**Portfolio Milestone Assignment Module Eight Assignment**

**Portfolio Project Part 2**

Victor Enogwe

Computer Science, Colorado State University - Global Campus

CSC450-1: Programming III

Reginald Haseltine

June 09, 2024

**Multithreaded Integer Counter Program**

This assignment demonstrates concurrent execution in Java using threads. The application executes two counter threads, counting up to twenty and counting down to zero. The following analysis details the various concepts encountered during the program's implementation.

**Performance issues with concurrency**

Concurrency in our counter program uses threads. The program has three threads: the “main” thread, the “countToTwenty” thread, and the “countToZero” thread. Since memory space is shared between the threads, no special performance issues arise during memory access across the threads.

Concurrent execution means the threads execute based on some time slot scheduling mechanism. Since the process shared by the threads is run on a single processor core, performance is slower than when the thread functions are executed in parallel.

The shared memory space introduces a state management problem, where each thread must ensure that it correctly reads and writes to the shared state exclusively. This necessitates using a mutex lock to protect the read and write to the global “threadCount” property “value” variable, ensuring that each thread has exclusive access to read and write to the counter.

To do this, we must recognise and protect the code's various critical variables accordingly. Starting with the “Count” class, we mark the integer “value” property, its setter and getter methods as synchronised using the “synchronized” keyword ((Kumar, 2024). This creates a mutex that ensures the property can only be read and written to one thread at a time after acquiring its lock.

We then use a “ReentrantLock” mutex to control access to the critical section and ensure an all-or-nothing run for our while loop counter blocks. This ensures that each loop completes before the next loop, which reads the counter property, is run. If this is not implemented, the shared counter state will have indeterminate values at the end of the program run.

Many mutex lock types exist in Java, such as “Semaphores” and the “ReentrantReadWriteLock;” for the sake of this program and simplicity, I have used the “synchronize” keyword and a “Reentrant” lock.

**Vulnerabilities exhibited with the use of strings.**

This program does not use unbounded string inputs dynamically within an evaluation. To avoid injection attacks, unbounded string inputs should be sanitised and validated against a set of possible values.

Integer data types could be prone to underflows and overflows if their sizes are not appropriately accounted for when allocating and assigning them. We use the 32-bit “int” type, sufficient for our counting purposes.

**Security of the data types exhibited.**

The two data types used in this program are string and integer. For the string data type, there is no externally provided value; the strings used are constants with predefined memory allocations. Hence, the program does not need to validate, sanitise, or account for buffer overflows when storing the strings (Ballman, 2016, p. 200).

Signs and sizes must be accounted for when using integers to avoid an integer overflow (Coker et al., n.d., p.1). We do not need to specify an “int” size type such as “short” since we are going to run this on a 32/64-bit computer, where the “int” type will be converted appropriately.

**Performance and Implementation Comparison Between the Java and C++ version.**

Given their respective pseudo code, the two program versions are functionally identical and can be implemented interchangeably. The Java program implementation uses object-oriented programming, while the C++ implementation uses procedural programming techniques. Both languages, however, support object-oriented programming, and the C++ implementation can be rewritten using object-oriented programming constructs.

C++ is a lower-level language than Java, so its language abstractions are much closer to the operating system assembly language (Rodriguez, 2021). Given the same algorithm and similar data structures, any implementation in C++ is faster than that in Java.

Both implementations use similar data structures, the string and integer types. For the sake of this analysis, I assumed one-to-one performance similarities between these types, as they are inbuilt data types and have internal optimisations to make them perform as optimally as possible given any runtime environment.

The mutex abstract data type (ADT) is used in both implementations. The name is an abbreviation of the phrase – “mutual exclusion”. Mutexes aim to achieve atomicity and isolation during read-write operations on data.

In the C++ program implementation, a basic mutex locks and unlocks the thread runner function. The Java program, on the other hand, uses a reentrant lock-based mutex. In both cases, the runtime operating system and its scheduler are in charge of execution speed, allocating processing time for each thread and performance. There are no guarantees on the fairness of each thread run based on the selected mutexes. Any of the two threads may run before the other thread.

Both programs use a single while loop with a Big Oh constant time operation of O(n) + c(where c is a constant representing operations except the while loop initialiser). The C++ program implementation would be faster because it is implemented in a lower-level language with fewer abstractions over the data primitives and the ADT.

**Vulnerability Analysis of The C++ and Java Implementation.**

Abstractions help to hide complexity, enabling programming languages to present more tested and trusted APIs to developers. As a result, the Java program is less vulnerable to security threats than the C++ program.

In C++, data types like integers and strings require more effort to ensure proper sanitisation and memory management. The abstractions in Java free the programmer from knowing and worrying about the intricate details of allocation, deallocation, and most memory sizing issues found in C++.

For my implementations, the basic mutex lock we have used in C++ allows us to write less code to protect our critical section, and the amount of code we write to set up the thread is small. We need to instantiate the mutex and thread in Java and use more APIs to achieve feature parity. We also write more classes(five more) and use an object-oriented design. More code and cyclomatic complexity may open up the program to more bugs. A more experienced C++ or Java programmer might do more to ameliorate these shortcomings.

Conversely, the object-oriented design may be advantageous in partitioning and modularising the program so that each part can be tested in isolation, aiding bug and vulnerability reduction.

**Program Pseudocode**:

PROGRAM Thread Counters  
- Multithreaded concurrent program that counts up to twenty and down to zero.  
  
BEGIN  
 BEGIN  
 Define a "Count" class.  
 - Define a private volatile property "value" - the counter variable in the class.  
 - Define the constructor that takes an integer value and sets it to the counter's value using a setter method.  
 - Define a synchronised getter method for the "value" property.  
 - Define a synchronised setter method for the "value" property and use it in the constructor.  
 END  
  
 BEGIN  
 Define a "CountRunnable" class that implements the "Runnable" interface.  
 - Define a protected property "threadCount" - an instance of the "Count" class.  
 - Define the constructor that a "threadCount" value(an instance of the "Count" class) and sets it to the value of the "threadCount" property using a setter.  
 - Define a setter method for the "setThreadCount" property and use it in the constructor.  
 END  
  
 BEGIN  
 Define a "CountToTwentyRunnable" class that extends the "CountRunnable" class.  
 - Define a "countToTwenty" public method that counts up to twenty using a loop.  
 - The "countToTwenty" increments a global "threadCount" variable.  
 - Print the result of each increment.  
 - Define a "run" method that calls the "countToTwenty" method.  
 END  
  
 BEGIN  
 Define a "CountToZeroRunnable" class that extends the "CountRunnable" class.  
 - Define a "countToZero" public method that counts down to zero using a loop.  
 - The "count\_to\_zero" decrements a global "threadCount" variable.  
 - Print the result of each decrement.  
 - Define a "run" method that calls the "countToZero" method.  
 END  
  
 BEGIN  
 Define a "CounterThread" class that extends the "Thread" class.  
 - Define a private "ReentrantLock" "mutex" variable  
 - Define the constructor  
 - that takes a "string" value "name" and sets its value to the value of the "name" property of the thread using its setter.  
 - that takes a "ReentrantLock" value "mutex" and sets its value to the value of the "mutex" property using its setter.  
 - that takes a "Runnable" value "runnable" and sets its value to the value of the "runnable" property of the thread.  
 - Define a getter method for the "mutex" property.  
 - Define a setter method for the "mutex" property and use it in the constructor.  
 - Define a static method - "details", that takes a String argument - "name" and prints the following thread details:  
 - thread id  
 - process id  
 - Define a method - "details", that calls the static "details" method with the current thread’s name  
 - Define a "run" function that executes the thread's runnable class and controls access to the shared count variable using the mutex property.  
 END  
  
 Define a "Main" class.  
 Define a "main" method inside the class.  
 - The main method should create the "countToTwenty(CounterThread instance)" thread using the corresponding "CountToZeroRunnable" instance, and the global counter("Count" instance) and mutex("ReentrantLock" instance).  
 - The main method should create the "countToZero(CounterThread instance)" thread using the corresponding "CountToZeroRunnable" instance, and the global counter("Count" instance) and mutex("ReentrantLock" instance).  
 - Both threads should be started and run concurrently  
 - The "counter" global variable state should be valid at the end of the program run.  
END

**Count.Java Class:**

package counter;

/\*\*

\* Count class

\*/

public class Count {

private volatile int value;

/\*\*

\* Constructor

\* Enforced Parameterized constructor

\*

\* @param value a shared integer counter

\*/

public Count(int value) {

setValue(value);

}

/\*\*

\* setValue

\*

\* @param value an integer counter

\*/

public synchronized void setValue(int value) {

this.value = value;

}

/\*\*

\* getValue

\* @return int the count value

\*/

public synchronized int getValue() {

return value;

}

}

**CountRunnable.Java Class:**

package counter;

public abstract class CountRunnable implements Runnable {

protected Count threadCount;

public CountRunnable(Count threadCount) {

setThreadCount(threadCount);

}

/\*\*

\* setThreadCount

\*

\* @param threadCount an integer counter

\*/

private void setThreadCount(Count threadCount) {

this.threadCount = threadCount;

}

}

**CountToTwentyRunnable.Java Class:**

package counter;

/\*\*

\* CountToTwenty class

\* counts from zero to twenty

\*/

public class CountToTwentyRunnable extends CountRunnable {

/\*\*

\* Constructor

\* Enforced Parameterized constructor

\*

\* @param threadCount a shared integer counter state

\*/

public CountToTwentyRunnable(Count threadCount) {

super(threadCount);

}

/\*\*

\* countToTwenty

\* counts up from zero to twenty and prints the count value.

\*/

private void countToTwenty() {

// critical section reading a shared variable.

System.out.printf("\nCount From %d to 20\n", threadCount.getValue());

while (threadCount.getValue() <= 20) {

System.out.println(threadCount.getValue());

// critical section writing to a shared variable.

threadCount.setValue(threadCount.getValue() + 1);

}

threadCount.setValue(threadCount.getValue() - 1);

System.out.println("");

}

/\*\*

\* run in thread

\*/

public void run() {

countToTwenty();

}

}

**CountToZeroRunnable.Java Class:**

package counter;

/\*\*

\* CountToTwenty class

\* counts from twenty to zero

\*/

public class CountToZeroRunnable extends CountRunnable {

/\*\*

\* Constructor

\* Enforced Parameterized constructor

\*

\* @param threadCount a shared integer counter state

\*/

public CountToZeroRunnable(Count threadCount) {

super(threadCount);

}

/\*\*

\* countToZero

\* counts down to zero and prints the count value.

\*/

public void countToZero() {

// critical section reading a shared variable.

System.out.printf("\nCount From %d to 0\n", threadCount.getValue());

while (threadCount.getValue() >= 0) {

System.out.println(threadCount.getValue());

// critical section writing to a shared variable.

threadCount.setValue(threadCount.getValue() - 1);

}

System.out.println("");

}

/\*\*

\* run in thread

\*/

public void run() {

countToZero();

}

}

**CounterThread.Java Class:**

package counter;

import java.util.concurrent.locks.ReentrantLock;

/\*

\* Thread is a class that represents a Thread.

\*/

public final class CounterThread extends Thread {

private ReentrantLock mutex;

public CounterThread(

String name,

ReentrantLock mutex,

Runnable runnable

) {

super(runnable);

super.setName(name);

setMutex(mutex);

}

/\*\*

\* getMutex

\* mutex lock

\* @return ReentrantLock

\*/

private ReentrantLock getMutex() {

return this.mutex;

}

/\*\*

\* setMutex

\* @param mutex the mutex lock

\*/

private void setMutex(ReentrantLock mutex) {

this.mutex = mutex;

}

/\*\*

\* details

\* @static

\* print thread details

\* @param name thread name

\*/

static public void details(String name) {

long processId = ProcessHandle.current().pid();

long threadId = Thread.currentThread().threadId();

System.out.printf(

"\nThread(%s) with id(%d) started\n",

name,

threadId

);

System.out.printf("Running within process: %d\n", processId);

}

/\*\*

\* details

\* get thread details

\*/

private void details() {

details(getName());

}

public void run() {

details();

// critical section

// mutex lock

getMutex().lock();

super.run();

// mutex unlock

getMutex().unlock();

}

}

**Main.Java Class:**

import java.util.concurrent.locks.ReentrantLock;

import counter.Count;

import counter.CountToTwentyRunnable;

import counter.CountToZeroRunnable;

import counter.CounterThread;

public class Main {

public static void main(String[] args) {

System.out.println("Count Up and Down Program.");

CounterThread.details("Main");

Count counter = new Count(0);

ReentrantLock mutex = new ReentrantLock();

CountToTwentyRunnable countToTwentyRunnable = new CountToTwentyRunnable(

counter

);

CountToZeroRunnable countToZeroRunnable = new CountToZeroRunnable(

counter

);

CounterThread countToTwentyThread = new CounterThread(

"Count To Twenty",

mutex,

countToTwentyRunnable

);

CounterThread countZeroThread = new CounterThread(

"Count To Zero",

mutex,

countToZeroRunnable

);

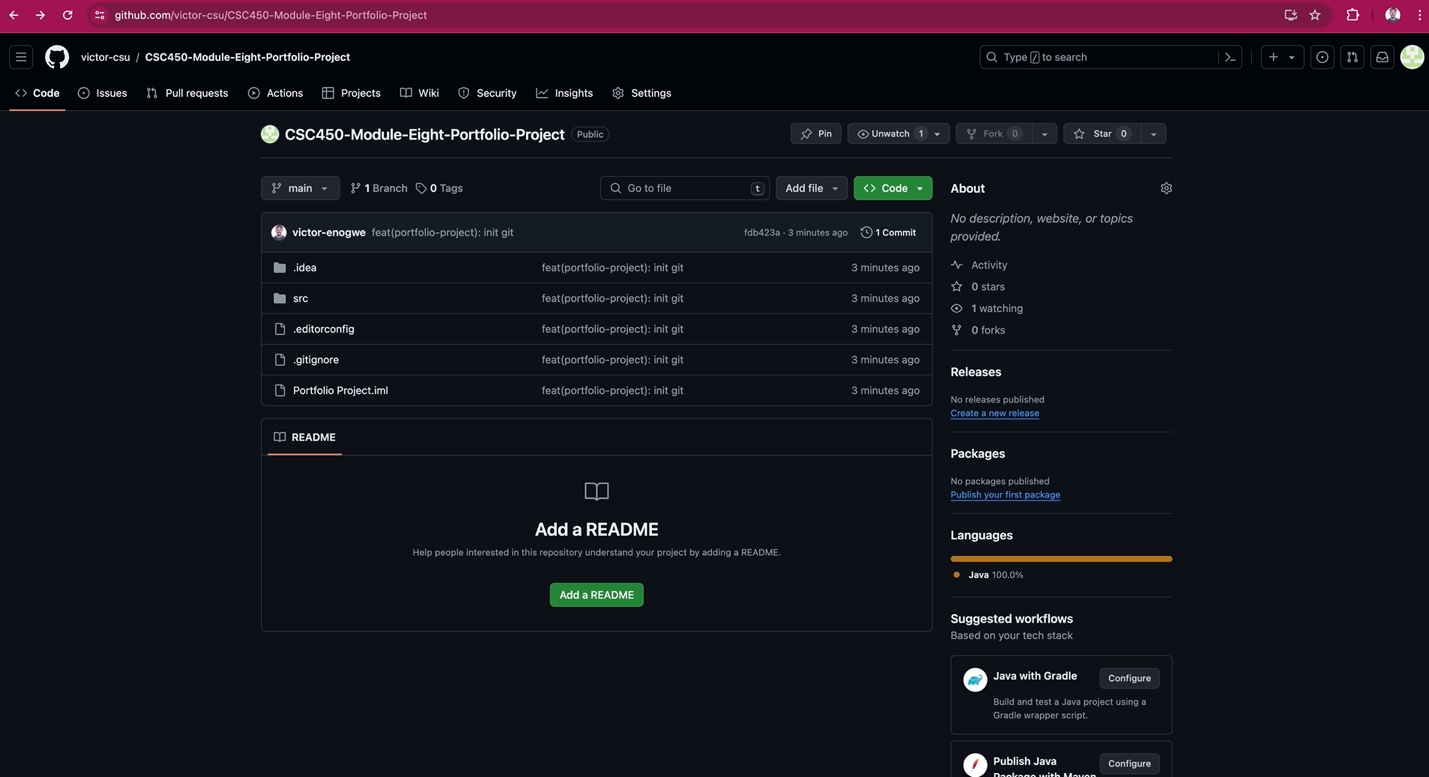
countToTwentyThread.start();

countZeroThread.start();

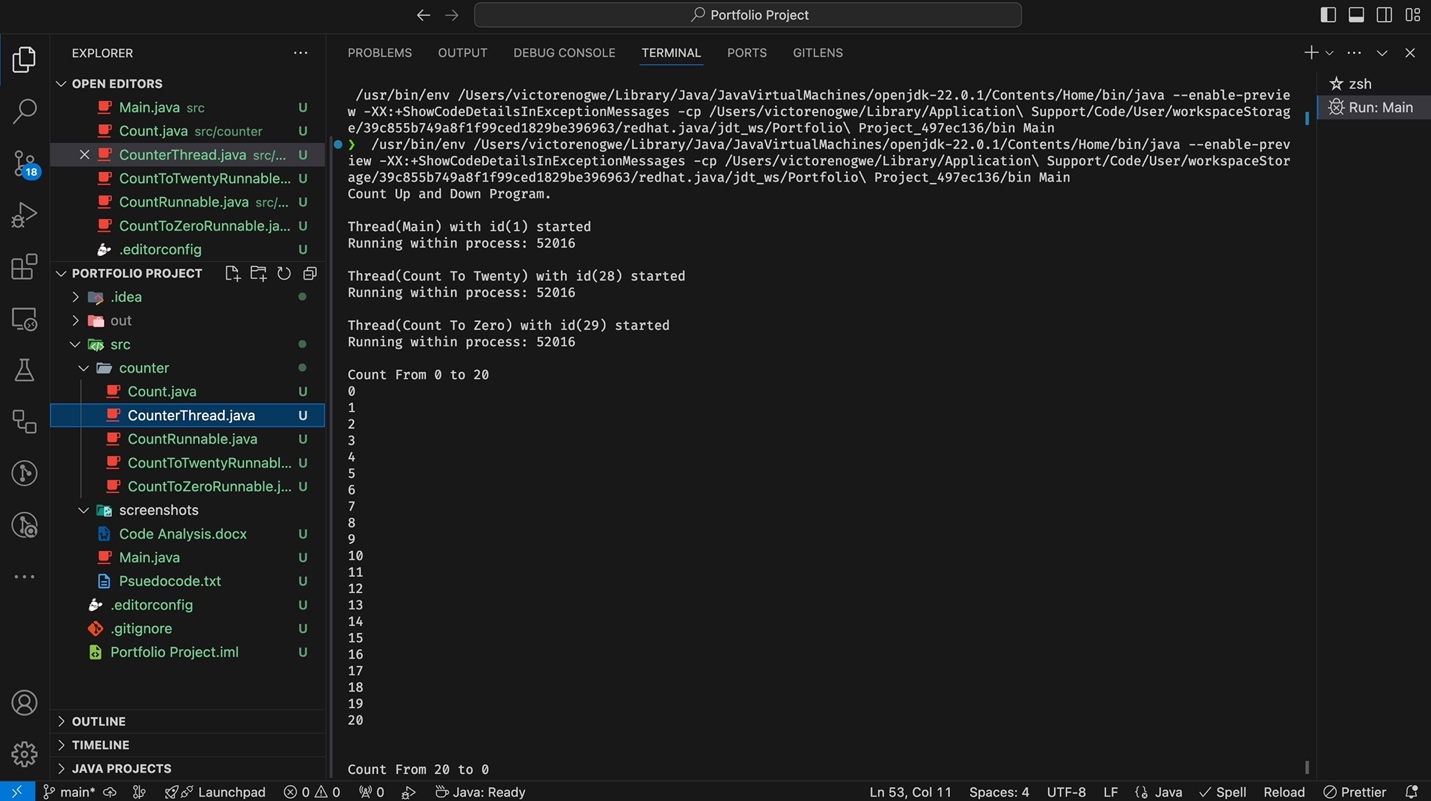
}

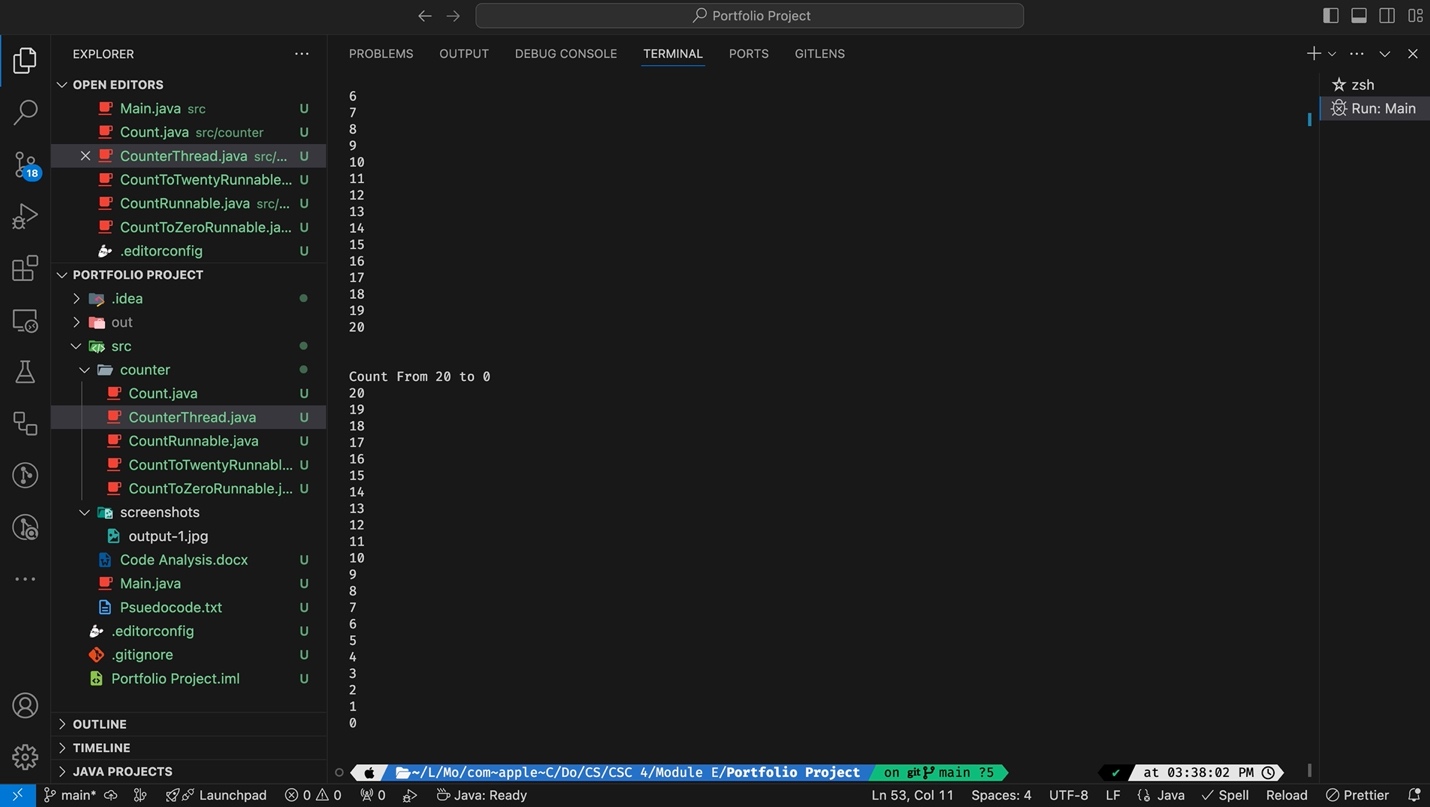
}

**Git Repository Image: Git Branch = Main** **-** **<https://github.com/victor-csu/CSC450-Module-Eight-Portfolio-Project>**

****

**Happy Path Execution Screenshot – Fictional Person - CSC450\_PP1\_mod7-execution-output:**





References

Coker, Z., Hassan, S., Overbey, J., Hafiz, M., & Kästner†, C. (n.d.). Integers In C: An Open Invitation To Security Attacks? <https://www.cs.cmu.edu/~ckaestne/pdf/csse14-01.pdf>

Ballman, A. (2016). *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++* (V01-20170309-0910). Rules for Developing Safe, Reliable, and Secure Systems in C++. <https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf>

Kumar, K. (2024). *Using a Mutex Object in Java*. Baeldung.com. <https://www.baeldung.com/java-mutex>

Rodriguez, K. (2021). *Java vs C++: Similarities, Differences, and Advantages*. Mvpmatch. https://www.mvpmatch.co/blog/java-vs-c-pp