Synchronization & it's Need (vvvvI xxxxxx)

(1) Definition:

- **Synchronization:** Threads share the same memory space i.e. they can also share objects

(b) There might arise some situations (critical) where it's desirable that only one thread at a time can access to a shared resources.

(Let's deep dive to it by an example)

- If there is a ticket booking applications and there's a field called remaining seats.
- There are multiple threads here responsible to book remaining seats for multiple users.
- If two (or multiple threads) try to book the remaining seats simultaneously.
- Assume, `Remaining seat > 0`
- When 1st thread books the seat and perform some other operations before updating seats.
- At the same time other threads also come into the actions and book the seat as `Seat Visibility | Remaining seat > 0` as not updated yet.

This is called race condition not recommended.

understand race condition practically (Hands-on) and how to prevent it.

Package: Synchronization
File: Stack.java

STACK → LIFO (Last In, First Out)

Implementing Stack using an Array with threads | Synchronization Demo with Stacks (Synchronized methods & synchronized Blocks)

```
Package Synchronization;

public class Stack {
    //1) Attributes
    private int[] Array; //Stack Storage
    private int
```

- When 1st [illegible] other operations before updating seats.
- At the same time other threads also come into the actions and book the seat as seat visibility | remaining seat > 0 as not updated yet.

This is called race condition not recommended.

understand race condition practically (Hands-on) and how to prevent it.

Package: Synchronization
File: Stack.java

STACK → LIFO (Last In, First Out)

Implementing Stack using an Array with threads | Synchronization Demo with Stacks (Synchronized methods & Synchronized Blocks)

```
Package Synchronization;
public class Stack {
  //1) Attributes
private int[] array; //Stack Storage
private int stackTopIndex;

//2) Constructor with capacity of stack as an argument
Public Stack(int capacity) {
  array = new int[capacity]; //object for array
  stackTopIndex = -1; // No element in the stack initially
}

//3) methods()
  (//3.1) check if stack is empty
  Public boolean isEmpty() {
    return stackTopIndex < 0;
  }
  //3.2) check if stack is full
    public boolean isFull() {
    return stackTopIndex >= array.length - 1;
    }
```

Package → Synchronization
File → Stackk.java

// 3.3) PUSH() → Insert element into STACK

```
public boolean push(int element){
    if(isFull()){
     return false; // STACK is full, cannot insert
    }
    ++ stackTopIndex; // increment stack top index.

    try {
Thread.sleep(1000); // simulate delay (optional)
                    make thread wait (1 sec = 1000ms)
}Catch (Exception e) {
System.out.println("Thread interrupted: " + e.getMessage());
}
array[stackTopIndex] = element; // Insert element into stack
return true; // Push successful
}
}
```

// 3.4) POP() → Delete element from STACK

```
public int pop() {
    if(isEmpty()){
      returns Integer.MIN_VALUE; // stack underflow case
    }
    int obj = array[stackTopIndex]; // store the top element

    try {
```

```
}
}

//3.4) POP() → Delete element from STACK

public int pop() {
  if(isEmpty()){
   return Integer.MIN_VALUE; //Stack underflow case
  }
  int obj = array[stackTopIndex]; //store the top element

  try{
  Thread.sleep(1000); //Stimulate delay (optional)
                      make thread wait (1sec=1000ms)
  } catch (Exception e){
  System.out.println("Thread interrupted: " + e.getMessage());
  }
  stackTopIndex--; //correctly decrement stack pointer
  return obj; //Return popped element
  }
}

//Create class ThreadTester
class ThreadTester{
  public static void main(String[] args){
  System.out.println("main is starting");
  //Object for class Stack
Stack stack = new Stack(capacity: 5);
new Thread(()->{
  int counter=0;
  while(++ counter<10)
  System.out.println("Pushed: " + stack.push(element: 1000 ));
  }, name: "pusher").start();

new Thread(()->{
  int counter=0;
  while(++ counter<10)
   System.out.println("Popped: " + stack.pop());
  }, name: "popper").start();
  System.out.println("Main is exiting");

  }
}
```

Output

Output

Synchronization.ThreadTester (Package.Class)
Main is starting
Main is exiting
Exceptin in thread "pusher" java.lang.ArrayIndexOutOfBoundException
Popped: 0
Popped: -2147483648
Popped: -2147483648

Topic → **Synchronized Block**

[Package: Synchronization.SynchronizedBlock
File: StackK.java

→ In order to ensure one thread at a time to access a particular resource / particular piece of code we need to use synchronized keyword (synchronized block with lock).

→ There are 2 ways to make code block synchronized.
(a) Apply synchronized on the method itself.
(b) To make particular piece of code synchronized within the method()
(c) Use synchronized block with explicit lock.

```
Synchronized {
// holds critical code
// Allows just one thread to access a particular block of code at a time so called synchronized block.
}
```

→ Now we made just access to one thread out of many within the task to get executed by performing it's task.

→ Now Question arises how will I know which

to access a particular at a time so called synchronized block.

→ Now we made just access to one thread out of many within the task to get executed by performing it's task.

→ Now question arises how will I ensure which thread would be able to access this particular synchronized block.

- So thread need to acquire lock.
- Whichever thread will have access to the lock it will be able to enter this critical section (inside synchronized block) & execute that piece of code.
- In java every object can use lock (All wrapper class can use it easily).
- Object/Instance of wrapper can use it.

Synchronized (lock) {

Pass lock object to the synchronized block

}

- In java we can use any objects as a lock.

Questions VVVI

- If a multiple threads (t_1, t_2, t_3) trying to gain access to push() to execute it.
- If multiple threads (t_1, t_4, t_5) trying to gain access to pop() to execute it.

Package:- Synchronization · SynchronizationBlock · multipleThreads Access same time
File:→ StackK.java

Answer

(a) Now thread(t_1) trying to gain access to push() method with explicit lock to get access to it.

(b) and at the same time thread (t_1) tries to gets access to pop() with lock as well.

(c) But if thread (t_2) gets a lock and gets access to push() and perform it's execution in the meantime thread (t_1) is waiting for lock.

(d) But thread (t_1) can't go and execute pop() because pop() & push() are bounded by the same lock object, since this lock object can only be with one thread at a time.

(e) Therefore, both this synchronized methods (push() & pop()) are bounded by the same lock objects.

(f) Therefore, whever threads gets access to the explicit lock he/she will only be able to access any of this methods() other threads need to wait.

(g) These two methods() [push() & pop()] might be completely different from each other (Based on their functions) but since both are bounded by same lock therfore

Crisp & clear answer for Interview

If a particular synchronized method has a lock and that lock also restricts access to other synchronized methods as well all the access to synchronized methods() will be blocked for all other threads that doesn't have that particular lock.

(Ques) If there are 2 separate lock 1 → for push() other for pop() then ?

Package → Synchronization . SynchronizedBlock . SeparateLocks
File → StackK.java

Answer

(a) If we assign 2 different explict lock for two different methods(). i.e. push() & pop().

(b) Ideally we allow two threads to simultaneously run the push() & pop() (at the same time).

(c) This will not solve problem for push() & pop() → race condition arises.

(d) So it's strongly recommended, for push() and pop() in the stack ~~we need~~ to have same explict lock so that we if push() getting executed other threads won't be able to call pop() that is fine but it is also

(d) so it's stongly recommended to have same explicit lock for both the methods push() and pop() so that if one method() getting executed, other threads can't Call pop() the methods(other method) to avoid race conditions (as both methods() are bounded by the same lock)

Ex

Synchronized Block ends here..!!

Define race condition (VVVI) xxxxx

Race condition occurs when two (or) more threads simultaneously update the same value and as a consequence, leave the value in an undefined (or) in consistent state.

(Q) How can we make entire method() synchronized?

→ By applying synchronized on the method() itself

[Package: Synchronization.SynchronizedMethod
File: StackK.java]

(Q2) When making method() synchronized what is the lock being used?

Define race condition (WWW I) xxxxx

Race condition occurs when two (or) more threads simultaneously update the same value and as a consequence, leave the value in an undefined (or) in consistent state.

(Q) How can we make entire method() synchronized?

→ By applying synchronized on the method() itself

[Package: Synchronization.SynchronizedMethod
File: StackK.java]

(Q2) When making method() synchronized what is the lock being used?

→ Behind the scenes / At background the entire piece of code is wrapped using synchronized (this) keyword.

→ Compiler uses instance of the current object as the lock i.e. this keyword is a lock.

→ for non-static synchronized method this keyword is used as lock.

Topic → Static Synchronized methods ()

Package → Synchronization · StaticSynchronizedmethods
File → StackK.java · StaticSynchronizedmethods

Question

How to synchronized static methods ?

→ we synchronized static methods on the class lock itself.

→ In static synchronized methods ()

StackK.class ——— is used as lock

(Class_name.class)

Topic → Thread safety (VVVVI)

Intervicw

Thread safety means that a program (or) code segment can be safely accessed by multiple threads without leading to race conditions, data corruption (or) unexpected behavior.

Eg → StringBuffer