

CMP3001 Operating Systems

Project (Synchronization)

The Reader Writer Problem

Overview

To demonstrate a read-write lock mechanism:

- ✓ It showcases how to manage concurrent access to shared data in a multithreaded environment using a read-write lock. This mechanism allows multiple threads to read data simultaneously (improving performance) while ensuring exclusive access for writing operations to maintain data integrity.

To illustrate semaphore usage for synchronization:

- ✓ It demonstrates the use of semaphores, a synchronization tool, to control access to the shared data. Semaphores act as gatekeepers, ensuring that only permitted threads can access the data at specific times, preventing race conditions and data corruption.

To simulate reader and writer threads:

- ✓ It creates multiple reader and writer threads to mimic real-world scenarios where multiple entities might access shared data concurrently. This allows observation of how the read-write lock manages their interactions and prevents conflicts.

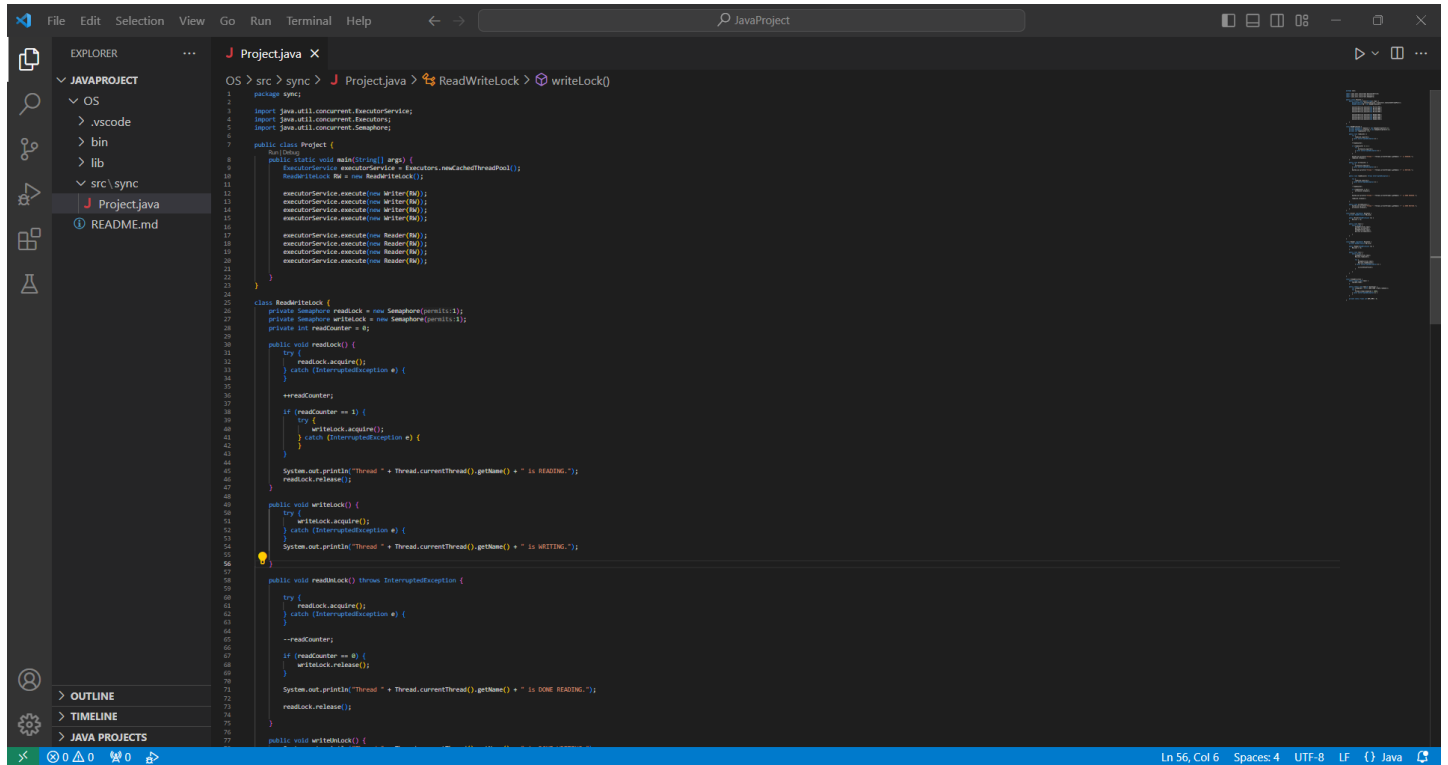
To provide a basic example for educational purposes:

- ✓ The code serves as a simplified model for understanding the concept of read-write locks and their implementation using semaphores. It's often used in tutorials and learning materials for concurrency in Java.

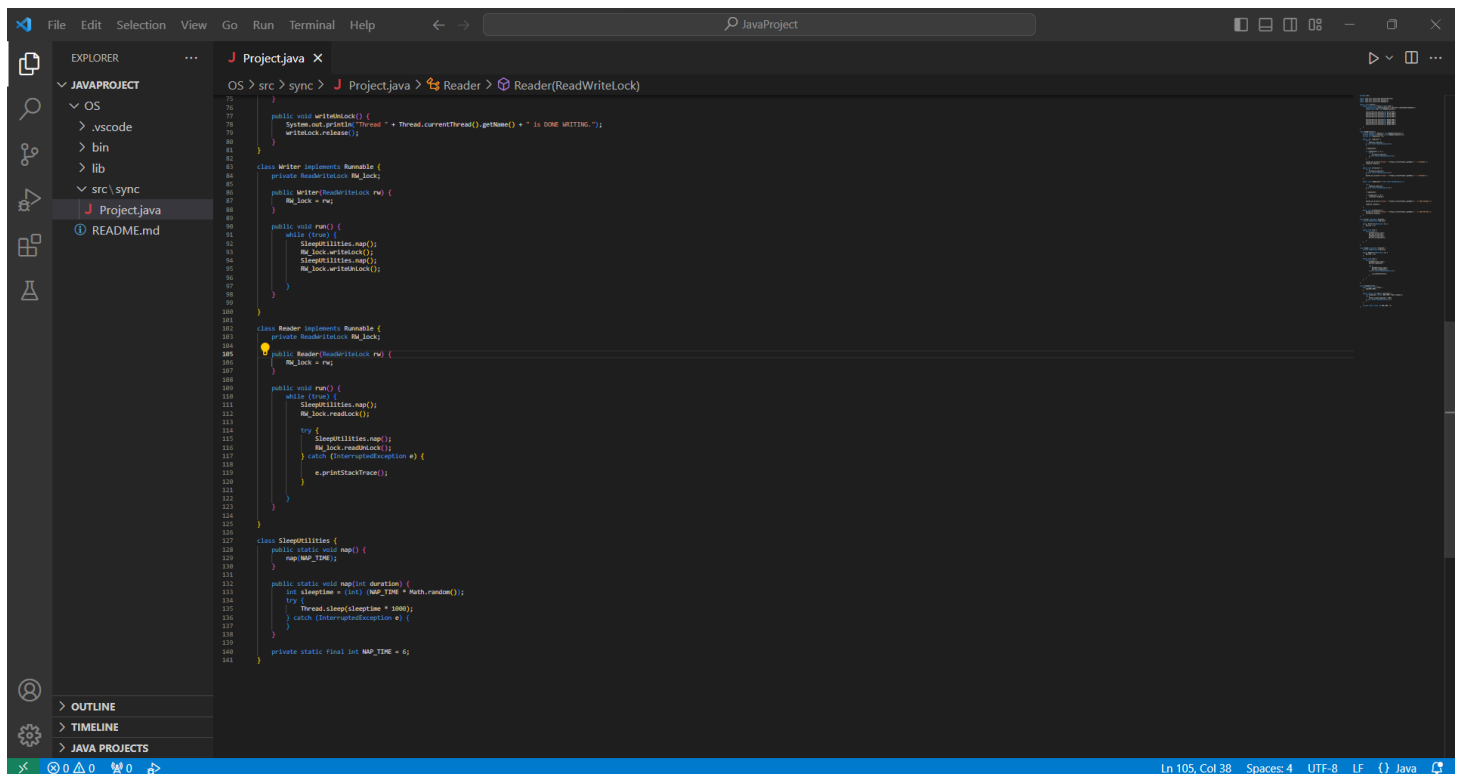
To explore potential performance benefits:

- ✓ It allows experimentation with different lock strategies and observation of their impact on performance. Read-write locks can potentially improve performance in scenarios with frequent reads and less frequent writes, as multiple readers can access data concurrently without blocking each other.

Codes (Visual Studio Code)



```
1 package sync;
2
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executor;
5 import java.util.concurrent.Semaphore;
6
7 public class Project {
8     public static void main(String[] args) {
9         ExecutorService executorService = Executors.newCachedThreadPool();
10         ReaderLock rl = new ReaderLock();
11         executorService.execute(new Writer(rl));
12         executorService.execute(new Writer(rl));
13         executorService.execute(new Writer(rl));
14         executorService.execute(new Writer(rl));
15         executorService.execute(new Reader(rl));
16         executorService.execute(new Reader(rl));
17         executorService.execute(new Reader(rl));
18         executorService.execute(new Reader(rl));
19         executorService.execute(new Reader(rl));
20     }
21 }
22
23 class ReaderLock {
24     private Semaphore readLock = new Semaphore(permits);
25     private Semaphore writeLock = new Semaphore(permits);
26     private int readCounter = 0;
27
28     public void readLock() {
29         try {
30             readLock.acquire();
31         } catch (InterruptedException e) {
32             e.printStackTrace();
33         }
34         ++readCounter;
35
36         if (readCounter == 1) {
37             try {
38                 writeLock.acquire();
39             } catch (InterruptedException e) {
40                 e.printStackTrace();
41             }
42         }
43
44         System.out.println(Thread.currentThread().getName() + " is READING.");
45         readLock.release();
46     }
47
48     public void writeLock() {
49         try {
50             writeLock.acquire();
51         } catch (InterruptedException e) {
52             e.printStackTrace();
53         }
54         System.out.println(Thread.currentThread().getName() + " is WRITING.");
55     }
56
57     public void readLock() throws InterruptedException {
58         try {
59             readLock.acquire();
60         } catch (InterruptedException e) {
61             e.printStackTrace();
62         }
63         --readCounter;
64
65         if (readCounter == 0) {
66             writeLock.release();
67         }
68
69         System.out.println(Thread.currentThread().getName() + " is DONE READING.");
70         readLock.release();
71     }
72
73     public void writeLock() {
74         try {
75             writeLock.acquire();
76         } catch (InterruptedException e) {
77             e.printStackTrace();
78         }
79         System.out.println(Thread.currentThread().getName() + " is DONE WRITING.");
80         writeLock.release();
81     }
82 }
83
84 class Writer implements Runnable {
85     private ReaderLock rl;
86     public Writer(ReaderLock rl) {
87         rl.lock = rl;
88     }
89     public void run() {
90         while (true) {
91             sleep(Utilities.map());
92             rl.lock.writeLock();
93             sleep(Utilities.map());
94             rl.lock.writeLock();
95         }
96     }
97 }
98
99 class Reader implements Runnable {
100     private ReaderLock rl;
101     public Reader(ReaderLock rl) {
102         rl.lock = rl;
103     }
104     public void run() {
105         while (true) {
106             sleep(Utilities.map());
107             rl.lock.readLock();
108             try {
109                 sleep(Utilities.map());
110                 rl.lock.readLock();
111             } catch (InterruptedException e) {
112                 e.printStackTrace();
113             }
114         }
115     }
116 }
117
118 class Utilities {
119     public static void map() {
120         int sleepTime = (int) (MAP_TIME * Math.random());
121         try {
122             Thread.sleep(sleepTime * 1000);
123         } catch (InterruptedException e) {
124             e.printStackTrace();
125         }
126     }
127     private static final int MAP_TIME = 0;
128 }
```



```
105     public void writeLock() {
106         try {
107             writeLock.acquire();
108         } catch (InterruptedException e) {
109             e.printStackTrace();
110         }
111         System.out.println(Thread.currentThread().getName() + " is DONE WRITING.");
112         writeLock.release();
113     }
114 }
115
116 class Writer implements Runnable {
117     private ReaderLock rl;
118     public Writer(ReaderLock rl) {
119         rl.lock = rl;
120     }
121     public void run() {
122         while (true) {
123             sleep(Utilities.map());
124             rl.lock.writeLock();
125             sleep(Utilities.map());
126             rl.lock.writeLock();
127         }
128     }
129 }
130
131 class Reader implements Runnable {
132     private ReaderLock rl;
133     public Reader(ReaderLock rl) {
134         rl.lock = rl;
135     }
136     public void run() {
137         while (true) {
138             sleep(Utilities.map());
139             rl.lock.readLock();
140             try {
141                 sleep(Utilities.map());
142                 rl.lock.readLock();
143             } catch (InterruptedException e) {
144                 e.printStackTrace();
145             }
146         }
147     }
148 }
149
150 class Utilities {
151     public static void map() {
152         int sleepTime = (int) (MAP_TIME * Math.random());
153         try {
154             Thread.sleep(sleepTime * 1000);
155         } catch (InterruptedException e) {
156             e.printStackTrace();
157         }
158     }
159     private static final int MAP_TIME = 0;
160 }
```

Key Components

1) ReadWriteLock class:

- The heart of the read-write lock mechanism:
 - Manages access to the shared data using semaphores.
 - Ensures multiple readers can access data simultaneously while only one writer can modify it at a time.
- Key elements:
 - **readLock** and **writeLock** semaphores: Control access to the read and write locks.
 - **readCounter**: Keeps track of how many readers are currently active.
 - **readLock()**, **writeLock()**, **readUnLock()**, and **writeUnLock()** methods: Handle acquiring and releasing the locks.

2) Writer and Reader classes:

- Represent the different types of threads:
 - **Writer** threads modify the shared data.
 - **Reader** threads only read the data.
- Implement the Runnable interface:
 - Contain a **run()** method that dictates their behaviour.
 - Acquire the appropriate lock (write or read) before accessing the data.
 - Simulate writing or reading operations by sleeping for random durations.

3) SleepUtilities class:

- Provides utility methods for thread sleep:
 - **nap()** method: Introduces random delays to simulate real-world processing times.
 - Helps observe how the read-write lock handles concurrent access under different scenarios.

4) Semaphores:

- Synchronization tools for controlling access to shared resources:
 - Act as gatekeepers, ensuring only permitted threads can access the data at specific times.
 - Used to implement the read-write lock logic in this code.

Breakdown The Codes

package sync; :

- Declares that the code belongs to the package named "sync." Packages organize code into logical groups for better structure and reusability.

import java.util.concurrent.ExecutorService; :

- Imports the **ExecutorService** interface from the **java.util.concurrent** package. It's used for managing thread execution in a flexible and efficient way.

import java.util.concurrent.Executors; :

- Imports the **Executors** class from the same package. It provides factory methods for creating different types of **ExecutorService** instances, offering convenient ways to manage threads.

import java.util.concurrent.Semaphore; :

- Imports the **Semaphore** class, also from the **java.util.concurrent** package. It's a synchronization tool used for controlling access to shared resources by multiple threads. It acts like a gatekeeper, ensuring only a limited number of threads can access a resource at a given time.

```
public class Project
{
    .
    .
}
```

1. Setting the stage:

- `public class Project { ... }` : Declares a public class named "Project" containing the main code.
- `public static void main(String[] args) { ... }` : This is the entry point of the program, where execution begins.

2. Managing threads:

- `ExecutorService executorService = Executors.newCachedThreadPool();` : Creates a thread pool that can efficiently manage multiple threads.

3. Creating a read-write lock:

- `ReadWriteLock RW = new ReadWriteLock();` : Instantiates a `ReadWriteLock` object to coordinate access to shared data.

4. Launching writer threads:

- `executorService.execute(new Writer(RW));` (repeated four times): Submits four **Writer** threads to the thread pool, each using the same **ReadWriteLock** object. These threads simulate writing to shared data.

5. Launching reader threads:

- `executorService.execute(new Reader(RW));` (repeated four times): Submits four **Reader** threads to the thread pool, also using the same **ReadWriteLock** object. These threads simulate reading from the shared data.

Key takeaways:

- The code sets up a multithreaded environment with multiple threads accessing shared data.
- The **ReadWriteLock** object manages concurrent access, allowing multiple readers or a single writer at a time to ensure data consistency.
- The **ExecutorService** handles thread execution, efficiently managing the multiple threads.

```
class ReadWriteLock
{
.
.
}
```

Key Elements:

- Semaphores:
 - **readLock**: Controls access to the read lock, allowing multiple readers.
 - **writeLock**: Controls access to the write lock, allowing only one writer at a time.
- **readCounter**: Tracks the number of active readers.

Methods:

- **readLock()**:
 - 1) Acquires the **readLock** semaphore.
 - 2) Increments the **readCounter**.
 - 3) If this is the first reader, acquires the **writeLock** semaphore to prevent writers from accessing data during reads.
 - 4) Prints a message indicating the thread is reading.
 - 5) Releases the **readLock** semaphore.
- **writeLock()**:
 - 1) Acquires the **writeLock** semaphore, ensuring exclusive access for writing.
 - 2) Prints a message indicating the thread is writing.
- **readUnLock()**:
 - 1) Acquires the **readLock** semaphore.
 - 2) Decrements the **readCounter**.
 - 3) Releases the **writeLock** semaphore if there are no more readers, allowing writers to access data again.
 - 4) Prints a message indicating the thread is done reading.
 - 5) Releases the **readLock** semaphore.
- **writeUnLock()**:
 - 1) Prints a message indicating the thread is done writing.
 - 2) Releases the **writeLock** semaphore, allowing other threads to acquire it.

class Writer implements Runnable

```
{  
.  
.  
}
```

Purpose:

- Represents a writer thread that simulates writing to shared data in a multithreaded environment.

Key Elements:

- **RW_lock**: A reference to a **ReadWriteLock** object, used to acquire and release the write lock for exclusive access to the shared data.

run() method:

- 1) Infinite loop: Ensures the thread continuously attempts to write data.
- 2) **SleepUtilities.nap()**; : Pauses the thread for a random duration, simulating varying processing times and potential contention for the lock.
- 3) **RW_lock.writeLock()**; : Acquires the write lock, blocking other threads from reading or writing until the lock is released.
- 4) **SleepUtilities.nap()**; : Pauses again to simulate writing operations.
- 5) **RW_lock.writeUnLock()**; : Releases the write lock, allowing other threads to access the data.

Key Points:

- Represents a thread that needs exclusive access to shared data for writing.
- Uses the **ReadWriteLock** object to coordinate access and prevent conflicts with other threads.
- Simulates writing operations by acquiring the write lock, pausing, and then releasing the lock.
- Demonstrates how a write lock can be used to ensure data integrity in a multithreaded environment.

class Reader implements Runnable

```
{  
.  
.  
}
```

Purpose:

- Represents a reader thread that simulates reading from shared data in a multithreaded environment.

Key Elements:

- **RW_lock**: A reference to a **ReadWriteLock** object, used to manage access to the shared data.

run() method:

- 1) **while (true) { ... }**: Loops indefinitely, repeatedly attempting to read data.
- 2) **SleepUtilities.nap();** : Pauses the thread for a random duration, simulating varying processing times.
- 3) **RW_lock.readLock();** : Acquires the read lock, allowing multiple readers to access the data concurrently.
- 4) **try { ... } catch (InterruptedException e) { ... }** : Handles potential interruptions during reading or unlocking.
- 5) **SleepUtilities.nap();** : Pauses again to simulate reading operations.
- 6) **RW_lock.readUnLock();** : Releases the read lock, potentially allowing a waiting writer to access the data.

Key Points:

- Represents a thread that needs shared data only for reading.
- Uses the **ReadWriteLock** object to coordinate access and prevent conflicts with writers.
- Simulates reading operations by acquiring the read lock, pausing, and then releasing the lock.
- Demonstrates how multiple readers can access data simultaneously under a read-write lock.
- Handles potential interruptions gracefully.


```
class SleepUtilities
```

```
{  
.  
.  
}
```

Purpose:

- Provides utility methods for pausing threads for random durations, adding realism and simulating varying processing times in a multithreaded environment.

Methods:

- **public static void nap():**
 - Calls **nap(NAP_TIME)** with a default duration of 6 seconds.
- **public static void nap(int duration):**
 - 1) Generates a random sleep time between 0 and **duration** seconds.
 - 2) Pauses the current thread for the calculated sleep time using **Thread.sleep()**.
 - 3) Catches and ignores **InterruptedException** for simplicity in this context.

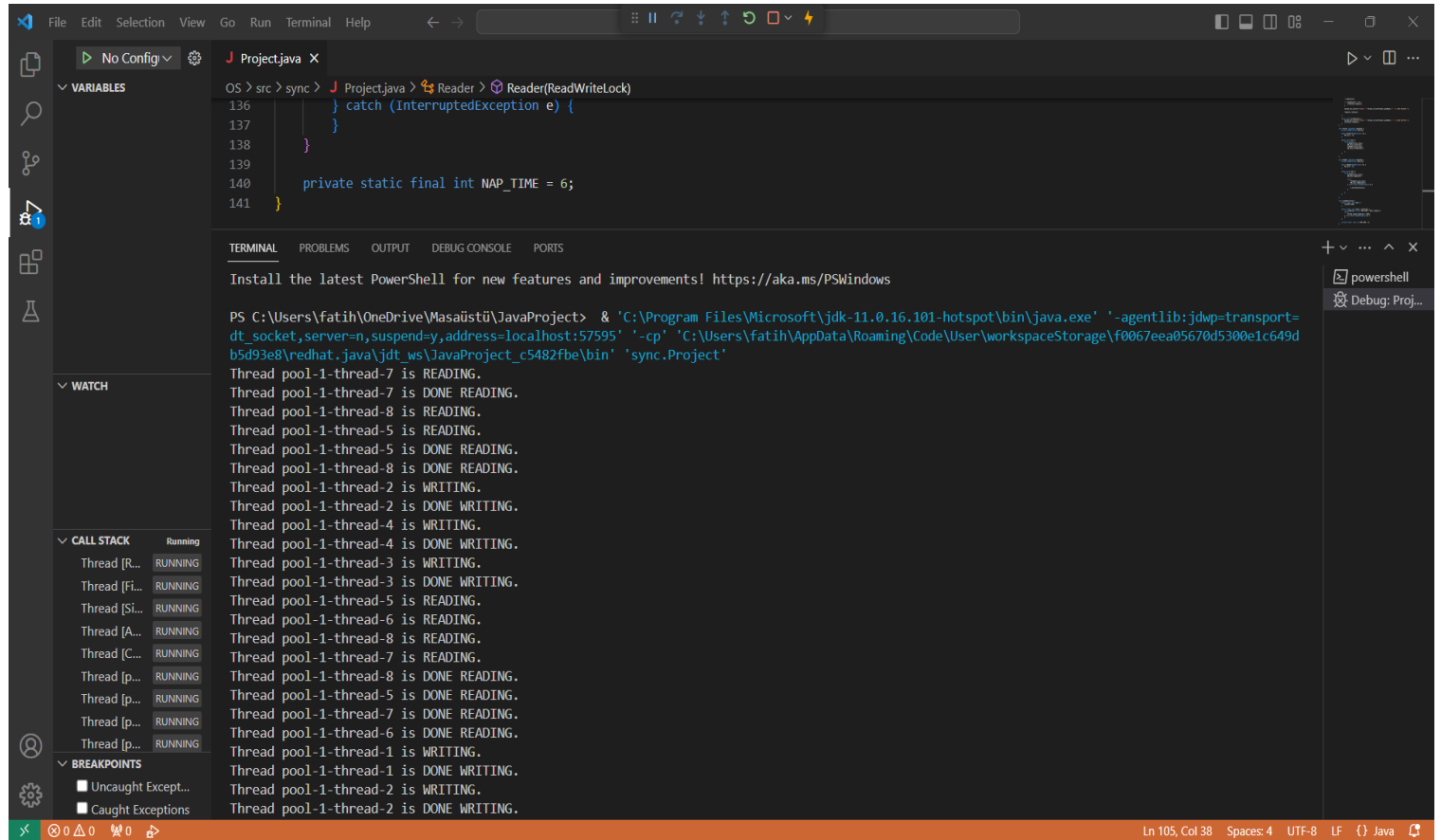
Constant:

- **private static final int NAP_TIME = 6; :**
 - Sets the default maximum sleep time in seconds for the **nap()** method.

Key Points:

- Offers convenient methods for introducing random delays in thread execution.
- Helps simulate real-world scenarios where processing times can vary.
- Contributes to observing read-write lock behavior under different thread interactions.

Output of The Codes



The screenshot displays an IDE interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The main editor window shows a Java file named 'Project.java' with the following code:

```
OS > src > sync > J Project.java > Reader > Reader(ReadWriteLock)
136         } catch (InterruptedException e) {
137         }
138     }
139
140     private static final int NAP_TIME = 6;
141 }
```

Below the code editor, the 'TERMINAL' tab is active, showing the command prompt output:

```
PS C:\Users\fatih\OneDrive\Masaüstü\JavaProject> & 'C:\Program Files\Microsoft\jdk-11.0.16-hotspot\bin\java.exe' '-agentlib:jdwp=transport=dt_socket,server=n,suspend=y,address=localhost:57595' '-cp' 'C:\Users\fatih\AppData\Roaming\Code\User\workspaceStorage\f0067eea05670d5300e1c649db5d93e8\redhat.java\jdt_ws\JavaProject_c5482fbc\bin' 'sync.Project'
```

The output shows the progress of the program execution, including thread states and completion messages:

```
Thread pool-1-thread-7 is READING.
Thread pool-1-thread-7 is DONE READING.
Thread pool-1-thread-8 is READING.
Thread pool-1-thread-5 is READING.
Thread pool-1-thread-5 is DONE READING.
Thread pool-1-thread-8 is DONE READING.
Thread pool-1-thread-2 is WRITING.
Thread pool-1-thread-2 is DONE WRITING.
Thread pool-1-thread-4 is WRITING.
Thread pool-1-thread-4 is DONE WRITING.
Thread pool-1-thread-3 is WRITING.
Thread pool-1-thread-3 is DONE WRITING.
Thread pool-1-thread-5 is READING.
Thread pool-1-thread-6 is READING.
Thread pool-1-thread-8 is READING.
Thread pool-1-thread-7 is READING.
Thread pool-1-thread-8 is DONE READING.
Thread pool-1-thread-5 is DONE READING.
Thread pool-1-thread-7 is DONE READING.
Thread pool-1-thread-6 is DONE READING.
Thread pool-1-thread-1 is WRITING.
Thread pool-1-thread-1 is DONE WRITING.
Thread pool-1-thread-2 is WRITING.
Thread pool-1-thread-2 is DONE WRITING.
```

The left sidebar contains several panels: 'VARIABLES', 'WATCH', 'CALL STACK', and 'BREAKPOINTS'. The 'CALL STACK' panel shows a list of threads, all with a 'RUNNING' status. The 'BREAKPOINTS' panel shows two breakpoints: 'Uncaught Except...' and 'Caught Exceptions'. The bottom status bar indicates the current line and column (Ln 105, Col 38), the number of spaces (4), the encoding (UTF-8), the line feed (LF), and the file type (Java).