MID

Let's start with **Exercise 1** and tackle it in both Java and C++.

Exercise 1: Find Integer with Largest Number of Divisors (1 to 10,000)

The task is to identify the integer between 1 and 10,000 with the most divisors and output that integer along with the number of divisors it has.

Here’s the approach we'll use:

1. Iterate over each integer in the range 1 to 10,000.
2. For each integer, find all divisors.
3. Track the maximum number of divisors found and the integer that has this maximum.

Java Solution

```
public class LargestDivisorCount {
    public static void main(String[] args) {
        int maxDivisors = 0; // Holds the maximum number of d
ivisors found
        int numberWithMaxDivisors = 0; // Holds the number wi
th the maximum divisors

        // Loop through each integer from 1 to 10000
        for (int i = 1; i <= 10000; i++) {
            int divisorCount = countDivisors(i); // Get divis
or count for i

            // Update maxDivisors and numberWithMaxDivisors i
f current count is higher
            if (divisorCount > maxDivisors) {
                maxDivisors = divisorCount;
```

```
                numberWithMaxDivisors = i;
            }
        }

        // Output result
        System.out.println("Number with the largest number of
divisors: " + numberWithMaxDivisors);
        System.out.println("Number of divisors: " + maxDiviso
rs);
    }

    // Method to count divisors of a given number
    private static int countDivisors(int number) {
        int count = 0;

        // Loop to find divisors by iterating up to the squar
e root of the number
        for (int i = 1; i * i <= number; i++) {
            if (number % i == 0) {
                count += 2; // i and (number / i) are both di
visors

                // Avoid counting the square root twice if i
t's a perfect square
                if (i * i == number) {
                    count--;
                }
            }
        }

        return count;
    }
}
```

C++ Solution

```
#include <iostream>
#include <cmath>
using namespace std;

// Function to count divisors of a given number
int countDivisors(int number) {
    int count = 0;

    // Loop to find divisors by iterating up to the square ro
ot of the number
    for (int i = 1; i * i <= number; i++) {
        if (number % i == 0) {
            count += 2; // i and (number / i) are both diviso
rs

            // Avoid counting the square root twice if it's a
perfect square
            if (i * i == number) {
                count--;
            }
        }
    }

    return count;
}

int main() {
    int maxDivisors = 0; // Holds the maximum number of divis
ors found
    int numberWithMaxDivisors = 0; // Holds the number with t
he maximum divisors

    // Loop through each integer from 1 to 10000
    for (int i = 1; i <= 10000; i++) {
        int divisorCount = countDivisors(i); // Get divisor c
```

```
ount for i

        // Update maxDivisors and numberWithMaxDivisors if cu
rrent count is higher
        if (divisorCount > maxDivisors) {
            maxDivisors = divisorCount;
            numberWithMaxDivisors = i;
        }
    }

    // Output result
    cout << "Number with the largest number of divisors: " <<
numberWithMaxDivisors << endl;
    cout << "Number of divisors: " << maxDivisors << endl;

    return 0;
}
```

Explanation of Key Parts:

1. **Divisor Counting**: By iterating up to the square root, we efficiently find all divisors. Each divisor `i` has a pair `(number / i)`.
2. **Optimization for Perfect Squares**: If `i * i == number`, we only count it once to avoid duplicate counting of the square root.

This program will find an integer in the range from 1 to 10,000 with the maximum divisors and print it along with its divisor count.

Let me know when you're ready for the next exercise!

Exercise 2: Find Integer with Largest Number of Divisors (1 to 100,000) using Multithreading

This exercise is similar to Exercise 1, but it involves finding the integer between 1 and 100,000 with the most divisors using multiple threads for faster computation.

Here, we'll divide the range across threads, compute divisor counts for each range segment, and aggregate the results.

We'll implement this in both Java and C++.

Java Solution

```
import java.util.concurrent.*;

// Class to handle divisor counting in a given range
class DivisorCounter implements Callable<int[]> {
    private final int start;
    private final int end;

    public DivisorCounter(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public int[] call() {
        int maxDivisors = 0;
        int numberWithMaxDivisors = 0;

        for (int i = start; i <= end; i++) {
            int divisorCount = countDivisors(i);
            if (divisorCount > maxDivisors) {
                maxDivisors = divisorCount;
                numberWithMaxDivisors = i;
            }
        }

        return new int[]{numberWithMaxDivisors, maxDivisors};
    }

    // Method to count divisors of a given number
```

```
    private int countDivisors(int number) {
        int count = 0;
        for (int i = 1; i * i <= number; i++) {
            if (number % i == 0) {
                count += 2;
                if (i * i == number) {
                    count--;
                }
            }
        }
        return count;
    }
}

public class LargestDivisorWithThreads {
    public static void main(String[] args) throws Interrupted
Exception, ExecutionException {
        int maxThreads = 8;  // Use 8 threads for the calcula
tion
        ExecutorService executor = Executors.newFixedThreadPo
ol(maxThreads);
        int range = 100000 / maxThreads;

        // Track start time for elapsed time calculation
        long startTime = System.currentTimeMillis();

        // Create tasks for each thread
        Future<int[]>[] futures = new Future[maxThreads];
        for (int i = 0; i < maxThreads; i++) {
            int start = i * range + 1;
            int end = (i == maxThreads - 1) ? 100000 : start
+ range - 1;
            futures[i] = executor.submit(new DivisorCounter(s
tart, end));
        }
```

```
        // Find the max divisor result among threads
        int maxDivisors = 0;
        int numberWithMaxDivisors = 0;
        for (Future<int[]> future : futures) {
            int[] result = future.get();
            if (result[1] > maxDivisors) {
                maxDivisors = result[1];
                numberWithMaxDivisors = result[0];
            }
        }

        // Shutdown executor and calculate elapsed time
        executor.shutdown();
        long elapsedTime = System.currentTimeMillis() - start
Time;

        // Output the result
        System.out.println("Number with the largest number of
divisors: " + numberWithMaxDivisors);
        System.out.println("Number of divisors: " + maxDiviso
rs);
        System.out.println("Elapsed time: " + elapsedTime + "
ms");
    }
}
```

C++ Solution

```
#include <iostream>
#include <thread>
#include <vector>
#include <cmath>
#include <mutex>
#include <chrono>
using namespace std;
```

```
struct Result {
    int number;
    int divisorCount;
};

mutex mtx; // Mutex for shared result update
Result globalResult = {0, 0}; // Global result to store max d
ivisor information

// Function to count divisors of a given number
int countDivisors(int number) {
    int count = 0;
    for (int i = 1; i * i <= number; i++) {
        if (number % i == 0) {
            count += 2;
            if (i * i == number) {
                count--;
            }
        }
    }
    return count;
}

// Thread function to find max divisors in a given range
void findMaxDivisors(int start, int end) {
    Result localResult = {0, 0};

    for (int i = start; i <= end; i++) {
        int divisors = countDivisors(i);
        if (divisors > localResult.divisorCount) {
            localResult = {i, divisors};
        }
    }

    // Lock and update the global result if the local result
```

```
is greater
    lock_guard<mutex> lock(mtx);
    if (localResult.divisorCount > globalResult.divisorCount)
{
        globalResult = localResult;
    }
}

int main() {
    int maxThreads = 8;
    int range = 100000 / maxThreads;
    vector<thread> threads;

    // Track start time
    auto startTime = chrono::high_resolution_clock::now();

    // Create and start threads
    for (int i = 0; i < maxThreads; i++) {
        int start = i * range + 1;
        int end = (i == maxThreads - 1) ? 100000 : start + ra
nge - 1;
        threads.emplace_back(findMaxDivisors, start, end);
    }

    // Join all threads
    for (auto& th : threads) {
        th.join();
    }

    // Track end time and calculate elapsed time
    auto endTime = chrono::high_resolution_clock::now();
    auto elapsedTime = chrono::duration_cast<chrono::millisec
onds>(endTime - startTime).count();

    // Output the result
    cout << "Number with the largest number of divisors: " <<
```

```
globalResult.number << endl;
    cout << "Number of divisors: " << globalResult.divisorCou
nt << endl;
    cout << "Elapsed time: " << elapsedTime << " ms" << endl;

    return 0;
}
```

Explanation of Key Parts:

1. **Dividing the Range**: We divide the range of 1 to 100,000 across multiple threads. Each thread handles a specific segment.
2. **Result Aggregation**: Each thread computes the max divisors for its range and, using a mutex (C++) or `Future` tasks (Java), the results are consolidated to find the global maximum.
3. **Elapsed Time**: The elapsed time for the computation is calculated to demonstrate the performance gain from multithreading.

This code will output the number with the maximum divisors, the divisor count, and the elapsed time.

Let me know when you'd like to proceed to the next exercise!

Exercise 3: Museum Entry-Exit Control

In this exercise, we have four components in a museum:

1. **East (Entrance)**: Tracks visitors entering.
2. **West (Exit)**: Tracks visitors exiting.
3. **Control**: Manages entry and exit permissions based on whether the museum is open or closed.
4. **Director**: Opens and closes the museum.

We will use Java's `Monitor` mechanism to create thread-safe methods in `Control` that allow entry when the museum is open and only exit when it's closed. We'll use synchronized methods to control access, with `Control` acting as the monitor.

Java Solution

```
class MuseumControl {
    private boolean open = false; // Museum initially closed
    private int visitorsInside = 0; // Track number of visito
rs inside

    // Synchronized method to signal museum opening
    public synchronized void openMuseum() {
        open = true;
        System.out.println("Museum is now open.");
        notifyAll(); // Notify all waiting entry threads
    }

    // Synchronized method to allow entry only if museum is o
pen
    public synchronized void enterMuseum() throws Interrupted
Exception {
        while (!open) {
            wait(); // Wait until museum opens
        }
        visitorsInside++;
        System.out.println("Visitor entered. Visitors inside:
" + visitorsInside);
    }

    // Synchronized method to allow exit even if museum is cl
osed
    public synchronized void exitMuseum() throws InterruptedE
xception {
        while (visitorsInside == 0) {
            wait(); // Wait if no visitors are inside
        }
        visitorsInside--;
        System.out.println("Visitor exited. Visitors inside:
" + visitorsInside);
```

```
        notifyAll(); // Notify any waiting threads
    }

    // Synchronized method to signal museum closing
    public synchronized void closeMuseum() {
        open = false;
        System.out.println("Museum is now closed. Only exits
allowed.");
    }
}

// Runnable class for Director to open and close the museum
class Director implements Runnable {
    private final MuseumControl control;

    public Director(MuseumControl control) {
        this.control = control;
    }

    @Override
    public void run() {
        try {
            control.openMuseum();
            Thread.sleep(5000); // Museum open for a while
            control.closeMuseum();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Runnable class for East (entrance) to allow visitors to en
ter
class East implements Runnable {
    private final MuseumControl control;
```

```
    public East(MuseumControl control) {
        this.control = control;
    }

    @Override
    public void run() {
        try {
            while (true) {
                control.enterMuseum();
                Thread.sleep(1000); // Simulate visitor enter
ing
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Runnable class for West (exit) to allow visitors to exit
class West implements Runnable {
    private final MuseumControl control;

    public West(MuseumControl control) {
        this.control = control;
    }

    @Override
    public void run() {
        try {
            while (true) {
                control.exitMuseum();
                Thread.sleep(1500); // Simulate visitor exiti
ng
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }
}

public class Museum {
    public static void main(String[] args) {
        MuseumControl control = new MuseumControl();
        Thread director = new Thread(new Director(control));
        Thread east = new Thread(new East(control));
        Thread west = new Thread(new West(control));

        director.start(); // Start Director to open and close
the museum
        east.start(); // Start East for entrance
        west.start(); // Start West for exit
    }
}
```

Explanation of Key Parts:

1. **Synchronization with `MuseumControl` Monitor**: `MuseumControl` uses synchronized methods to manage entry, exit, opening, and closing of the museum.
2. **Conditional Waiting with `wait()` and `notifyAll()`**:
    - `enterMuseum()` waits if the museum is closed.
    - `exitMuseum()` waits if there are no visitors inside.
3. **Director Controls Museum State**:
    - `Director` opens the museum, allowing `East` to permit entries.
    - When the museum is closed, only exits are permitted.

This code controls the flow of visitors based on the museum's open or closed state. Let me know when you're ready for the next exercise!

Exercise 4: Barrier Synchronization Monitor

In this exercise, we need to implement a Java `Barrier` monitor that ensures a specified number `N` of threads must call `sync` before any of them can proceed. Only after `N` threads have reached the barrier can all proceed.

This problem is solved using synchronization where each thread waits until all `N` threads reach the barrier.

Java Solution

```
class Barrier {
    private final int maxThreads; // Total threads needed to
release the barrier
    private int waitingThreads = 0; // Count of threads curre
ntly at the barrier

    public Barrier(int maxThreads) {
        this.maxThreads = maxThreads;
    }

    // Synchronized method to block threads until maxThreads
have called sync
    public synchronized void sync() throws InterruptedExcepti
on {
        waitingThreads++; // Thread arrives at the barrier

        // If not all threads are at the barrier, wait
        if (waitingThreads < maxThreads) {
            wait();
        } else {
            // Last thread to arrive will release all threads
            waitingThreads = 0; // Reset for potential reuse
            notifyAll(); // Release all waiting threads
        }
    }
}
```

```
// Runnable class to simulate threads reaching a barrier
class BarrierTask implements Runnable {
    private final Barrier barrier;

    public BarrierTask(Barrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName
() + " has reached the barrier.");
            barrier.sync(); // Thread waits until the barrier
is released
            System.out.println(Thread.currentThread().getName
() + " has crossed the barrier.");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public class BarrierExample {
    public static void main(String[] args) {
        int numberOfThreads = 5;
        Barrier barrier = new Barrier(numberOfThreads);

        // Start multiple threads
        for (int i = 0; i < numberOfThreads; i++) {
            new Thread(new BarrierTask(barrier), "Thread-" +
(i + 1)).start();
        }
    }
}
```

Explanation of Key Parts:

1. **Barrier Synchronization Logic**:
   - The `sync` method blocks threads until the required number of `maxThreads` has called it. When the count reaches `maxThreads`, all threads are released at once.
2. **Thread Waiting with `wait()`**:
   - Each thread reaching the barrier waits if the count hasn't reached `maxThreads`.
3. **Thread Release with `notifyAll()`**:
   - When the `maxThreads` count is reached, `notifyAll()` releases all waiting threads simultaneously.

This solution ensures that no thread proceeds past the barrier until the specified number has reached it. Let me know if you're ready to go on to the next exercise!

Exercise 5: Savings Account Problem

In this exercise, we implement a `SavingsAccount` class to handle deposits and withdrawals by multiple people, ensuring the balance does not go negative. We'll use synchronization to manage access to the account's balance.

Java Solution

```
class SavingsAccount {
    private int balance;

    public SavingsAccount(int initialBalance) {
        this.balance = initialBalance;
    }

    // Synchronized deposit method to safely add money to the
account
    public synchronized void deposit(int amount) {
```

```
        balance += amount;
        System.out.println(Thread.currentThread().getName() +
" deposited: " + amount + ", New balance: " + balance);
        notifyAll(); // Notify waiting withdrawal threads if
any
    }

    // Synchronized withdraw method to safely subtract money
from the account
    public synchronized void withdraw(int amount) throws Inte
rruptedException {
        // Wait if there are insufficient funds
        while (balance < amount) {
            System.out.println(Thread.currentThread().getName
() + " waiting to withdraw: " + amount + ", Current balance:
" + balance);
            wait(); // Wait until there is enough balance
        }
        balance -= amount;
        System.out.println(Thread.currentThread().getName() +
" withdrew: " + amount + ", New balance: " + balance);
    }
}

// Runnable class for a person to make deposits and withdrawa
ls
class AccountHolder implements Runnable {
    private final SavingsAccount account;
    private final int amount;
    private final boolean deposit;

    public AccountHolder(SavingsAccount account, int amount,
boolean deposit) {
        this.account = account;
        this.amount = amount;
        this.deposit = deposit;
```

```
    }

    @Override
    public void run() {
        try {
            if (deposit) {
                account.deposit(amount);
            } else {
                account.withdraw(amount);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public class SavingsAccountExample {
    public static void main(String[] args) {
        SavingsAccount account = new SavingsAccount(100); //
Initial balance of 100

        // Creating multiple threads for deposit and withdraw
al
        Thread depositor1 = new Thread(new AccountHolder(acco
unt, 50, true), "Depositor-1");
        Thread depositor2 = new Thread(new AccountHolder(acco
unt, 100, true), "Depositor-2");
        Thread withdrawer1 = new Thread(new AccountHolder(acc
ount, 30, false), "Withdrawer-1");
        Thread withdrawer2 = new Thread(new AccountHolder(acc
ount, 120, false), "Withdrawer-2");

        // Start threads
        depositor1.start();
        depositor2.start();
        withdrawer1.start();
```

```
        withdrawer2.start();
    }
}
```

Explanation of Key Parts:

1. **Synchronized `deposit` and `withdraw` Methods**:
    - The `deposit` method increases the balance and notifies all waiting threads if a withdrawal was waiting.
    - The `withdraw` method waits if the balance is insufficient, ensuring that the account does not go negative.
2. **Conditional Waiting with `wait()` and Notification with `notifyAll()`**:
    - `withdraw` uses a `while` loop to check if there's enough balance. If not, it waits until a deposit occurs.
    - `notifyAll()` in `deposit` awakens waiting threads when the balance changes.

This approach ensures safe access and modification of the shared account balance. Let me know if you have more questions or need additional examples!

Here's a quick guide to understanding the core concepts of concurrency in Java and C++, covering the main points that will give you a solid foundation for most concurrency-related tasks.

Core Concurrency Concepts in Java and C++

1. Threads:

- **Java**: Java has built-in thread support with the `Thread` class and the `Runnable` interface. `ExecutorService` provides thread pools for efficient resource use.
- **C++**: C++11 introduced `std::thread`, which allows for creating threads. Threads can be joined or detached, and their lifecycle is controlled using RAII principles.

2. Synchronization (Mutual Exclusion):

- **Java**:
  - Uses the `synchronized` keyword to lock methods or code blocks, preventing multiple threads from accessing critical sections at the same time.
  - `ReentrantLock` provides more flexible locking mechanisms than `synchronized` and includes features like timed locking.
- **C++**:
  - Provides `std::mutex` for mutual exclusion, and `std::lock_guard` or `std::unique_lock` as RAII wrappers for mutexes.
  - `std::recursive_mutex` allows reentrant locking, useful in recursive functions.

3. Condition Variables (Waiting and Signaling):

- **Java**:
  - Each object has a built-in monitor, allowing `wait()`, `notify()`, and `notifyAll()` for condition synchronization.
  - `Condition` objects in `java.util.concurrent.locks` offer more control when used with `Lock`.
- **C++**:
  - `std::condition_variable` allows threads to wait for or signal specific conditions.
  - `notify_one` wakes a single waiting thread, while `notify_all` wakes all waiting threads.

4. Atomic Operations:

- **Java**: The `java.util.concurrent.atomic` package provides atomic classes like `AtomicInteger` and `AtomicReference`, which support lock-free thread-safe operations.
- **C++**: `std::atomic` provides atomic operations for simple data types, allowing safe increment, decrement, and other atomic operations without using locks.

5. Deadlocks and Livelocks:

- **Deadlock**: Occurs when two or more threads are waiting on each other's locks, causing a cycle that prevents progress.
- **Livelock**: Similar to deadlock, but threads keep changing their state in response to each other without making progress.
- **Prevention**: Use a consistent lock ordering, avoid nested locks, and use timeouts where possible.

6. Thread Lifecycle Management:

- **Java**:
  - `Thread.start()` initiates a thread, `Thread.join()` waits for a thread to finish, and `Thread.interrupt()` signals it to stop.
  - `ExecutorService` provides lifecycle methods like `shutdown()` and `awaitTermination()` for managing thread pools.
- **C++**:
  - `thread.join()` blocks until the thread finishes, while `thread.detach()` allows it to run independently.
  - Use RAII principles to avoid `std::terminate` if threads go out of scope without joining.

7. Memory Visibility (Happens-Before Relationship):

- **Java**: The Java Memory Model (JMM) defines the `volatile` keyword, ensuring visibility of changes to a variable across threads.
  - `synchronized` blocks also establish a happens-before relationship, ensuring visibility of variables modified inside synchronized blocks.
- **C++**: The `std::atomic` and `memory_order` settings control memory visibility and ordering.
  - Use `std::memory_order` options (`memory_order_relaxed`, `memory_order_acquire`, etc.) to control how operations are observed across threads.

8. Concurrency Utilities:

- **Java**: The `java.util.concurrent` package includes:

    - **Thread-safe collections** like `ConcurrentHashMap`.
    - **CountDownLatch** and **CyclicBarrier** for synchronizing multiple threads at a certain point.
    - **Semaphore** for controlling access to resources.
- **C++**:
    - C++ lacks built-in equivalents to all Java concurrency utilities, but libraries like Intel TBB or Boost offer features like barriers and thread-safe collections.
    - `std::barrier` and `std::latch` were introduced in C++20 for similar synchronization purposes.

9. Thread Safety and Best Practices:

- Avoid sharing mutable data between threads whenever possible.
- Use higher-level abstractions (e.g., thread-safe collections, atomic variables) instead of manual synchronization.
- Prefer immutable objects and local variables over shared state to simplify thread safety.
- Always release locks in a `finally` block (Java) or use RAII (C++).

These are the foundational principles that cover most concurrency requirements in Java and C++. Starting with these, you'll understand how to safely work with shared data and avoid the common pitfalls of multi-threaded programming. Let me know if you want any examples on a specific topic!