

JAVA @17

Java Thread – Part III

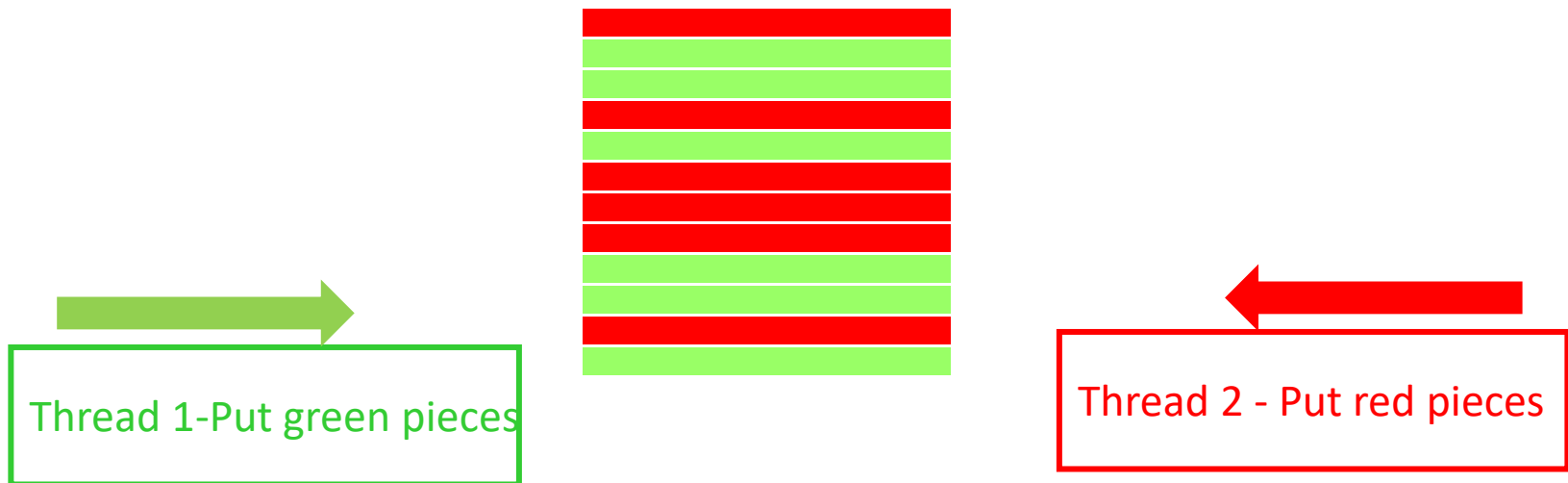
Objective

After completing this session, you will be able to understand,

- What is synchronization?
- Various thread method wait, notify and notifyAll

What is race condition?

Scenario: Consider a scenario in which you want to keep green blocks sequentially at the bottom and red blocks be placed on top of the green blocks. You have developed a program in such a way that two threads one for placing the green blocks and other for red blocks. Lets see how the program places the blocks.



You couldn't get your expected output. The red blocks and greens blocks were placed randomly. This is what we call a **Race Condition**

When does race condition occur?

- A race condition occurs when two or more threads are able to access shared data and they try to change it at the same time.
- Both of them try to modify the data at the same time that is both are racing to get access/change the data
- Since the thread scheduling is system dependent we cannot predict a reliable behavior of the application.

So what can be done to make only one Thread access a shared resource at a time ?

The solution is **synchronization.**

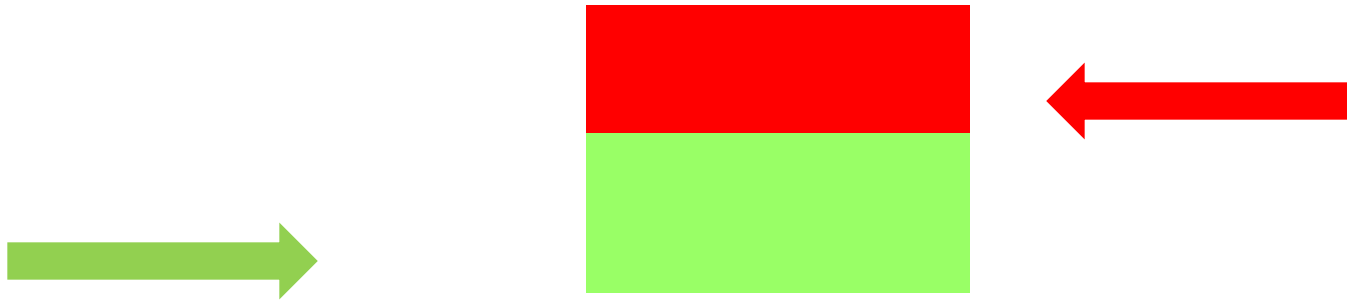
What is synchronization?

Synchronizing threads means access to shared data (or) method execution logic in a multithreaded application is controlled in such a way that only one thread can access the method/shared data at a time. The other threads will wait till the thread releases the control. The shared data is considered to be ***Thread Safe***.

Java uses the concept of ***monitors*** to implement synchronization.

Problem solved using Synchronization?

Lets see how synchronization can solve our problem in placing the colored blocks



Here the red thread waits till the green thread completes the job.
This is achieved using **Synchronization**.

Thread Monitor

- A **monitor** is like a room of a building that can be occupied by only one thread at a time.
- The room contains a data or logic.
- From the time a thread enters this room to the time it leaves, it has exclusive access to any data/logic in the room.
 - This is done by gaining control on the monitor which is nothing but the room.
- Entering the special room inside the building is called "**acquiring the monitor**".
- Occupying the room is called "**owning the monitor**" or exclusive access to the data.
- Leaving the room is called "**releasing the monitor**".

Synchronization methods

Synchronization is achieved by using one of the two methods,

Method 1: Synchronized blocks

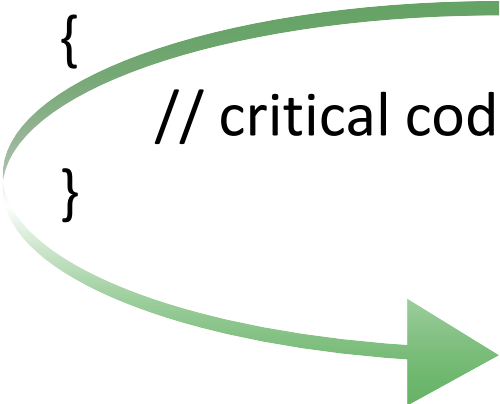
Method 2: Synchronized methods

Method 1: synchronized method

- ***Synchronized*** methods can be created by using the keyword ***synchronized*** for defining the method.
- The code which should be thread safe is written inside this method.
- During the execution of a synchronized method, the thread holds the monitor of that method's object (or) if the method is static it holds the monitor of that method's class.
- If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Synchronizing a Method

```
public synchronized void updateRecord()  
{  
    // critical code goes here ...  
}
```



Only one thread will be inside the body of the `updateRecord()` method. The second call will be blocked until the first call returns or `wait()` is called inside the synchronized method.

Synchronizing an Object

```
public void updateRecord()
{
    synchronized (this)
    {
        // critical code goes here ...
    }
}
```

Example – Synchronize method

Demonstration: In this demo we will see how to implement synchronization and how it works?. We will create a small application which prints a text message on the console.

This demo will also help you to understand the behavior of synchronized and unsynchronized methods.

Example – Synchronize method

We will develop a program to spawn three threads, each thread should accept two string arguments and print the String message.

The following two messages should be printed by each thread,

Thread 1 – “Hello..” and “There”

Thread 2 – “How” and “are you”

Thread 3 – “Thank you,” and “very much”

Output Expected

Hello..There

How are you

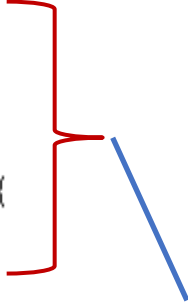
Thank you,very much

Classes

1. **PrinterThread** : Thread class used for printing the String.
2. **StringPrinter** : This class is used by the Thread class to print the Strings, contains a method which prints the strings.
3. **SyncExMain** : Main method to initiate the threads.

Solution – Synchronize method

```
public class StringPrinter {  
    static void printStrings(String stringA, String stringB) {  
        System.out.print(stringA);  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
        System.out.println(stringB);  
    }  
}
```



Methods to print the strings.

NOTE:

- The strings are printed one after the other with a pause. This has been implemented using the sleep method.
- The first message should be printed without a new line.
- Second message should be printed with a new line.


Solution – Synchronize method

```
public class PrinterThread implements Runnable {  
    Thread t;  
    String stringA;  
    String stringB;  
  
    public PrinterThread(String stringA, String stringB) {  
        this.stringA = stringA;  
        this.stringB = stringB;  
        Thread t = new Thread(this);  
        t.start();  
    }  
  
    public void run() {  
        StringPrinter.printStrings(stringA, stringB);  
    }  
}
```

Creates a thread object on “this” Runnable object and starts the thread.

Solution – Synchronize method

```
public class SyncExMain {  
    public static void main(String args[]) {  
        new PrinterThread("Hello..", "There");  
        new PrinterThread("How", " are you");  
        new PrinterThread("Thank you,", "very much");  
    }  
}
```



Create three threads and pass the appropriate String messages.

Output – without sync method

Execute the class, the output could be displayed as below.

```
Thank you, HowHello..very much  
are you  
There
```

This is the output we get, not the expected one. Note the order in which the messages are printed could be different depending upon the OS scheduler.

Inference:

Each thread overruns the other thread resulting in the message being jumbled.

add - synchronization

Now lets control the threads by using *synchronized* key word.

Make the *printStrings()* method of the *StringPrinter* class *synchronized*.

```
public class StringPrinter {  
    static synchronized void printStrings(String stringA, String stringB) {  
        System.out.print(stringA);  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
        System.out.println(stringB);  
    }  
}
```

Add the *synchronized* keyword in method declaration.

Output – with sync method

Execute the class, the output could be displayed as below.

Hello..There

How are you

Thank you,very much

The output is as expected.

Inference:

The *synchronized* keyword has ensured that the thread executes in order. Only after a thread completes printing both the messages the other thread starts the printing.

Method 2: sync statements

Where is Synchronized statement used?

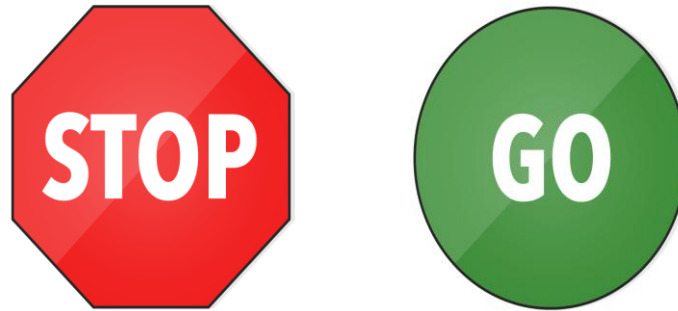
If only a small portion of code in a method needs to be Thread safe we can mark that portion only as synchronized rather than marking the entire method as ***synchronized***.

Syntax:

```
public class SyncMethods{  
    public void methodA() {  
        synchronized(this){  
            //This block contains a set of  
            //statements which needs to be synchronized  
        }  
    }  
}
```

The syntax is **synchronized(this)** where **this** denotes the current object meaning the thread should acquire a lock on the current object to execute the statements.

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is the need of synchronization in multi threaded programming?
- What is a race condition?
- How can Synchronization be implemented?

Inter Thread Communication


In many instance we end up in a situation where we need one thread to communicate with other threads in a process. This is referred to as “***Inter Thread Communication***”

Inter Thread communication is achieved using one or more of the below methods,

1.wait()

2.notify()

3.notifyAll()



Occasionally used in
application
development.

wait()

- ***wait()*** method causes a thread to release the lock it is holding on an object allowing another thread to run.
- ***wait()*** can only be invoked within a synchronized code.
- It should always be wrapped in a try block and handle ***IOException***.
- ***wait()*** can only be invoked by the thread that owns the lock on the object using **synchronized(this)**.

How wait method works?

- When ***wait()*** is invoked, the thread becomes dormant until one of the below four things occur,
 - Another thread invokes the ***notify()*** method for this object and the scheduler arbitrarily chooses to run the thread.
 - Another thread invokes the ***notifyAll()*** method for this object.
 - Another thread interrupts this thread.
 - The specified ***wait()*** time elapses.
- When one of the above occurs, the thread becomes re-available to the Thread scheduler and competes for a lock on the object.
- Once it regains the lock on the object, everything resumes as if no suspension had occurred.

notify()

- **notify()** method wakes up a single thread that is waiting on the monitor of this object
- If many threads are waiting on this object, then one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the JVM implementation.
- **notify()** method can only be used within synchronized code.
- The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.

notifyAll()

- Similar to notify method **notifyAll()** is also used to wake up threads that wait on an object
- The difference is **notify()** awakens only one thread whereas **notifyAll()** awakens all the threads

Deadlocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Example to understand dead locks:

Tim & Ron are two good friends they belong to a country where they have a weird culture, whenever they meet each other one has to bow and wish the other “*Good Day*” he has to remain bowed till the other person bows back and greets back “*Thank you*”

Lets see what happened?

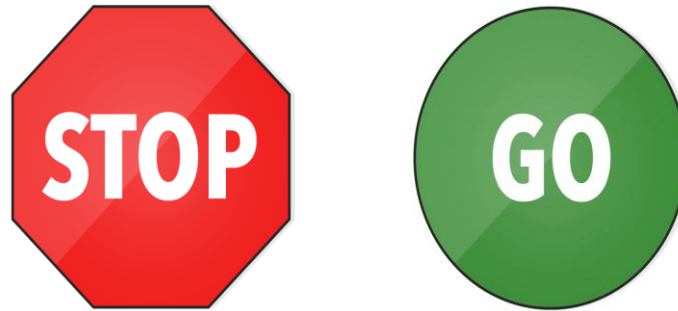
Example – Inter thread communication

```
class Customer{
    int balance = 10000;
    synchronized void withdraw(int amount) {
        System.out.println("I am going to withdraw amount...");
        if(this.balance<amount) {
            System.out.println("Less balance; wait for some time.");
            try {
                wait();
            }catch(Exception e) {}
        }
        this.balance-=amount;
        System.out.println("Amount Received. Thank you.!");
    }
    synchronized void deposit(int amount) {
        System.out.println("Going to Deposit...");
        this.balance+=amount;
        System.out.println("Deposit Completed..");
        notify();
    }
}
```

Example – Inter thread communication

```
public class InterThreadDemo {  
    public static void main(String[] args) {  
        final Customer cust = new Customer();  
        new Thread() {  
            public void run() {  
                cust.withDraw(9000);  
            }  
        }.start();  
        new Thread() {  
            public void run() {  
                cust.deposit(10000);  
            }  
        }.start();  
    }  
}
```

Time To Reflect



Trainees to reflect the following topics before proceeding.

- How can we make a thread wait on an object it holds the lock?
- What is the difference between notify and notify all?
- What is a dead lock?

Thank you

You have successfully completed
Thread-III