines a package `day_18`, imports `java.util.concurrent.atomic.AtomicInteger`, and defines a public class `ThreadSafeCounter`. The `main` method creates a `Counter` object and three threads: `incrementer1`, `incrementer2`, and `decrementer`. `incrementer1` and `incrementer2` both increment the counter 1000 times, while `decrementer` decrements it 1000 times. The console output shows the program terminated successfully with a final count of 1000.

```

1 package day_18;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 public class ThreadSafeCounter {
6     public static void main(String[] args) throws InterruptedException {
7         Counter counter = new Counter();
8         Thread incrementer1 = new Thread(() -> {
9             for (int i = 0; i < 1000; i++) {
10                 counter.increment();
11             }
12         });
13
14         Thread incrementer2 = new Thread(() -> {
15             for (int i = 0; i < 1000; i++) {
16                 counter.increment();
17             }
18         });
19
20         Thread decrementer = new Thread(() -> {
21             for (int i = 0; i < 1000; i++) {
22                 counter.decrement();
23             }
24         });
25
26         incrementer1.start();

```

Console ×

<terminated> ThreadSafeCounter [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v2023
Final count: 1000

