RACE CONDITION & SYNCHRONIZATION

1 OBJECTIVES

The main goals of this assignment are:

- Studying race condition read-modify-write and check-then-act.
- Identifying critical sections where race condition may occur.
- Using locks for Mutual Exclusion (ME) in critical sections to prevent race condition.

2 DESCRIPTION AND REQUIREMENTS

This assignment is divided into two mandatory parts, both of which involve the use of threads and occurrence of race conditions, and one optional part. In Part 1, we will simulate transactions in a simple banking system using multiple threads. In Part 2, we will simulate a booking system for reserving tickets to an event using threads. In both parts, we will identify critical sections where a race condition can occur, and we will implement Mutual Exclusion to prevent it. Additionally, there is an optional part, Part 3, in the assignment that deals with Deadlock. It is not mandatory but is highly recommended to give it a thought and think of a possible solution to avoid deadlock.

For a pass grade, you must complete both parts. You may use different application projects or integrate them into the same solution (C# VS) or two packages (Java) in the same project. At the end of Part 2 there is an optional part about deadlock. It is not required for a pass grade.

You may choose to complete the assignment using a console/terminal text-based project or a project with a graphical user interface (GUI). However, GUI is not provided for any part of the assignment, so you will need to create your own GUI if you choose to use one.

For both parts of the assignment, you can choose to develop either a text-based command prompt/terminal application or a GUI-based application. If you opt for the former, you can either have a controller class or directly use the main method in a start class, which can be named **Program** or **Main**. Let's refer to this start class as the **Main** class. In the case of a Windows Form application, the form's code file is where the operations should be initiated.

To ensure that your code is easy to read and understand, it should be well-structured and documented. Use comments to describe the purpose of your classes and methods, and use appropriate names for your classes, methods, and variable names. Follow the general naming conventions used in the programming language you are using. For example, in Java, method names should begin with a lowercase letter, while in C#, they should begin with a capital letter. Additionally, in Java, constant names are usually all caps, whereas in C#, they are named exactly like variables.

It is also recommended that you divide your application into as many classes as possible and use several shorter methods instead of one long method in each class. Always keep Object-Oriented Programming (OOP) in mind. Poorly structured code and solutions that lack meaning may require additional work and resubmission.

3 An Overview of Race Condition

"Read-modify-write" are atomic (indivisible) operations that CPUs can execute. Even the simplest operations in high-level programming languages are compiled into multiple CPU instructions. The problem arises when shared mutable data is accessible by multiple threads that run concurrently.

Thread A

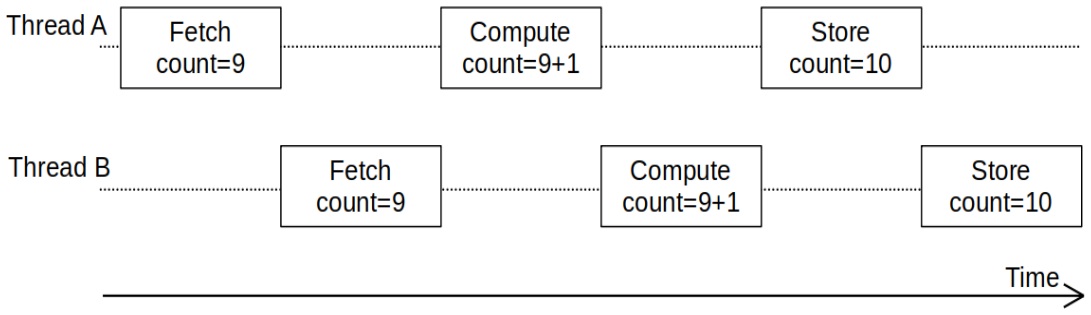
```
count = count + 1
```

Thread B

```
count = count + 1
```

Even a simple line of code, like the one shown in the image above, is compiled into at least three operations: (1) the variable "count" is loaded from memory, (2) its value is increased by 1, and (3) the result is saved back into memory.

When multiple threads run the same code simultaneously, they may attempt to access shared data concurrently, potentially leading to data loss or corruption. The image below illustrates a possible scenario.



As the example above demonstrates, when multiple threads change the value of a shared variable without proper control, the result can be an incorrect value. In the given scenario, thread A assigns a value of 10 to the variable "count," but before it has a chance to save this value, thread B enters the code and reads the old value of 9, resulting in the final value of "count" being 10 instead of 11.

To prevent such inconsistencies, the code can be synchronized so that only one thread is permitted to execute and finish the operation before the next one is allowed to access the shared variable (Mutual Exclusion). With synchronization in place, the final value of "count" should be 11, as expected.

Mutual exclusion (ME) is a mechanism that prevents concurrent execution of code that accesses shared resources to avoid race condition. In C#, the lock keyword can be used for ME, while in Java, the synchronized keyword can be used. To apply synchronization, it is recommended to use it only around the critical sections of code, i.e., sections that access shared resources. Do not include non-critical sections and do not lock a whole object or a method.

Java: It is important to note that the synchronized statement should be used only around the critical section and not with the method definition. This helps in identifying the critical and non-critical parts of the code

C#

```
      private Object lockObj = new Object();
      lock (lockObj) {
               //code
      }
```

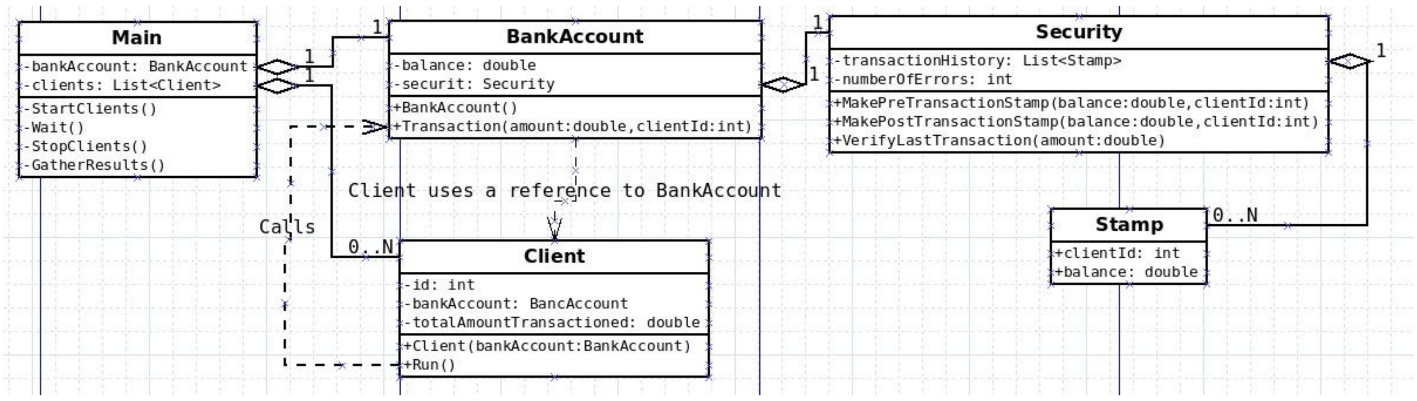
Java: Replace lock in the above code with synchronized and the rest is the same.

```
private Object lockObj = new Object();
synchronize (lockObj){
        //code
}
```

4 PART 1: BANKING TRANSACTIONS

Write a program that simulates banking operations in a bank, allowing multiple threads to deposit or withdraw random amounts from the same bank account. The threads should run in an infinite loop without any pauses. In this application, we intentionally create a read-modify-write type of race condition, which can result in inconsistencies or errors in the final account balance.

Write classes to represent clients, banking transactions and other classes that find necessary. The following class diagram shows a suggested structure of the project although you don't have to follow it in detail.


Main
-bankAccount: BankAccount
-clients: List<Client>
-StartClients()
-Wait()
-StopClients()
-GatherResults()
BankAccount
-balance: double
-securit: Security
+BankAccount()
+Transaction(amount:double,clientId:int)
Security
-transactionHistory: List<Stamp>
-numberOfErrors: int
+MakePreTransactionStamp(balance:double,clientId:int)
+MakePostTransactionStamp(balance:double,clientId:int)
+VerifyLastTransaction(amount:double)
Stamp
+clientId: int
+balance: double
Client
-id: int
-bankAccount: BancAccount
-totalAmountTransactioned: double
+Client(bankAccount:BankAccount)
+Run()
Client uses a reference to BankAccount
Calls
1
1
1
1
1
1
0..N
0..N
1

Each client tries to do transactions on the **BankAccount,** which is the shared resource. By accessing the account in an uncontrolled manner, threads modify the value of the balance, corrupting the shared data. This is the consequence of the race conditions that occur. To illustrate the problem, we make clients keep track of the total amount of all their transactions. With this data in hand for all clients, we can compare it with the balance of the **BankAccount** at the end of the program and analyze the results.

The Client class:

4.1 The Client class represents a client by the Id of the client. The operations expected from the class is to perform a deposit or withdrawal transaction. It represents either a thread object (Java) or has actions assigned to a client thread (C#) and simulates a client performing randomly transactions until it is ordered to stop. You can randomize a Boolean value to decide whether to deposit or withdraw an amount from the **BankAccount**.

4.2 Add a **totalAmountTransactioned** counter in this class to store the amount of each transaction.

Main operations in the client thread class:

```
private double totalAmountTransactioned;

while(operating) {
     account.deposit(amount);
     totalAmountTransactioned += amount;
     }
}
```

The BankAccount class

Due to the occurrence of race conditions, the balance of the account after a transaction may be incorrect and this is obviously an error; we would like to track the occurrences of such errors in the **BankAccount** class.

In order to observe and store information about the correctness of each transaction performed by a client, let us create a **Security** class using another class **Stamp** to store some relevant data, as shown in the above class diagram. This can be done by using a stamp before and after a transaction. The stamp is simply data about the balance before and after the transaction. It stores data about the client making the transaction, type of transaction, e.g., deposit or withdrawal, and the balance.

The balance after a transaction is to be verified in the Security class by checking whether the balance after the transaction is completed matches the balance saved in **BankAccount**. While a transaction is being realized by a client thread, other client threads may have changed the balance of the **BankAccount**.

4.3 Write a method in the **BankAccount** class for performing a security check by using the methods of the Security class. by saving the balance before and after each transaction, as described above. You may choose to keep a history of all transactions or only save the last two values.

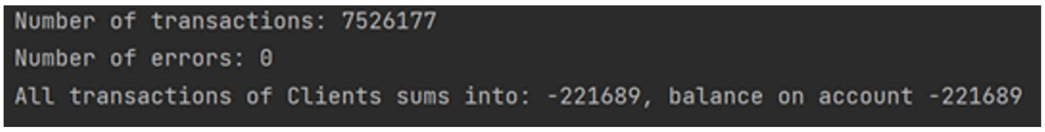
4.4 Add a counter that keeps track of the number of incorrect transactions. Allow the client threads to run for several seconds before stopping them and comparing the expected balance (sum of the total amount transacted in the thread classes) with the actual balance of the account. This comparison will help you identify inconsistencies or errors resulting from race conditions.

Main operations in the **BankAccount** Class:

```
public void Transaction(double amount, int clientId){
         security.MakePreTransactionStamp(balance,
 clientId);
         balance = balance + amount;
 numberOfTransactions ++;
 security.MakePostTransactionStamp(balance, clientId);
 security.VerifyLastTransaction();
 }
```

4.5 Once all client threads have been stopped, calculate the total sum of all transactions performed by the clients, and compare it with the resulting balance in the **BankAccount** class. Due to the race condition introduced in the program, it is expected that a significant number of transaction errors will occur.

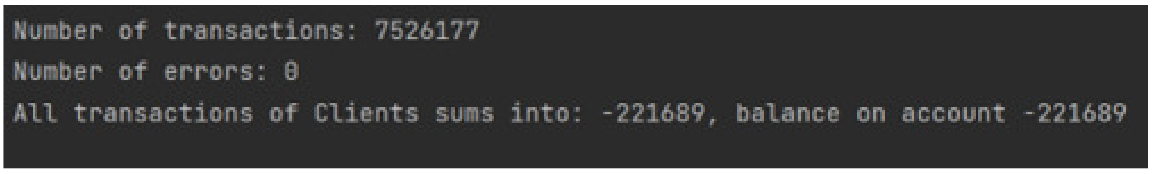
4.6 The expected output of the program is to see the number of transaction errors and compare it with the expected results.



```
Number of transactions: 7526177
Number of errors: 0
All transactions of Clients sums into: -221689, balance on account -221689
```

4.7 In order to prevent race condition, it is necessary to identify all the critical sections in the code where multiple threads could access and modify shared data. Once identified, Mutual Exclusion (ME) can be used to protect these sections. As mentioned earlier, In C#, the lock(object) keyword can be used to achieve ME, while in Java, the synchronized(object) statement can be used..

4.8 After implementing ME, the expected output of the program should be accurate and consistent.



```
Number of transactions: 7526177
Number of errors: 0
All transactions of Clients sums into: -221689, balance on account -221689
```

4.9 Run your application multiple times with varying sleep times inserted at different parts of the code to observe how it affects the outcome. Analyze and explain how race conditions can impact the correctness of shared data, and be prepared to demonstrate how you protect against race conditions by implementing mutual exclusion using locks, when presenting your work.

5 PART 2: TICKET RESERVATION

Write a program that simulates a booking system for a popular event, such as a football game or a music concert. In this scenario, the system experiences a high demand for tickets; when the booking opens, and customers rush to book tickets.

Implement the booking system using multithreading to handle multiple requests from customers simultaneously. Ensure that the program protects against race condition to prevent errors that may arise when multiple threads access shared data, such as the number of available tickets. When presenting your solution, you should be able to explain how your program handles concurrency and how it ensures the correct and consistent behavior of the booking system.

5.1 Clients are separate threads racing to change the status of a seat. To make the situation more realistic, the **SeatManager** class (representing a booking system) should check if the seat is available before setting its status to “booked”. We will see that even such a check is prone to errors, although they are much less frequent than in the bank account scenario. This scenario is the example of check-then-act type of race condition.

5.2 In this scenario, clients rush to book the first seat that they find available. The **SeatManager** class may have just one public method for clients to carry out the booking. The clients should spin in a while loop trying to book a seat until they either are lucky to get one or all tickets are sold.

5.3 The main operation using pseudo code may be described as follows:

SeatManager:

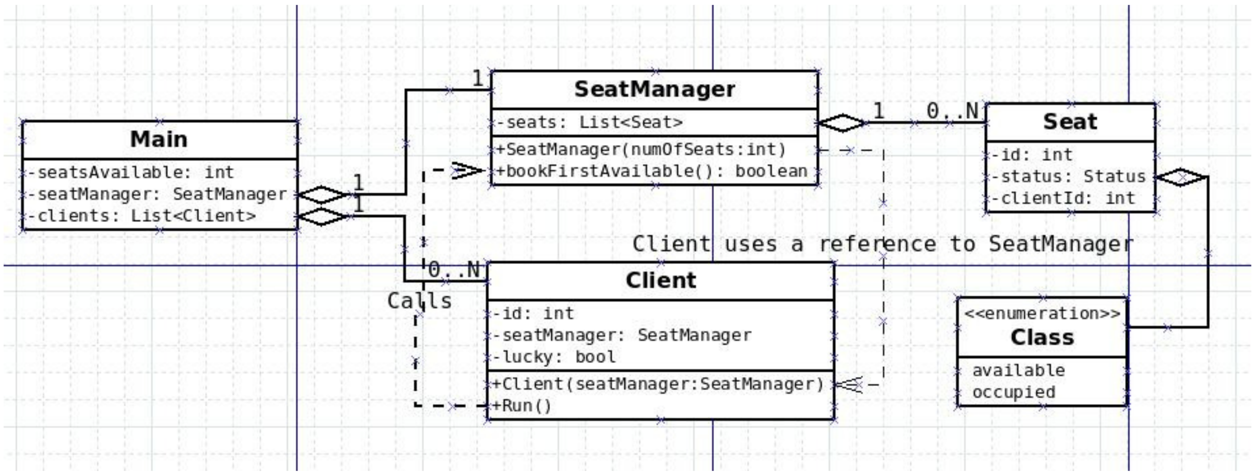
```
public bool bookFirstAvailable(){
seat = getFirstAvailable();

if (seat.isAvailable){
        seat.book(client);
        bookedSeats++;
      }
}
```

Client thread:

```
bool lucky = false;
while(manager.seatIsAvailable() and not lucky){
        lucky = manager.bookFirstAvailable();
}
```

5.4 In addition to a controller or any other start class, you may need to create a **Seat** class to represent a seat, a **Client** class to represent a client, and a **SeatManager** class to contain and manage a list of the Seat objects. The class diagram below illustrates the classes and their instance variables.



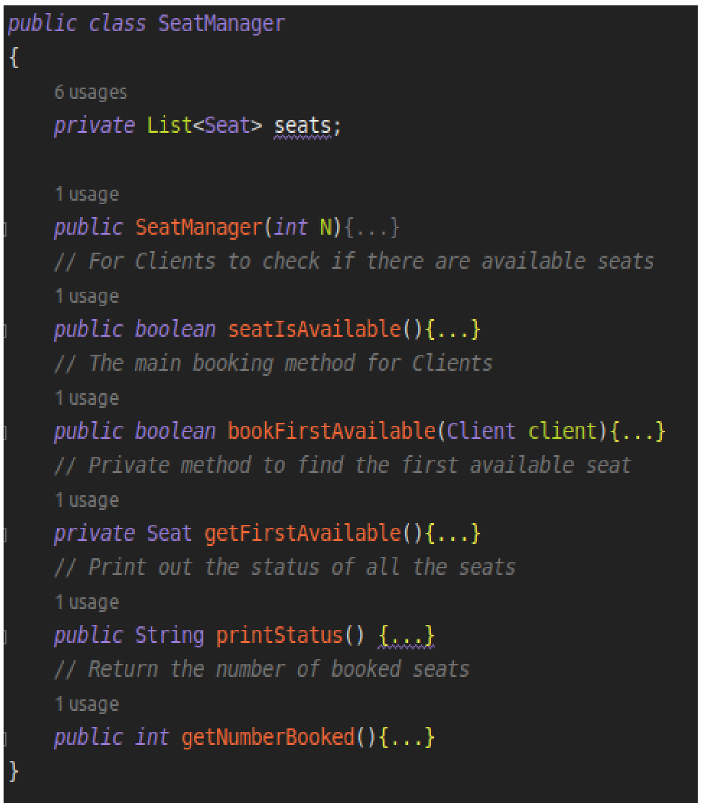
5.5 In the **Main** class, instantiate a **SeatManager** object and create a list of **Client** objects, which can also be referred to as customers. You can create a separate class, such as **ClientManager**, to handle the management of clients. If you don't create such a class, you can create a list of clients in the Main class.

5.6 Use two constants, e.g., **seatsAvailable**, **currNumOfClients**, for setting the number of seats available and the current number of clients, respectively. You can set the constant, **seatsAvailable**, to 10, and **currNumOfClients** to 100, although you may test with other values as well. Using a loop, create the clients and assign each one a unique **id**, which can be the same as the value of the counter variable.

5.7 To handle multiple clients efficiently, create one thread for each client. This will ensure that multiple clients can access the **SeatManager** simultaneously with (no ME) or without (ME) interfering with each other.

5.8 As an additional aid, a suggested structure for the **SeatManager** is provided in the image at the right side. The intension is only to give you an idea of the expected functionalities from the manager class. If you encounter any confusion with any of the methods, you can come up with your own solutions and ideas.

5.9 It is important to keep track of the number of seats that are booked in the **SeatManager** and the number of clients who have been successful in booking a seat. By doing so, you can monitor the performance of the system and easily find out when errors happened to make necessary improvements. A sample output from running a possible solution may look like the image on the right-hand side.

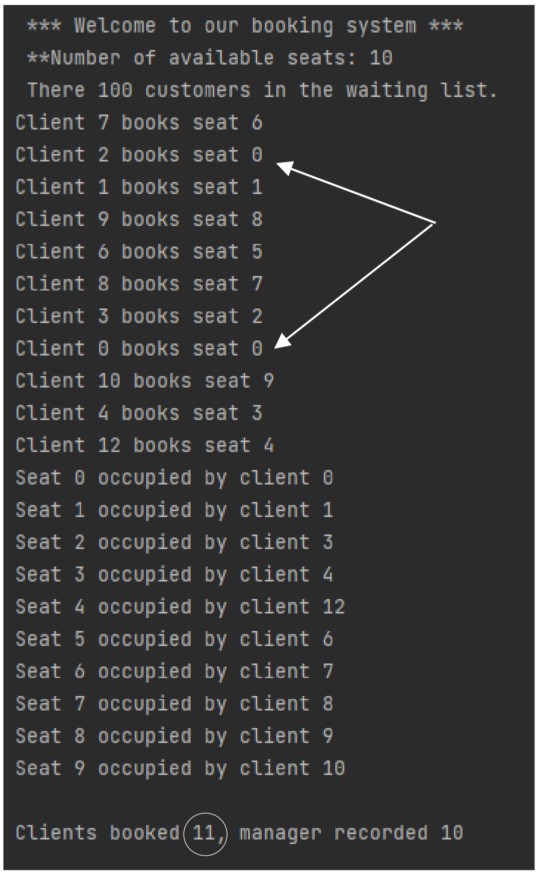


```java
public class SeatManager
{
    6 usages
    private List<Seat> seats;

    1 usage
    public SeatManager(int N){...}
    // For Clients to check if there are available seats
    1 usage
    public boolean seatIsAvailable(){...}
    // The main booking method for Clients
    1 usage
    public boolean bookFirstAvailable(Client client){...}
    // Private method to find the first available seat
    1 usage
    private Seat getFirstAvailable(){...}
    // Print out the status of all the seats
    1 usage
    public String printStatus() {...}
    // Return the number of booked seats
    1 usage
    public int getNumberBooked(){...}
}
```

Note that it can be challenging to detect an error caused by a race condition in the multithreading implementation. Our tests, which involved 10 seats and 100 threads, resulted in an error approximately once every 50 runs. The image on the right-hand side shows an example of running the application in which Seat 0 is booked twice by two different threads.

However, if you insert a small sleep-time in your code when booking a seat, you will see the impact of race condition very quickly.



```
 *** Welcome to our booking system ***
 **Number of available seats: 10
 There 100 customers in the waiting list.
Client 7 books seat 6
Client 2 books seat 0
Client 1 books seat 1
Client 9 books seat 8
Client 6 books seat 5
Client 8 books seat 7
Client 3 books seat 2
Client 0 books seat 0
Client 10 books seat 9
Client 4 books seat 3
Client 12 books seat 4
Seat 0 occupied by client 0
Seat 1 occupied by client 1
Seat 2 occupied by client 3
Seat 3 occupied by client 4
Seat 4 occupied by client 12
Seat 5 occupied by client 6
Seat 6 occupied by client 7
Seat 7 occupied by client 8
Seat 8 occupied by client 9
Seat 9 occupied by client 10

Clients booked 11, manager recorded 10
```

5.10 To ensure the correctness of your implementation, run your code multiple times and watch for cases in which a race condition causes erroneous output. If you manage to obtain such output, take a screenshot of the output, and include it in your submission. However, if you are not able to reproduce the error despite multiple attempts, you may choose to give up.

5.11 Regardless of whether you encounter an error or not, it is important to identify the critical sections of the code where race conditions may occur. To prevent the occurrence of errors due to race conditions, you should implement mutual exclusion mechanisms such as locks or semaphores, and monitors. However, in this assignment we use locks. With proper mutual exclusion in place, race conditions should never happen.

6 ANTICIPATED QUESTIONS

Consider the following questions. You are not required to provide written answers, and there is no correct answer for any of them. Instead, be prepared to discuss and present your thoughts and reasoning. These questions are relevant to both parts of the assignment:

Q1: Why do errors occur less frequently in the ticket booking application (Part 2) than in the bank account application (Part 1)?

Q2: Where in your code does race conditions occur, and why?

Q3: Is it possible to ensure correct execution of the code in both cases without using locks?

Q4: What are the drawbacks of using locks for mutual exclusion?

7 PART 3 (**NOT** MANDATORY) - DEADLOCK IN THE TICKET RESERVATION

This part is optional and its aim is to practice studying occurrence of deadlock.To check if your code is prone to deadlock, you can allow clients to buy multiple seats, each seat with its own lock. A client could hold a lock to one seat while waiting for the lock to another seat to be released, leading to a potential deadlock. You can simulate this situation using the random object to select one or two clients attempting to purchase two or more seats.

Questions to consider for this part (no written answers required):

Q1: Have you encountered a deadlock situation in your code? If so, where in the code did it occur?

Q2: Does Mutual Exclusion (ME) play a role in this scenario?

Q3: What solution can you suggest for preventing deadlocks from occurring?

8 GRADING AND SUBMISSION

You are required to present your solution to your supervisor during the lab sessions in person. Additionally, you must upload the solution to Canvas. To upload your files, compress all the folders, and subfolders into a single archive file such as **zip**, **rar**, or **7z**. Upload the archive file via the Assignment page in Canvas.

Good Luck!

Farid Naisan,
Course Responsible and Instructor