

DAY-18 JAVA ASSIGNMENT

Day 18:

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



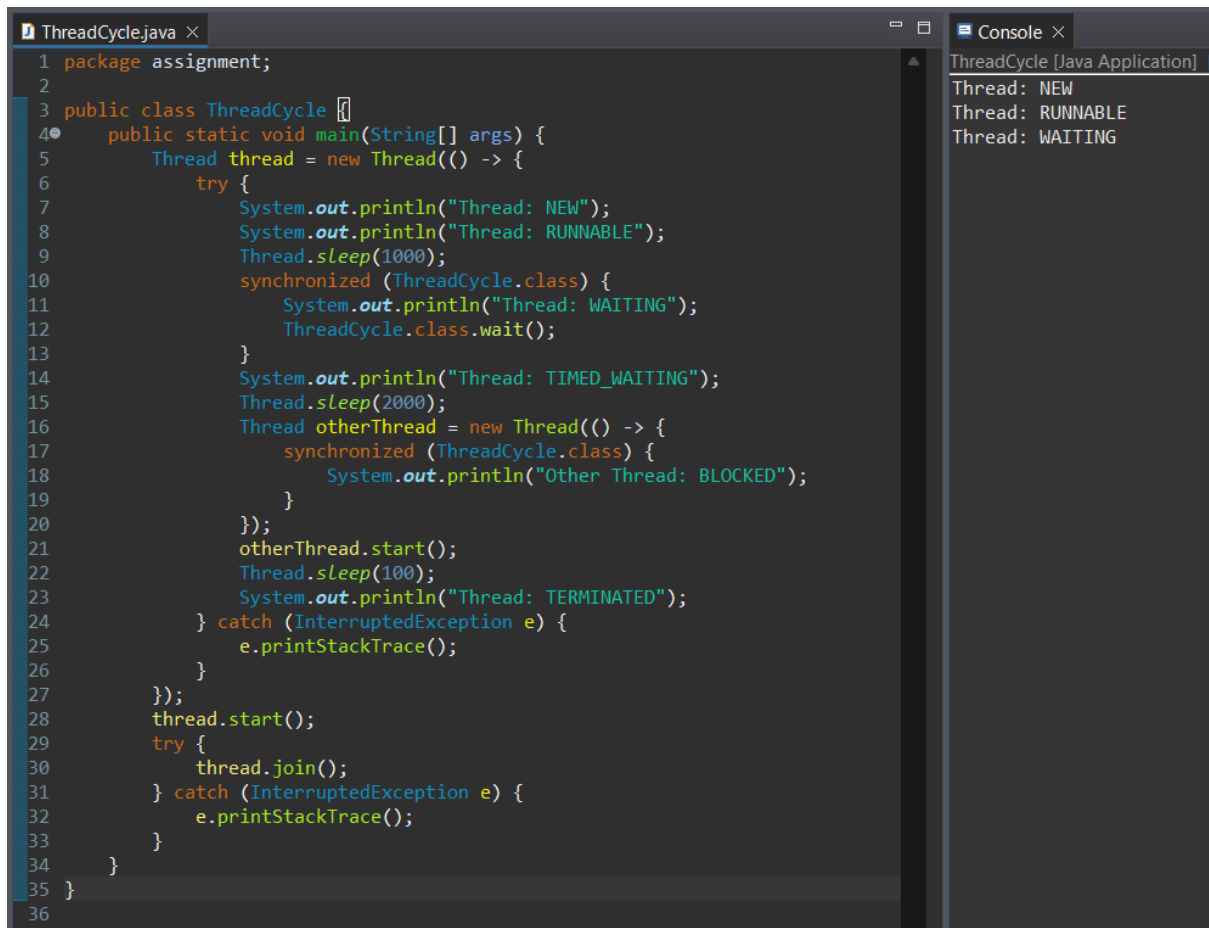
```
1 package assignment;
2
3 class ThreadA extends Thread {
4     public void run() {
5         for (int i = 1; i <= 10; i++) {
6             System.out.println(i);
7             try {
8                 Thread.sleep(1000);
9             } catch (InterruptedException e) {
10                e.printStackTrace();
11            }
12        }
13    }
14 }
15
16
17 public class CreatingThreads {
18     public static void main(String args[]) {
19         ThreadA thread1 = new ThreadA();
20         ThreadA thread2 = new ThreadA();
21         thread1.run();
22         thread2.run();
23     }
24 }
```

Console Output:

```
<terminated> Cr
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
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..



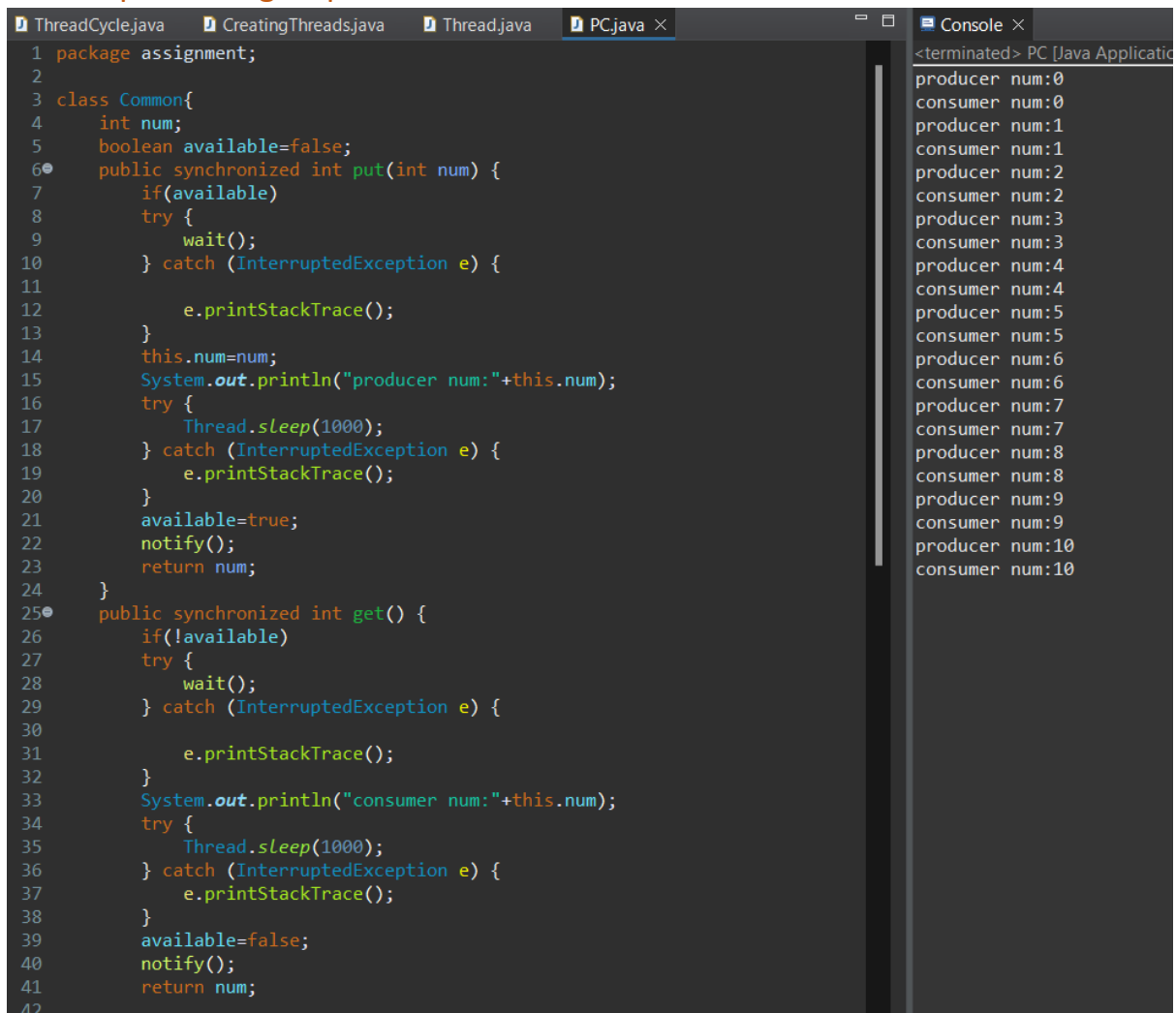
The screenshot shows an IDE with two panels. The left panel displays the source code for `ThreadCycle.java`, and the right panel shows the console output.

```
1 package assignment;
2
3 public class ThreadCycle {
4     public static void main(String[] args) {
5         Thread thread = new Thread(() -> {
6             try {
7                 System.out.println("Thread: NEW");
8                 System.out.println("Thread: RUNNABLE");
9                 Thread.sleep(1000);
10                synchronized (ThreadCycle.class) {
11                    System.out.println("Thread: WAITING");
12                    ThreadCycle.class.wait();
13                }
14                System.out.println("Thread: TIMED_WAITING");
15                Thread.sleep(2000);
16                Thread otherThread = new Thread(() -> {
17                    synchronized (ThreadCycle.class) {
18                        System.out.println("Other Thread: BLOCKED");
19                    }
20                });
21                otherThread.start();
22                Thread.sleep(100);
23                System.out.println("Thread: TERMINATED");
24            } catch (InterruptedException e) {
25                e.printStackTrace();
26            }
27        });
28        thread.start();
29        try {
30            thread.join();
31        } catch (InterruptedException e) {
32            e.printStackTrace();
33        }
34    }
35 }
36
```

The console output on the right shows the following sequence of states:

```
ThreadCycle [Java Application]
Thread: NEW
Thread: RUNNABLE
Thread: WAITING
```

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.



The screenshot shows an IDE with two windows. The left window, titled 'PC.java', contains the following Java code:

```
1 package assignment;
2
3 class Common{
4     int num;
5     boolean available=false;
6     public synchronized int put(int num) {
7         if(available)
8             try {
9                 wait();
10            } catch (InterruptedException e) {
11                e.printStackTrace();
12            }
13            this.num=num;
14            System.out.println("producer num:"+this.num);
15            try {
16                Thread.sleep(1000);
17            } catch (InterruptedException e) {
18                e.printStackTrace();
19            }
20            available=true;
21            notify();
22            return num;
23        }
24    }
25    public synchronized int get() {
26        if(!available)
27            try {
28                wait();
29            } catch (InterruptedException e) {
30                e.printStackTrace();
31            }
32            System.out.println("consumer num:"+this.num);
33            try {
34                Thread.sleep(1000);
35            } catch (InterruptedException e) {
36                e.printStackTrace();
37            }
38            available=false;
39            notify();
40            return num;
41        }
42    }
```

The right window, titled 'Console', shows the output of the program:

```
<terminated> PC [Java Applicatio
producer num:0
consumer num:0
producer num:1
consumer num:1
producer num:2
consumer num:2
producer num:3
consumer num:3
producer num:4
consumer num:4
producer num:5
consumer num:5
producer num:6
consumer num:6
producer num:7
consumer num:7
producer num:8
consumer num:8
producer num:9
consumer num:9
producer num:10
consumer num:10
```

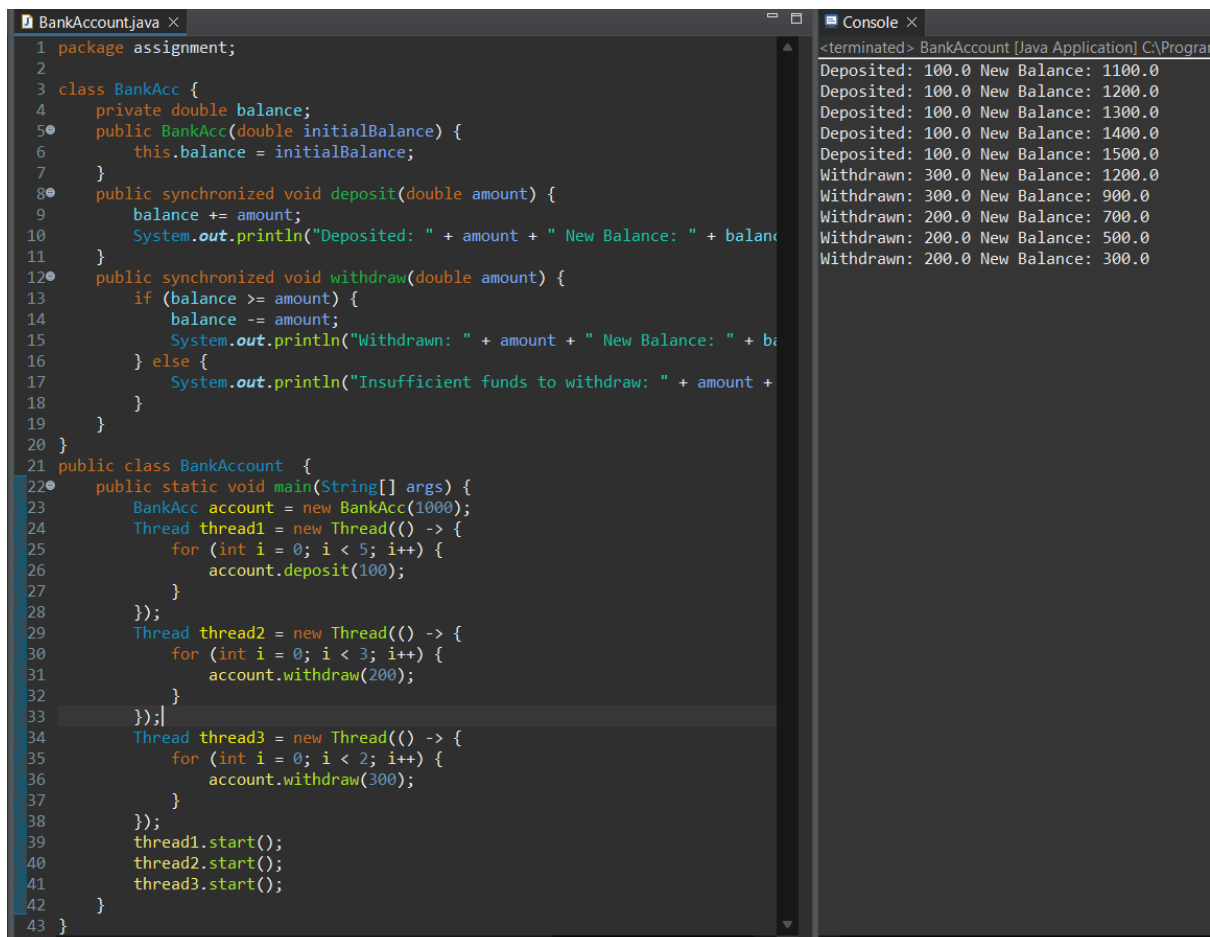
```
ThreadCycle.java  CreatingThreads.java  Thread.java  PC.java x
40     notify();
41     return num;
42
43 }
44 }
45 class Producer extends Thread{
46     Common c;
47     public Producer(Common c) {
48         this.c=c;
49         new Thread(this,"prod").start();
50     }
51     public void run() {
52         int x=0,i=0;
53         while(x<=10) {
54             c.put(i++);
55             x++;
56         }
57     }
58 }
59 class Consumer extends Thread{
60     Common c;
61     public Consumer(Common c) {
62         this.c=c;
63         new Thread(this,"Consumer").start();
64     }
65     public void run() {
66         int x=0;
67         while(x<=10) {
68             c.get();
69             x++;
70         }
71     }
72 }
73 }
74 public class PC {
75     public static void main(String[] args) {
76         Common c=new Common();
77         new Producer(c);
78         new Consumer(c);
79     }
80 }
81 }
82 }
```

<terminated> PC [Java Applic

producer num:0
consumer num:0
producer num:1
consumer num:1
producer num:2
consumer num:2
producer num:3
consumer num:3
producer num:4
consumer num:4
producer num:5
consumer num:5
producer num:6
consumer num:6
producer num:7
consumer num:7
producer num:8
consumer num:8
producer num:9
consumer num:9
producer num:10
consumer num:10

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.



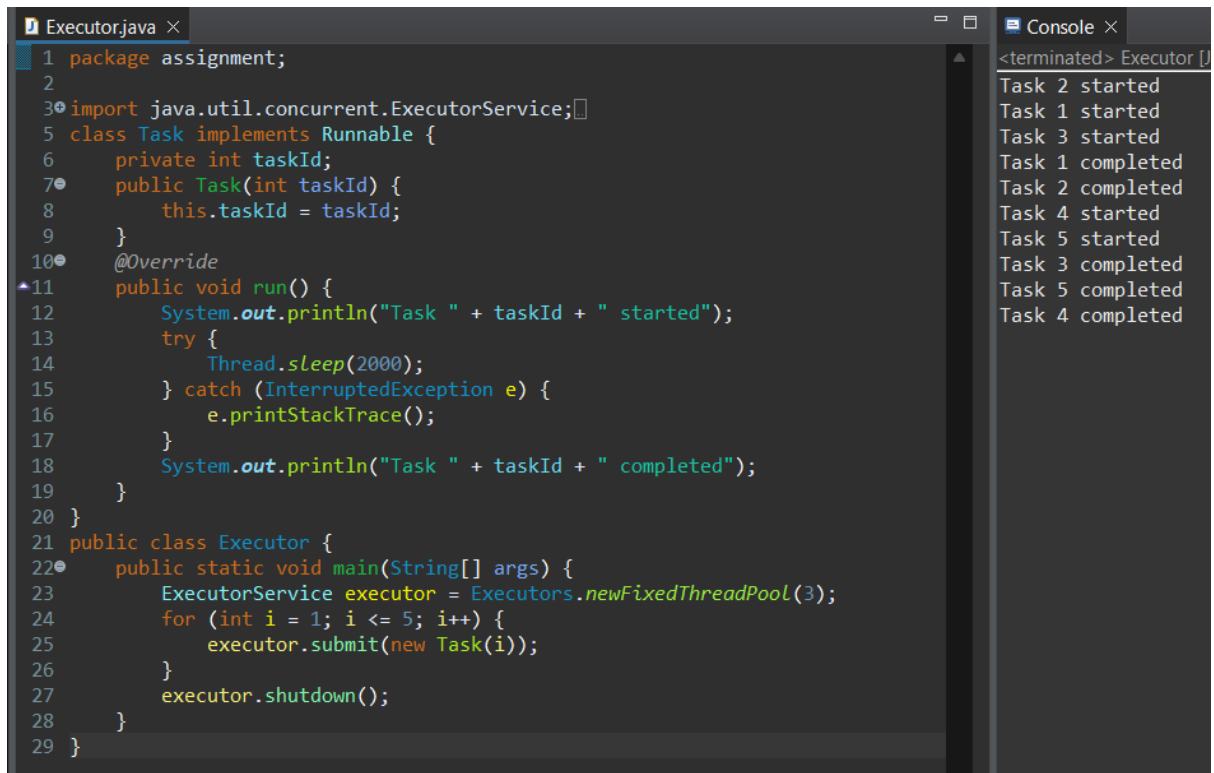
```
1 package assignment;
2
3 class BankAcc {
4     private double balance;
5     public BankAcc(double initialBalance) {
6         this.balance = initialBalance;
7     }
8     public synchronized void deposit(double amount) {
9         balance += amount;
10        System.out.println("Deposited: " + amount + " New Balance: " + balance);
11    }
12    public synchronized void withdraw(double amount) {
13        if (balance >= amount) {
14            balance -= amount;
15            System.out.println("Withdrawn: " + amount + " New Balance: " + balance);
16        } else {
17            System.out.println("Insufficient funds to withdraw: " + amount + " Current Balance: " + balance);
18        }
19    }
20 }
21 public class BankAccount {
22     public static void main(String[] args) {
23         BankAcc account = new BankAcc(1000);
24         Thread thread1 = new Thread(() -> {
25             for (int i = 0; i < 5; i++) {
26                 account.deposit(100);
27             }
28         });
29         Thread thread2 = new Thread(() -> {
30             for (int i = 0; i < 3; i++) {
31                 account.withdraw(200);
32             }
33         });
34         Thread thread3 = new Thread(() -> {
35             for (int i = 0; i < 2; i++) {
36                 account.withdraw(300);
37             }
38         });
39         thread1.start();
40         thread2.start();
41         thread3.start();
42     }
43 }
```

Console Output:

```
<terminated> BankAccount [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\java.exe
Deposited: 100.0 New Balance: 1100.0
Deposited: 100.0 New Balance: 1200.0
Deposited: 100.0 New Balance: 1300.0
Deposited: 100.0 New Balance: 1400.0
Deposited: 100.0 New Balance: 1500.0
Withdrawn: 300.0 New Balance: 1200.0
Withdrawn: 300.0 New Balance: 900.0
Withdrawn: 200.0 New Balance: 700.0
Withdrawn: 200.0 New Balance: 500.0
Withdrawn: 200.0 New Balance: 300.0
```

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.



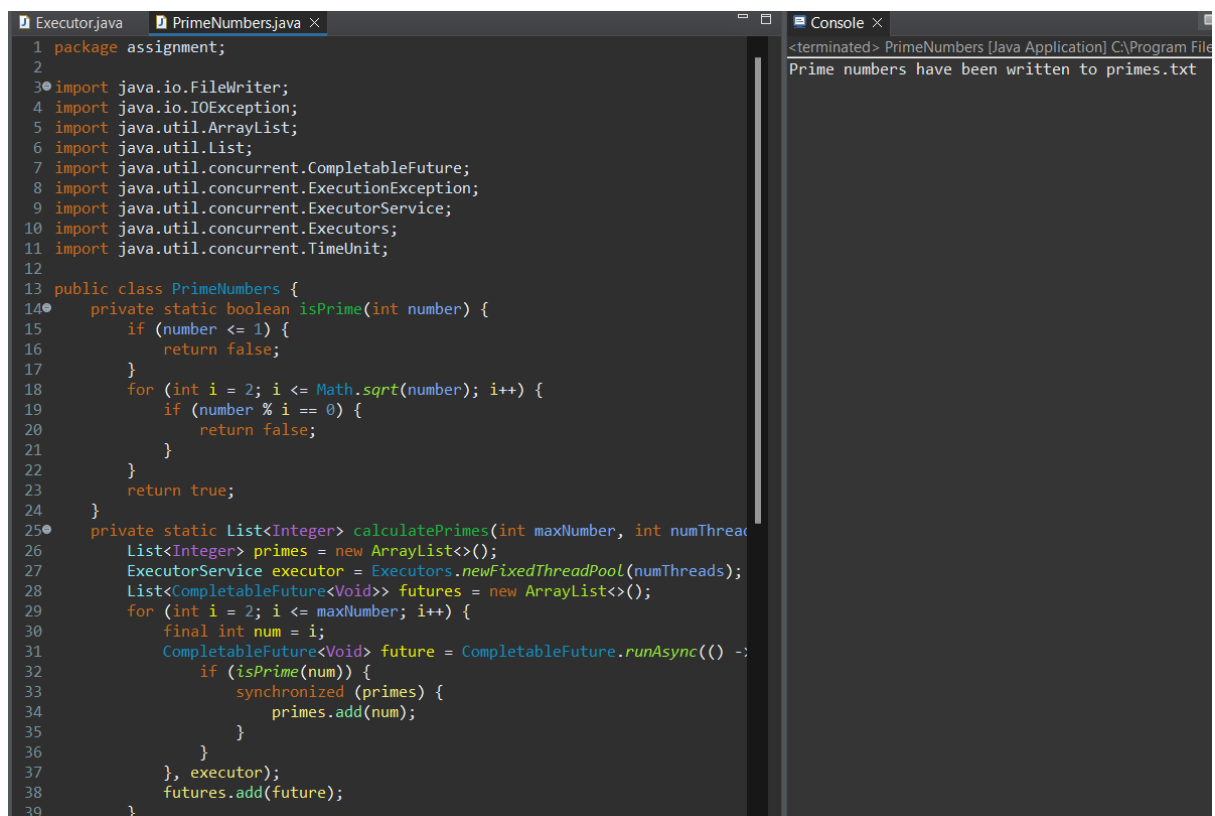
```
1 package assignment;
2
3 import java.util.concurrent.ExecutorService;
4
5 class Task implements Runnable {
6     private int taskId;
7     public Task(int taskId) {
8         this.taskId = taskId;
9     }
10    @Override
11    public void run() {
12        System.out.println("Task " + taskId + " started");
13        try {
14            Thread.sleep(2000);
15        } catch (InterruptedException e) {
16            e.printStackTrace();
17        }
18        System.out.println("Task " + taskId + " completed");
19    }
20 }
21 public class Executor {
22    public static void main(String[] args) {
23        ExecutorService executor = Executors.newFixedThreadPool(3);
24        for (int i = 1; i <= 5; i++) {
25            executor.submit(new Task(i));
26        }
27        executor.shutdown();
28    }
29 }
```

Console Output:

```
<terminated> Executor [J
Task 2 started
Task 1 started
Task 3 started
Task 1 completed
Task 2 completed
Task 4 started
Task 5 started
Task 3 completed
Task 5 completed
Task 4 completed
```

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.



```
1 package assignment;
2
3 import java.io.FileWriter;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import java.util.List;
7 import java.util.concurrent.CompletableFuture;
8 import java.util.concurrent.ExecutionException;
9 import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.TimeUnit;
12
13 public class PrimeNumbers {
14     private static boolean isPrime(int number) {
15         if (number <= 1) {
16             return false;
17         }
18         for (int i = 2; i <= Math.sqrt(number); i++) {
19             if (number % i == 0) {
20                 return false;
21             }
22         }
23         return true;
24     }
25     private static List<Integer> calculatePrimes(int maxNumber, int numThreads) {
26         List<Integer> primes = new ArrayList<>();
27         ExecutorService executor = Executors.newFixedThreadPool(numThreads);
28         List<CompletableFuture<Void>> futures = new ArrayList<>();
29         for (int i = 2; i <= maxNumber; i++) {
30             final int num = i;
31             CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
32                 if (isPrime(num)) {
33                     synchronized (primes) {
34                         primes.add(num);
35                     }
36                 }
37             }, executor);
38             futures.add(future);
39         }
40     }
41 }
```

<terminated> PrimeNumbers [Java Application] C:\Program Files\Java\jdk-11.0.10\bin\java.exe
Prime numbers have been written to primes.txt

```
Executor.java PrimeNumbers.java Console
34         primes.add(num);
35     }
36 }
37 }, executor);
38 futures.add(future);
39 }
40 CompletableFuture<Void> allFutures = CompletableFuture.allOf(futures);
41 try {
42     allFutures.get();
43 } catch (InterruptedException e) {
44     e.printStackTrace();
45 } catch (ExecutionException e) {
46     e.printStackTrace();
47 }
48 executor.shutdown();
49 executor.awaitTermination(1, TimeUnit.MINUTES);
50
51 return primes;
52 }
53 private static CompletableFuture<Void> writeToFileAsync(List<Integer> primes, String filename) {
54     return CompletableFuture.runAsync(() -> {
55         try (FileWriter writer = new FileWriter(filename)) {
56             for (Integer prime : primes) {
57                 writer.write(prime + "\n");
58             }
59             System.out.println("Prime numbers have been written to " + filename);
60         } catch (IOException e) {
61             e.printStackTrace();
62         }
63     });
64 }
65 public static void main(String[] args) {
66     int maxNumber = 50;
67     int numThreads = Runtime.getRuntime().availableProcessors();
68     try {
69         List<Integer> primes = calculatePrimes(maxNumber, numThreads);
70         CompletableFuture<Void> fileWriteFuture = writeToFileAsync(primes, "primes.txt");
71         fileWriteFuture.get();
72     } catch (Exception e) {
73         e.printStackTrace();
74     }
75 }
76 }
```

<terminated> PrimeNumbers [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\java.exe
Prime numbers have been written to primes.txt

```
File Edit View
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```


Task 7: Writing Thread-Safe Code, Immutable Objects

Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

```
1 package assignment;
2
3 class Counter {
4     private int count = 0;
5     public synchronized void increment() {
6         count++;
7     }
8     public synchronized void decrement() {
9         count--;
10    }
11    public synchronized int getCount() {
12        return count;
13    }
14 }
15
16 final class ImmutableData {
17     private final int value;
18     public ImmutableData(int value) {
19         this.value = value;
20     }
21     public int getValue() {
22         return value;
23     }
24 }
25
26 public class ThreadSafe {
27     public static void main(String[] args) {
28         Counter counter = new Counter();
29         Runnable incrementTask = () -> {
30             for (int i = 0; i < 1000; i++) {
31                 counter.increment();
32             }
33         };
34         Runnable decrementTask = () -> {
35             for (int i = 0; i < 1000; i++) {
36                 counter.decrement();
37             }
38         };
39         Thread incrementThread = new Thread(incrementTask);
40         Thread decrementThread = new Thread(decrementTask);
41         incrementThread.start();
42         decrementThread.start();
43         try {
44             incrementThread.join();
45             decrementThread.join();
46         } catch (InterruptedException e) {
47             Thread.currentThread().interrupt();
48         }
49         System.out.println("Counter value: " + counter.getCount());
50         ImmutableData data = new ImmutableData(10);
51         System.out.println("Immutable data value: " + data.getValue());
52     }
53 }
```

<terminated> ThreadSafe [Java Applet]
Counter value: 0
Immutable data value: 10
Immutable data value: 10

```
37     Thread incrementThread = new Thread(incrementTask);
38     Thread decrementThread = new Thread(decrementTask);
39     incrementThread.start();
40     decrementThread.start();
41     try {
42         incrementThread.join();
43         decrementThread.join();
44     } catch (InterruptedException e) {
45         e.printStackTrace();
46     }
47     System.out.println("Counter value: " + counter.getCount());
48     ImmutableData immutableData = new ImmutableData(10);
49     Runnable readTask = () -> {
50         System.out.println("Immutable data value: " + immutableData.get\
51     };
52     Thread readThread1 = new Thread(readTask);
53     Thread readThread2 = new Thread(readTask);
54     readThread1.start();
55     readThread2.start();
56 }
57 }
```

