Documentation: Portfolio Part-2

Alexander Ricciardi

Colorado State University Global

CSC450: Programming III

Professor: Reginald Haseltine

December 1, 2024

Documentation: Portfolio Part-2

This documentation is part of the Module-8 Portfolio from CSC450: Programming III at Colorado State University Global. The program's name is Thread Counting Synchronization. It provides an overview of the program's functionality and testing scenarios, including an analysis write-up of the important concepts of concurrency in Java related to the program. The program is coded in Java SE 21.

The Assignment Direction:

Portfolio Project Part II
Concurrency Concepts
For your Portfolio Project, you will demonstrate an understanding of the various concepts discussed in each module. For the second part of your Portfolio Project, you will create a Java application that will exhibit concurrency concepts. Your application should create two threads that will act as counters. One thread should count up to 20. Once thread one reaches 20, then a second thread should be used to count down to 0. For your created code, please provide a detailed analysis of appropriate concepts that could impact your application. Specifically, please address:

- Performance issues with concurrency
- Vulnerabilities exhibited with use of strings
- Security of the data types exhibited.

Submit the following components to your GIT repository in a single document:

- Word document with appropriate screenshots of your program executing, program analysis responses, and source code in the Word file.
- Submit your .java source code file(s). If more than 1 file, submit a zip file.
- Provide a detailed comparison between the performance implementations between the Java and C++ versions of your applications. Which implementation may be considered less vulnerable to security threats and why? Your detailed comparison should be 3-4 pages in length. Any citations should be formatted according to guidelines in the CSU Global Writing Center (located in the course navigation menu).

To receive full credit for the packaging requirements for your Module 8 Portfolio Project assignment you must:
1) Put your Java source code in .java text files. Note that I execute all your programs to check them out.
2) In a Word or PDF "documentation and analysis" file, labeled as such, put a copy of your Java source code and execution output screen snapshots. This week this regular documentation file must also include a detailed analysis write-up of the important concepts of concurrency with Java to cover in detail performance issues, string vulnerabilities, and security of data types.
3) Some positive evidence that you've definitely stored your source code in a GitHub repository on GitHub.com.
4) Include a separate 3-4 page APA Edition 7 paper comparing the performance and security differences between C++ and Java multi-threaded programs. Here's the link to the school's Writing Center for APA Edition 7 requirements -> https://csuglobal.libguides.com/writingcenterLinks to an external site.
5) Put all 4 of the above files into a single .zip file, and submit ONLY that .zip file for grading. Do not submit any additional separate files.

⚠ **My notes:**

- The simple Java console application is in the file **ThreadCountingSynchronization.java**
- The program follows the following SEI CERT Oracle Coding Standards for Java:
    - STR00-J. Don't form strings containing partial characters from variable-width encodings

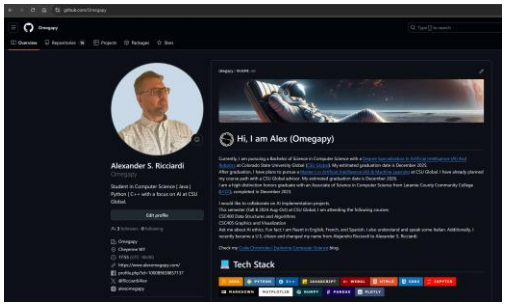
- STR01-J. Do not assume that a Java char fully represents a Unicode code point
- ERR00-J. Do not suppress or ignore checked exceptions
- ERR01-J. Do not allow exceptions to expose sensitive information
- ERR03-J. Restore prior object state on method failure
- ERR09-J. Do not allow untrusted code to terminate the JVM
- LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code
- LCK08-J. Ensure actively held locks are released on exceptional conditions
- THI02-J. Notify all waiting threads rather than a single thread
- THI03-J. Always invoke wait() and await() methods inside a loop
- TSM00-J. Do not override thread-safe methods with methods that are not thread-safe

Program Description:

This program demonstrates the use of threads and how to synchronize them using ReentrantLocks and Conditions.
Thread 1 counts up from 0 to a maximum count, while Thread 2 waits until Thread 1 completes, and then counts down from the maximum count to 0.

Git Repository

I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.
My GitHub repository that is used to store this assignment is named My-Academics-Portfolio.
The link to this specific assignment is:



Images of the source code in the GitHub are in the *Screenshot.docx* file. On the next page is the detailed analysis write-up of the important concepts of concurrency with Java to cover in detail performance issues, string vulnerabilities, and security of data types.

Detailed Analysis:

In Java as in C++, concurrency, or multithreading, is used to improve applications' responsiveness and performance. However, concurrency may invertedly introduce vulnerabilities such as performance issues, string vulnerabilities, and security data type issues that need to be addressed. This section explores a Java program called "Thread Counting Synchronization" by exploring how those issues can impact it and solutions to address them.

The Program

The program is a small Java program that uses two threads, one to count up and the other to count down. It synchronizes the two threads by implementing a mutex, a condition variable, and by following SEI CERT Oracle Coding Standards for Java. The first thread function '*countUp*' counts up from 0 to the maximum count which is set to 20, and increments the shared atomic variable '*counter*', it also displays the value of the '*counter*' variable for each incrementation of it. On the other hand, the second thread waits until the first thread is done counting, then the thread function '*countDown*' counts down from the value of the shared variable '*counter*' to 0, decrements the '*counter*', and displays the value of the 'counter' variable for each decremention.

Performance Issues with Concurrency

Java concurrency allows threads to execute simultaneously. This allows the applications, especially with multi-core Central Processing Units CPUs, to be more computing efficient and responsive. However, like in C++ in Java, it also adds significant overhead in the form of resource management tasks to address issues that are associated with threat synchronization and shared data, such as data races, deadlocks, and undefined behavior.

Data race happens when several threads, two or more, try to access the same shared data at the same time, and at least one thread modifies the same data. The program addresses data races by declaring the shared variable '*counter*' as an atomic data type; this means that when an operation is performed on the variable it can be interrupted by another thread. Note that the program uses atomic variables '*AtomicInteger*' and '*AtomicBoolean*'. Using atomic variables adheres to the SEI CERT Oracle Coding Standards for Java TSM00-J: " *Do not override thread-safe methods with methods that are not thread-safe*" (CMU, n.d. a, TSM00-J). It also follows the SEI CERT Oracle Coding Standards for Java LCK08-J: "*Ensure actively held locks are released on exceptional conditions*" (CMU, n.d. b, LCK08-J). As in C++, this approach addresses data races, but it also introduces delays and latency in real-world scenarios.

Deadlock usually happens when several threads, at least two, are waiting indefinitely for resources that are held by each other, preventing any of them from proceeding further in their computation. The program addresses Deadlock by using '*ReentrantLock*' which is a class that implements the lock interface and provides safe synchronization, it surrounds the shared data with calls to the lock and unlock method ensuring that the data is only accessed by one thread at a time and preventing deadlocks. This is very useful when using a recursive method or when the shared data is manipulated recursively in a loop. The program also encapsulated the operation of the thread in '*try-catch-finally'* blocks, ensuring that if an exception occurs the lock is released safely preventing deadlocks and starvation. This adheres to the SEI CERT Oracle Coding Standards for Java LCK08-J: "*Ensure actively held locks are released on exceptional conditions*" (CMU, n.d. b, LCK08-J). Additionally, the program uses a condition object '*condition*' combined with a Boolean variable '*isCountingUpDone*' to signal the second thread, the waiting thread to begin execution once the first thread finishes by notifying all the threads when it is done. This adheres to the rules THI02-J: "*Notify all waiting threads rather than a single thread*" (CMU, n.d. c, THI02-J) and THI03-J "*Always invoke wait() and await() methods inside a loop*" (CMU, n.d. c, THI03-J) as the wait condition is checked by a while-loop. While these approaches handle thread

synchronization well and prevent deadlock, in more complex systems if not handled properly, with more threads accessing the same data, it can create performance bottlenecks.

Vulnerabilities in String Usage

Strings can be used for thread identification; the program uses two string variables called '*thread1Name*' and *'thread2Name*' to store the thread names. Strings are immutable and generally safe in Java; nonetheless, if used improperly they can invertedly reveal sensitive data and create synchronization issues and vulgarities if used to define locks. The program addresses these issues by using '*private final ReentrantLock lock = new ReentrantLock();*' object, it cannot be modified or accessed externally, thus maintaining encapsulation and security and it does not use a string to define lock but an object. This adheres to SEI CERT Oracle Coding Standards for Java LCK00-J: "*Use private final lock objects to synchronize classes that may interact with untrusted code*" (CMU, n.d. b, LCK00-J) and STR01-J: '*STR01-J. Do not assume that a Java char fully represents a Unicode code point*' (CMU, n.d. d, STR01-J). Additionally, the program addresses the possibility of forming malformed strings. '*threadName == null*', adhering to STR00-J: "*Don't form strings containing partial characters from variable-width encodings*" (CMU, n.d. d, STR00-J).

Security of Data Types

As shown previously, the program secures data types by using '*AtomicInteger*' and '*AtomicBoolean*' atomic types that act as a flag to signal thread completion. Additionally, it uses '*ReentrantLock*', ensuring data integrity, and safe '*try-catch-finally*' blocks to handle exceptions. The handling of the '*try-catch*' adheres to SEI CERT Oracle Coding Standards for Java ERR00-J: "Do not allow exceptions to expose sensitive information" (CMU, n.d. e, ERR00-J) and ERR01-J: "*Do not allow exceptions to expose sensitive information*" (CMU, n.d. e, ERR01-J). Additionaly, it implements scoped resource management through Resource Acquisition Is Initialization (RAII) principle, by using '*try-finally*' blocks to ensure that locks are always released, even in unexpected conditions assuring the security of data types.

Considerations for Scalability

The program is a simple Java program made to demonstrate the handling of concurrency for two threads and a single shared counter. However, when scaling it up can introduce challenges like handling a greater number of shared data and the increased complexity of the program due to added functionalities. Using thread pools ('*ExecutorService*') would better manage resources as the number of threads increases and atomic data structures or objects like '*ConcurrentLinkedQueue*' or '*AtomicReference*' would increase performance while maintaining thread safety.

Conclusion

To summarize, the program, "Thread Counting Synchronization," as its C++ counterpart, demonstrates concurrency but in Java and addresses challenges related to it like performance issues, vulnerabilities in string usage, and the security of data types. The program also implements SEI CERT Oracle Coding Standards for Java that address the safe functionality of the thread and their synchronization, as well as the secure manipulation of strings. However, the use of synchronization may introduce significant performance overhead. While the program is optimized for its small size, scaling up the program may require ensuring that new data types are handled properly, and it also may require integrating different approaches for tread synchronization such as using thread pooling with '*ExecutorService*' and implementing lock-free data structures, such as '*ConcurrentLinkedQueue*' or '*AtomicReference*', that can improve performance without having to use explicit locks.

References:

CMU (n.d. a). Rule 12. Thread-Safety Miscellaneous (TSM). *SEI CERT Oracle Coding Standards for Java*. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487731

CMU (n.d. b). Rule 09. Locking (LCK). *SEI CERT Oracle Coding Standards for Java*. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487666

CMU (n.d. c). Rule 10. Thread APIs (THI). *SEI CERT Oracle Coding Standards for Java*. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487735

CMU (n.d. d). Rule 04. Characters and Strings (STR). SEI CERT Oracle Coding Standards for Java. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487607

CMU (n.d. d). Rule 04. Rule 07. Exceptional Behavior (ERR). SEI CERT Oracle Coding Standards for Java. Carnegie Mellon University. Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487665