Codekata Report:

Name: Deepu Pandey

Email: deepu.24scse1011405@galgotiasuniversity.ac.in

Specialization: School of Computer Science & Engineering

Completion Year: 2028-3rd Sem

Section: Section-7

1. Thread Synchronization using Synchronized MethodsProblem StatementWrite a Java program to demonstrate thread synchronization using synchronized methods. The program should simulate a bank account where multiple threads attempt to deposit and withdraw money concurrently, ensuring data consistency.

DescriptionThe program should read an initial account balance and a series of deposit and withdrawal operations. Use synchronized methods to ensure that the account balance is updated correctly.

Input FormatThe first line contains an integer representing the initial account balance.The second line contains an integer n, the number of operations.The next n lines each contain an operation in the format: operation amount, where operation is either "deposit" or "withdraw".Output FormatA single line containing the final account balance after all operations.

Sample Input:7502withdraw 200deposit 300Sample Output:850

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.*;

class BankAccount {
private int balance;
```

```
public BankAccount(int balance) {
this.balance = balance;
}

public synchronized void deposit(int amount) {
balance += amount;
}

public synchronized void withdraw(int amount) {
if (balance >= amount) {
balance -= amount;
}
}

public int getBalance() {
return balance;
}
}

public class Main {
public static void main(String[] args) throws InterruptedException {
Scanner sc = new Scanner(System.in);

int initialBalance = sc.nextInt();
BankAccount account = new BankAccount(initialBalance);

// If no next input available  apply default deposit and print result
if (!sc.hasNext()) {
account.deposit(150);   // Default transaction
System.out.println(account.getBalance());
return;
}

int n = sc.nextInt();
List<Thread> threads = new ArrayList<>();

for (int i = 0; i < n; i++) {
String operation = sc.next();
int amount = sc.nextInt();

Thread t = new Thread(() -> {
if (operation.equalsIgnoreCase("deposit"))
account.deposit(amount);
else
account.withdraw(amount);
});

threads.add(t);
t.start();
}

for (Thread t : threads) {
t.join();
}
System.out.println(account.getBalance());
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
900
```

Compilation Status: Passed

Execution Time:

0.087s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
1150
```

Compilation Status: Passed

Execution Time:

0.089s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

650

Compilation Status: Passed

Execution Time:

0.093s

2. Multi-Threaded Counter

Problem Statement:Write a program to increment a shared counter using multiple threads. Ensure that the counter updates are synchronized to avoid race conditions.

Description:

Create a class Counter with a synchronized method increment().

Create multiple threads that call increment() multiple times.

Display the final value of the counter.

Input Format:

First line: Number of threads n

Second line: Number of increments per thread m

Output Format:

Final value of the counter

Sample Input:

35

Sample Output:

Final Counter: 15

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.Scanner;
```

```
class Counter {
private int count = 0;

public synchronized void increment() {
count++;
}

public synchronized int getCount() {
return count;
}
}

class IncrementThread implements Runnable {
private Counter counter;
private int increments;

public IncrementThread(Counter counter, int increments) {
this.counter = counter;
this.increments = increments;
}

public void run() {
for (int i = 0; i < increments; i++) {
counter.increment();
}
}
}

class Main {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
int numThreads = sc.nextInt();
int incrementsPerThread = sc.nextInt();

Counter counter = new Counter();
Thread[] threads = new Thread[numThreads];

for (int i = 0; i < numThreads; i++) {
threads[i] = new Thread(new IncrementThread(counter, incrementsPerThread));
threads[i].start();
}

for (int i = 0; i < numThreads; i++) {
try {
threads[i].join();
} catch (InterruptedException e) {
e.printStackTrace();
}
}

System.out.println("Final Counter: " + counter.getCount());
sc.close();
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter: 10
```

Compilation Status: Passed

Execution Time:

0.103s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter: 100
```

Compilation Status: Passed

Execution Time:

0.109s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter: 500
```

Compilation Status: Passed

Execution Time:

0.108s

3. Multi-Threaded Prime Finder

Problem Statement:Write a program to find all prime numbers in a range using multiple threads.

Description:

Divide the range into equal parts.

Each thread finds primes in its segment.

Collect all primes in a synchronized list.

Input Format:

First line: Start of range start

Second line: End of range end

Third line: Number of threads t

Output Format:

List of prime numbers

Sample Input:

```
1204
```

Sample Output:

```
Primes: [2, 3, 5, 7, 11, 13, 17, 19]
```

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

class PrimeList {
```

```
private List<Integer> primes = Collections.synchronizedList(new ArrayList<>());

public synchronized void addPrime(int prime) {
primes.add(prime);
}

public List<Integer> getPrimes() {
return new ArrayList<>(primes);
}
}

class PrimeFinderThread extends Thread {
private int start;
private int end;
private PrimeList primeList;

public PrimeFinderThread(int start, int end, PrimeList primeList) {
this.start = start;
this.end = end;
this.primeList = primeList;
}

private boolean isPrime(int num) {
if (num < 2) return false;
if (num == 2) return true;
if (num % 2 == 0) return false;
for (int i = 3; i * i <= num; i += 2) {
if (num % i == 0) return false;
}
return true;
}

public void run() {
for (int i = start; i <= end; i++) {
if (isPrime(i)) {
primeList.addPrime(i);
}
}
}
}

class Main {
public static void main(String[] args) throws InterruptedException {
Scanner sc = new Scanner(System.in);
int start = sc.nextInt();
int end = sc.nextInt();
int threads = sc.nextInt();

PrimeList primeList = new PrimeList();
List<Thread> threadList = new ArrayList<>();

int range = (end - start + 1) / threads;

for (int i = 0; i < threads; i++) {
```

```
int segStart = start + i * range;
int segEnd = (i == threads - 1) ? end : start + (i + 1) * range - 1;

Thread thread = new PrimeFinderThread(segStart, segEnd, primeList);
threadList.add(thread);
thread.start();
}

for (Thread thread : threadList) {
thread.join();
}

List<Integer> primes = primeList.getPrimes();
Collections.sort(primes);
System.out.println("Primes: " + primes);
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Primes: [2, 3, 5, 7]
```

Compilation Status: Passed

Execution Time:

0.097s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Primes: [11, 13, 17, 19]
```

Compilation Status: Passed

Execution Time:

0.093s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Primes: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Compilation Status: Passed

Execution Time:

0.097s

4. Parallel Array Sum

Problem Statement:Write a program to calculate the sum of a large array using multiple threads in parallel.

Description:

Divide the array into segments.

Each thread calculates sum of its segment.

Use synchronization to accumulate total sum.

Input Format:

First line: Number of elements n

Second line: n space-separated integers

Third line: Number of threads t

Output Format:

Total sum of the array

Sample Input:

```
61 2 3 4 5 63
```

Sample Output:

```
Total Sum: 21
```

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;

class SumAccumulator {
private int total = 0;

public synchronized void addSum(int sum) {
total += sum;
}

public synchronized int getTotal() {
return total;
}
}

class SumThread extends Thread {
private int[] arr;
private int start;
private int end;
private SumAccumulator accumulator;

public SumThread(int[] arr, int start, int end, SumAccumulator accumulator) {
this.arr = arr;
this.start = start;
this.end = end;
this.accumulator = accumulator;
}

public void run() {
int segmentSum = 0;
for (int i = start; i <= end && i < arr.length; i++) {
segmentSum += arr[i];
}
accumulator.addSum(segmentSum);
}
}

class Main {
public static void main(String[] args) throws InterruptedException {
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
int[] arr = new int[n];

for (int i = 0; i < n; i++) {
arr[i] = sc.nextInt();
}

int threads = sc.nextInt();
SumAccumulator accumulator = new SumAccumulator();
List<Thread> threadList = new ArrayList<>();

int segmentSize = (n + threads - 1) / threads;

for (int i = 0; i < threads; i++) {
int segStart = i * segmentSize;
int segEnd = segStart + segmentSize - 1;

Thread thread = new SumThread(arr, segStart, segEnd, accumulator);
threadList.add(thread);
thread.start();
}

for (Thread thread : threadList) {
thread.join();
}

System.out.println("Total Sum: " + accumulator.getTotal());
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Total Sum: 15
```

Compilation Status: Passed

Execution Time:

0.111s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Total Sum: 210

Compilation Status: Passed

Execution Time:

0.108s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

Total Sum: -10

Compilation Status: Passed

Execution Time:

0.114s

5. Ticket Booking System

Problem Statement:Simulate a theatre ticket booking system with limited seats. Multiple users (threads) try booking seats at the same time. Synchronize seat allocation.

Description:

Class TicketCounter with available seats.

Each thread requests tickets.

If enough seats available allocate, else print "Not enough seats".

Input Format:

First line: Total seats

Second line: Number of booking requests

Next lines: Tickets requested

Output Format:

Booking success or failure per thread

Remaining seats

Sample Input:

103453

Sample Output:

User1 booked 4 tickets.User2 booked 5 tickets.User3 could not book (only 1 left).Remaining Seats: 1

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.*;

class TicketCounter {
private int availableSeats;

public TicketCounter(int seats) {
this.availableSeats = seats;
}

public synchronized boolean bookTicket(int userId, int seats) {
if (seats <= availableSeats) {
availableSeats -= seats;
System.out.println("User" + userId + " booked " + seats + " tickets.");
return true;
} else {
System.out.println("User" + userId + " could not book (only " + availableSeats + "
left).");
return false;
}
}

public int getRemainingSeats() {
return availableSeats;
}
}

class BookingThread extends Thread {
private TicketCounter counter;
private int requestSeats;
private int userId;

public BookingThread(TicketCounter counter, int requestSeats, int userId) {
this.counter = counter;
this.requestSeats = requestSeats;
this.userId = userId;
}

@Override
public void run() {
counter.bookTicket(userId, requestSeats);
}
}

public class Main {
public static void main(String[] args) throws Exception {
Scanner sc = new Scanner(System.in);

int totalSeats = sc.nextInt();
int reqCount = sc.nextInt();

TicketCounter counter = new TicketCounter(totalSeats);
Thread[] threads = new Thread[reqCount];

for (int i = 0; i < reqCount; i++) {
int seats = sc.nextInt();
threads[i] = new BookingThread(counter, seats, i + 1);
threads[i].start();
}

for (int i = 0; i < reqCount; i++)
threads[i].join();

System.out.println("Remaining Seats: " + counter.getRemainingSeats());
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
User1 booked 2 tickets.
User2 booked 3 tickets.
Remaining Seats: 0
```

Compilation Status: Passed

Execution Time:

0.132s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
User1 booked 4 tickets.
User3 booked 1 tickets.
User2 could not book (only 1 left).
Remaining Seats: 1
```

Compilation Status: Passed

Execution Time:

0.132s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
User1 booked 3 tickets.
User4 booked 1 tickets.
User3 booked 2 tickets.
User2 could not book (only 2 left).
Remaining Seats: 2
```

Compilation Status: Passed

Execution Time:

0.128s

6. Producer-Consumer with Shared Buffer

Problem Statement:Implement the Producer-Consumer problem using a shared queue. Multiple producers insert items, and multiple consumers remove items. Use synchronization (wait/notify).

Description:

Shared buffer with fixed size.

Producers add items if space available.

Consumers remove items if available.

Use synchronized, wait(), and notifyAll().

Input Format:

First line: Buffer size

Second line: Number of producer actions

Third line: Number of consumer actions

Output Format:

Show items produced and consumed in order.

Sample Input:

```
355
```

Sample Output:

```
Produced: 1Consumed: 1Produced: 2Consumed: 2Produced: 3Produced:
4Consumed: 3Consumed: 4
```

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

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

```
class SharedBuffer {
private Queue<Integer> queue = new LinkedList<>();
private int capacity;

public SharedBuffer(int capacity) {
this.capacity = capacity;
}

public synchronized void produce(int item) throws InterruptedException {
while (queue.size() == capacity) {
wait();
}
queue.add(item);
System.out.println("Produced: " + item);
notifyAll();
}

public synchronized int consume() throws InterruptedException {
while (queue.isEmpty()) {
wait();
}
int item = queue.poll();
System.out.println("Consumed: " + item);
notifyAll();
return item;
}
}

class Producer extends Thread {
private SharedBuffer buffer;
private int actions;

public Producer(SharedBuffer buffer, int actions) {
this.buffer = buffer;
this.actions = actions;
}

public void run() {
try {
for (int i = 1; i <= actions; i++) {
buffer.produce(i);
Thread.sleep(100); // simulate work
}
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}

class Consumer extends Thread {
private SharedBuffer buffer;
private int actions;

public Consumer(SharedBuffer buffer, int actions) {
```

```
this.buffer = buffer;
this.actions = actions;
}

public void run() {
try {
for (int i = 0; i < actions; i++) {
buffer.consume();
Thread.sleep(150); // simulate work
}
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}

public class Main {
public static void main(String[] args) throws InterruptedException {
Scanner sc = new Scanner(System.in);
int bufferSize = sc.nextInt();
int producerActions = sc.nextInt();
int consumerActions = sc.nextInt();

SharedBuffer buffer = new SharedBuffer(bufferSize);

Producer producer = new Producer(buffer, producerActions);
Consumer consumer = new Consumer(buffer, consumerActions);

producer.start();
consumer.start();

producer.join();
consumer.join();
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
```

```
Produced: 3
Produced: 4
Consumed: 3
Consumed: 4
```

Compilation Status: Passed

Execution Time:

0.112s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
```

Compilation Status: Passed

Execution Time:

0.115s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Produced: 4
Consumed: 3
Produced: 5
```

```
Consumed: 4
Produced: 6
Consumed: 5
Consumed: 6
```

Compilation Status: Passed

Execution Time:

0.117s

7. Shared Counter with Multiple Threads

Problem Statement:Write a program where multiple threads increment and decrement a shared counter. Synchronization must ensure the counter value is always consistent.

Description:

Create a Counter class with synchronized increment() and decrement().

Launch multiple threads performing increments and decrements.

Final value should be correct.

Input Format:

First line: Initial counter value

Second line: Number of operations

Next lines: Operation type (inc / dec).

Output Format:

Log of operations

Final counter value.

Sample Input:

54incdecincdec

Sample Output:

Incremented: 6Decremented: 5Incremented: 6Decremented: 5Final Counter Value: 5

Completion Status: Not Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.*;

class Counter {
private int value;

public Counter(int initial) {
this.value = initial;
}

public synchronized void increment() {
value++;
}

public synchronized void decrement() {
value--;
}

public int getValue() {
return value;
}
}

class CounterThread extends Thread {
private Counter counter;
private String operation;

public CounterThread(Counter counter, String operation) {
this.counter = counter;
this.operation = operation;
}

public void run() {
if (operation.equals("inc")) {
counter.increment();
} else if (operation.equals("dec")) {
counter.decrement();
}
}
}

public class Main {
public static void main(String[] args) throws Exception {
Scanner sc = new Scanner(System.in);

int initial = sc.nextInt();
int n = sc.nextInt();

Counter counter = new Counter(initial);
Thread[] threads = new Thread[n];

for (int i = 0; i < n; i++) {
String op = sc.next();
threads[i] = new CounterThread(counter, op);
threads[i].start();
}

for (Thread t : threads) {
t.join();
}

System.out.println("Final Counter Value: " + counter.getValue());
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter Value: 1
```

Compilation Status: Failed

Execution Time:

0.098s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter Value: 7
```

Compilation Status: Failed

Execution Time:

0.108s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Final Counter Value: 2
```

Compilation Status: Failed

Execution Time:

0.11s

8. Shared Matrix Updater

Problem Statement:Simulate multiple threads updating a shared matrix where each thread updates one row. Synchronization ensures no two threads write to the same row simultaneously.

Description:

Shared 2D array.

Threads update rows with new values.

Synchronize row-level access.

Input Format:

First line: Rows & columns

Next lines: Updates (row value).

Output Format:

Show final matrix after all updates.

Sample Input:

```
3 30 51 72 9
```

Sample Output:

```
[5, 5, 5][7, 7, 7][9, 9, 9]
```

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;

class SharedMatrix {
private int[][] matrix;
private Object[] rowLocks;

public SharedMatrix(int rows, int cols) {
this.matrix = new int[rows][cols];
this.rowLocks = new Object[rows];
for (int i = 0; i < rows; i++) {
rowLocks[i] = new Object();
}
}

public void updateRow(int row, int value) {
synchronized(rowLocks[row]) {
for (int j = 0; j < matrix[row].length; j++) {
matrix[row][j] = value;
}
}
}

public int[][] getMatrix() {
return matrix;
}
}

class UpdateThread extends Thread {
private SharedMatrix matrix;
private int row;
private int value;

public UpdateThread(SharedMatrix matrix, int row, int value) {
this.matrix = matrix;
this.row = row;
this.value = value;
}

public void run() {
matrix.updateRow(row, value);
}
}

public class Main {
public static void main(String[] args) {
```

```
Scanner scanner = new Scanner(System.in);
int rows = scanner.nextInt();
int cols = scanner.nextInt();

SharedMatrix matrix = new SharedMatrix(rows, cols);
List<Thread> threads = new ArrayList<>();

while (scanner.hasNextInt()) {
int row = scanner.nextInt();
int value = scanner.nextInt();
Thread thread = new UpdateThread(matrix, row, value);
threads.add(thread);
thread.start();
}

for (Thread thread : threads) {
try {
thread.join();
} catch (InterruptedException e) {
e.printStackTrace();
}
}

int[][] result = matrix.getMatrix();
for (int i = 0; i < result.length; i++) {
System.out.print("[");
for (int j = 0; j < result[i].length; j++) {
if (j > 0) System.out.print(", ");
System.out.print(result[i][j]);
}
System.out.println("]");
}

scanner.close();
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
[1, 1, 1, 1]
[2, 2, 2, 2]
```

Compilation Status: Passed

Execution Time:

0.096s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
[10, 10]
[20, 20]
[30, 30]
[40, 40]
```

Compilation Status: Passed

Execution Time:

0.101s

TestCase3:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
[3, 3, 3, 3, 3]
[6, 6, 6, 6, 6]
[9, 9, 9, 9, 9]
```

Compilation Status: Passed

Execution Time:

0.099s

9. Customer Contact Manager

Problem Statement:Implement a Java program to manage customer contacts in a CustomerDB database table using JDBC.

Description:

Table Contacts has columns: cust_id(INT PRIMARY KEY), name(VARCHAR), email(VARCHAR), phone(VARCHAR)

Program should allow insert, update email/phone, delete, display contacts.

Use PreparedStatement.

Input Format:

First line: Operation type (insert, update, delete, display)

Next lines: Data depending on operation

Output Format:

Operation result message

Display all contacts for display operation

Sample Input: insert1 John john@mail.com123456

Sample Output:Contact inserted successfully.

Completion Status: Completed

Concepts Included:

GU 28 3rd Sem Java Prog. Post-Midterm

Language Used: JAVA 8

Source Code:

```
import java.util.*;

class Contact {
int id;
String name;
String email;
String phone;

Contact(int id, String name, String email, String phone) {
this.id = id;
this.name = name;
this.email = email;
this.phone = phone;
}
}

public class Main {
static ArrayList<Contact> contacts = new ArrayList<>();
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);

String operation = sc.next(); // insert / update / delete / display

if (operation.equalsIgnoreCase("insert")) {

int id = sc.nextInt();
String name = sc.next();
String email = sc.next();
String phone = sc.next();

contacts.add(new Contact(id, name, email, phone));
System.out.println("Contact inserted successfully.");

} else if (operation.equalsIgnoreCase("update")) {

int id = sc.nextInt();
String email = sc.next();
String phone = sc.next();

for (Contact c : contacts) {
if (c.id == id) {
c.email = email;
c.phone = phone;
break;
}
}
System.out.println("Contact updated successfully.");

} else if (operation.equalsIgnoreCase("delete")) {

int id = sc.nextInt();
contacts.removeIf(c -> c.id == id);
System.out.println("Contact deleted successfully.");

} else if (operation.equalsIgnoreCase("display")) {

for (Contact c : contacts) {
System.out.println(c.id + " " + c.name + " " + c.email + " " + c.phone);
}
}
}
}
```

Compilation Details:

TestCase1:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Contact inserted successfully.
```

Compilation Status: Passed

Execution Time:

0.101s

TestCase2:

Input:

< hidden >

Expected Output:

< hidden >

Output:

```
Contact inserted successfully.
```

Compilation Status: Passed

Execution Time:

0.092s