Java Concurrency Interview Questions

Lokesh Gupta

May 3, 2023

Interview Questions

Concurrency, Interview Questions

Java concurrency has evolved a lot from introducing *Thread* and *Runnable* to releasing virtual threads in Java 21. Although the low-level and inner workings of concurrency constructs (such as Lock and synchronization) are abstracted by the higher-level classes (such as BlockingQueue and ThreadPool), it is still expected to know how things work under the hood.

The following questions and answers will help you to better prepare for your next Java interview. In most parts of this article, threads will refer to platform threads. Whenever needed, we will separately refer to virtual threads.

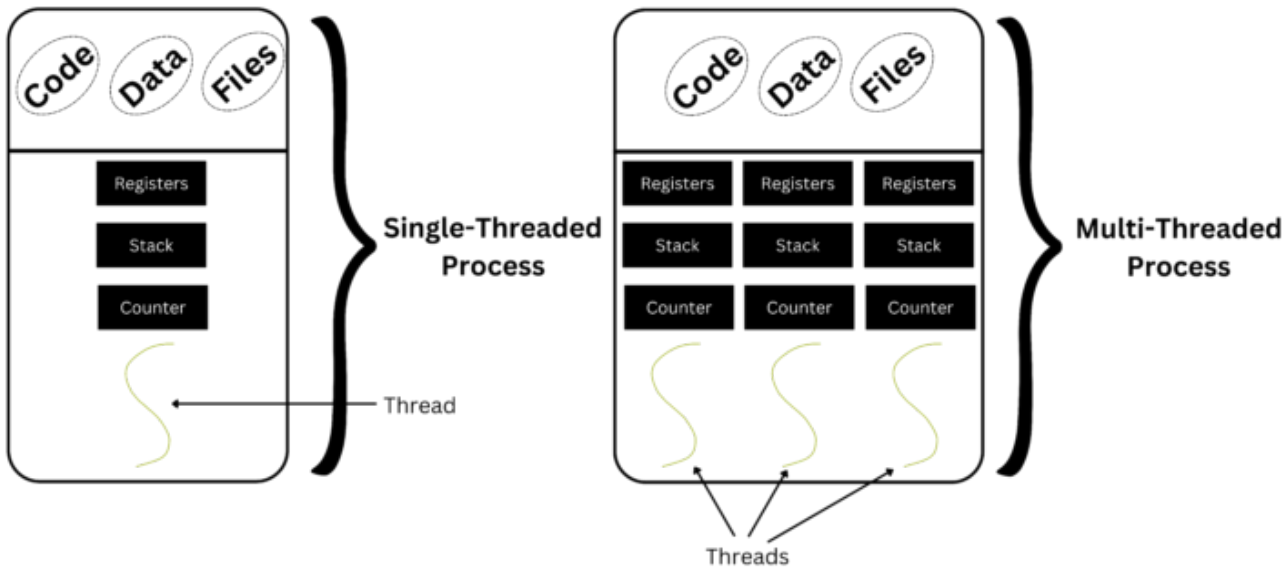
1. Basic Concepts

1.1. What is the difference between Process and Thread? What is Context Switching?

When we execute a Java program, the OS creates a process and allocates the necessary memory and resources. The JVM is responsible for managing it, thereafter, for memory allocations and I/O operations. Internally, JVM creates multiple threads to execute different parts of the program concurrently.

actions specified in the program. In contrast, **a thread is a subset of the process**, also known as the lightweight process.

- The OS manages the processes, and they exist in their own memory space. A process has its own data that is not shared with any other processes. Whereas the thread scheduler manages a thread, the thread always shares data, processor, and memory space with peer threads.
- Proper synchronization is required in the case of threads, but in the case of process, there is no need for synchronization as they work in an isolated manner.
- The process takes more time in creation, context switching, and termination compared to threads.



Context switching is a technique that includes saving the current state of a running thread, restoring it later when it has to be executed, and giving a chance to other waiting threads to execute. Context switching is necessary to execute multiple threads concurrently, although the time taken in switching impacts the application's performance. To optimize the performance, various techniques are used such as thread prioritization, preemption, and CPU caching.

We can create a *Thread* in the following ways:

- By extending the *Thread* class
- By implementing the *Runnable* interface
- Creating virtual thread using *Thread.startVirtualThread()* or *Thread.ofVirtual()*

To run the thread, we call the *Thread* class *start()* method, which is responsible for registering the thread with the thread scheduler and putting it in *RUNNABLE* state. The executable code is written in the *run()* method.

```
public class SubTask extends Thread {

  public void run() {
    System.out.println("SubTask started...");
  }
}

//Start the thread

Thread subTask = new SubTask();
subTask.start();
```

When using *Runnable* interface, we put the executable code in the overridden *run()* method and pass the *Runnable* instance to *Thread* constructor before starting it.

```
class SubTaskWithRunnable implements Runnable {

  public void run() {
    System.out.println("SubTaskWithRunnable started...");
  }
}
```

```
subTaskWithRunnable.start();
```

Virtual threads are a relatively new feature and are entirely handled by JVM, including context switching. The *Thread.startVirtualThread()* creates a new virtual thread to execute a given *Runnable* task and schedules it to execute.

```
Runnable runnable = () -> System.out.println("Inside Runnable");
Thread.startVirtualThread(runnable);
```

1.3. How can you define the stack size explicitly for a Thread?

The *Thread* class defines the following constructor through which we can determine the stack size explicitly for the new thread.

The effect of the `stackSize` parameter, if any, is highly platform dependent. In some platforms, a higher value of stack size may allow a deeper recursion depth before throwing `StackOverflowError`, and in other platforms, it may not have any effect at all.

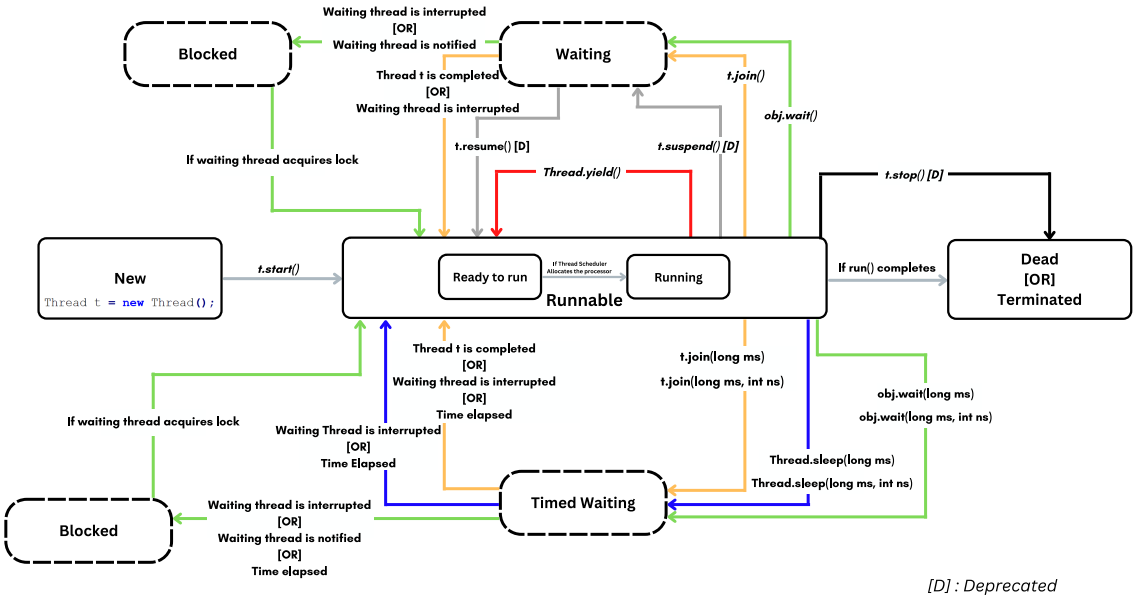
```
new Thread(ThreadGroup group, Runnable target, String name, long stackSize);
```

1.4. Explain various states of the Thread lifecycle. Explain the state transitions.

During its lifetime, a thread can be in the following states:

- NEW
- RUNNABLE (Ready or Running)
- WAITING

- BLOCKED
- DEAD or TERMINATED



- When you create an instance of the *Thread* class, the thread is in the *NEW* state.
- When you call the *start()* method on that thread class object, the thread is in the *RUNNABLE* (Ready to Run) state. In this state, the thread is waiting for the thread scheduler to assign the processor to it.
- When the thread acquired the processor from thread-scheduler it starts executing the *run()* method, and it's in the *RUNNABLE* (running) state.
- On the running thread, if you call *sleep(time)*, *join(time)* or *wait(time)*, the thread enters into the *TIMED WAITING* state, and when the corresponding event takes place, then the thread returns back to the *RUNNABLE* (Ready to Run) state.
- On the running thread, if you call *join()*, *wait()* or *suspend()*, the thread enters into the *WAITING* state, and when the corresponding event takes place then the thread

- While calling *notify()* or *notifyAll()* method thread does not immediately release the lock that's why the waiting thread that got the notification has to wait for the running thread to release the lock of the object. At this stage, the thread is in the *BLOCKED* state. Once the thread acquires the lock then it moves to the *RUNNABLE* (Ready to Run) state.
- If the *run()* method completes or we call the *stop()* method on the running thread, the thread enters into the *DEAD* or *TERMINATED* state.

> See Also: Java Thread Life Cycle and States

1.5. Can we override *Thread.start()*?

Yes, we can override the *Thread.start()* method, but the overridden method will be executed just like a normal method call executed by the *main* thread only. It will not spawn a new *Thread* similar to the default *start()* method.

```
class ChildThread extends Thread {

  public void start() {
    System.out.println("Overriding start() method in ChildThread");
  }

  public void run(){
    System.out.println("ChildThread run() method");
  }
}
```

Create a new *ChildThread* instance and call its start() method. It does not start a new thread and *run()* method is not executed.

```
childThread.start();
```

Output:

```
Overriding start() method in ChildThread
```

To initialize a new *Thread*, we must call the *super.start()* method from the child thread; only then the *run()* method will be executed.

```
public void start() {
  super.start();  //Calling Thread.start()
  System.out.println("Overriding start() method in ChildThread");
}
```

Output:

```
Overriding start() method in ChildThread
ChildThread run() method
```

1.6. What if we don't override the *run()* method?

If we don't override the *run()* method in the extended class then the default *Thread.run()* method will be executed which has an empty body. Hence a new *Thread* will be created and executed which does nothing.

```
class SubTask extends Thread { }
```

```
subTask.start();

// New thread will be created and no output will be printed to the console.
```

1.7. What is thread priority? What is the default thread priority?

Thread priority is an integer number between 1 to 10 assigned to each thread in the JVM. Thread priority is used by the thread scheduler at the time of thread execution. **While allocating the processor to the thread, the thread scheduler gives the first chance to the thread with the highest priority**.

The default priority for any thread is inherited from the parent thread. As the default priority for the *main* thread is 5, so all child threads will have the **default priority of 5**.

```
System.out.println(Thread.currentThread().getPriority());   //5

Thread.currentThread().setPriority(10);

//Create a new child thread
MyThread thread = new MyThread();
thread.start();

//Child thread has priority of parent thread
System.out.println(thread.getPriority()); //10
```

The *Thread* class defines 3 constants to represent the min, max, and default priority of any thread.

```
Thread.MIN_PRIORITY = 1;
Thread.NORM_PRIORITY = 5;
Thread.MAX_PRIORITY = 10;
```

thread?

The **user threads are created by the application** and are responsible for performing business tasks. For example, tasks submitted to *ExecutorService* are executed with user threads. Or the threads in a thread pool are user threads.

The daemon threads**, generally, are created by the JVM** with low priority than user threads and are executed in the background to support the user threads for smooth execution of the program. For example, garbage collection is a daemon thread started by JVM. Similarly, the finalizers are typically performed in daemon threads.

Note that daemon nature inherits from parent-to-child threads. The *main()* method is always a non-daemon thread so any other thread created from it will remain non-daemon until explicitly made daemon by calling `setDaemon(true)` method.

```
Task thread1 = new Task("thread1");
thread1.setDaemon(true);
thread1.start();
```

Also, we cannot convert a user thread to a daemon thread after it has started.

```
Task thread = new Task("user-thread");
thread.start();

thread.setDaemon(true);           // java.lang.IllegalThreadStateException
```

We can check the daemon nature of a thread using the *Thread* class method *isDaemon()*.

```
System.out.println(Thread.currentThread().isDaemon());    //false
```

all user threads are completed.

1.9. What is *ThreadGroup* and its purpose? What is the default *ThreadGroup*?

The *java.lang.ThreadGroup* class represents a group of threads and their subgroups. The purpose of *ThreadGroup* is to perform common activities for all threads present inside that thread group, like interrupting all threads, suspending all threads, setting max priority, etc.

Each thread in Java always belongs to a thread group. We can define the thread group of a new thread explicitly using the *Thread* class constructor.

```
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, String name)
...
```

If we don't define thread group explicitly, then the default thread group is inherited from the parent thread's thread group.

The *'main'* acts as the root thread group for all Java threads and thread groups.

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

  System.out.println(Thread.currentThread().getThreadGroup().getName());  //main
}
```

1.10. What is fairness policy?

In concurrency, a fairness policy means that different threads can *advance* whatever they are doing. In a 100% fairness policy, all threads should advance their work in almost equal

other threads never (or rarely) make any progress.

Due to an unfair waiting policy, a thread can be waiting infinitely to acquire a lock and face starvation. To avoid this starvation problem, the lock must follow a fair waiting policy which says that the longest waiting thread must get the higher priority and should acquire the lock first, which in-directly means that threads must acquire the lock in the order they requested it.

To provide fairness, *java.lang.ReentrantLock* class defines a constructor with a *boolean* parameter named *fair*. By default, the fairness policy is set to *false* which means the lock does not guarantee any particular access order. When set *true*, under contention, locks favor granting access to the longest-waiting thread.

```
ReentrantLock(boolean fair)
```

Note, however, that the fairness of locks does not guarantee the fairness of thread scheduling.

2. Thread Synchronization

2.1. What is a *synchronized* block, method, and statement?

In concurrency, it may often happen that multiple threads try to access and modify the same resource at the same time which ultimately produces erroneous results which is also known as **data-inconsistency**. To avoid data-inconsistency problems, some kind of **synchronization** is required, where only one thread can access and modify the resource at a given point of time.

Synchronization in Java guarantees that no two threads can execute a *synchronized* method, which requires the same lock, simultaneously or concurrently.

can call that method on the same object. Which requires a thread to get the object level lock to execute that method.

```
public synchronized int debitMoney(){...} //on the same object only one thread can cal
```

When *synchronized* is used for a *static* method, i.e. only one thread can call that method on any instance of a class. It requires a thread to get the class-level lock to execute that method.

```
public static synchronized int debitMoney(){...} //on any object only one thread can c
```

We can declare a block inside a method using the *synchronized* keyword to achieve synchronization for only some statements of the method present inside the synchronized block.

```
public synchronized int debitMoney(){
...
    synchronized(this){...} //achive synchronization for statments inside this block o
...
}
```

Any statements declared inside the synchronized area are known as **synchronized statements**.

2.2. What is the advantage of the 'synchronized block' over the 'synchronized method'?

entire method as *synchronized* reduces the performance of the method and thus the whole program, as it increases the waiting time of all threads accessing the method. At this time declaring those only statements inside the synchronized block is very effective.

```
public void bookTicket(){
    // ...

    synchronized(this){
        this.bookingRepository.book();
    }

    // ...
}
```

In the synchronized block we can define the object on which we require synchronization (on other class objects also) but in the case of the *synchronized* method, we can achieve synchronization only on the same class object only.

2.3. How is the *synchronized* keyword internally implemented?

The *synchronized* keyword is internally implemented using the concept of lock. Each and every class and object are having a **single unique lock** that can be acquired by a thread, to perform synchronous modifications.

If the thread not having the corresponding class or object lock then that thread can't call any *synchronized* methods or can't execute the *synchronized* block of that class. But no restriction for non-synchronized methods and blocks.

2.4. why can't we declare a *synchronized* method as *abstract*?

In general, synchronization requires a lock to be acquired before a method can be executed, and a lock is an implementation feature. Since an *abstract* method does not

Hence declaring the *synchronized* method as *abstract* makes no sense.

```
public abstract synchronized int debitMoney(); // CE: illegal combination of modifiers
```

3. Difference between

3.1. What is the difference between yield(), join() and sleep()?

yield()

The *yield()* method called on any thread pauses the currently executing thread as it's not doing any critical or important operations and gives the chance to other waiting threads with the same or higher priority. The *yield()* method is defined in *Thread* class as follows:

```
public static native void yield();
```

As it's the static method we can call the *yield()* method directly using the class name.

```
Thread.yield();
```

Calling the *yield()* method will move the thread from the RUNNABLE (Running) to the RUNNABLE (Ready to Run) state.

join()

called) to complete its execution. After completion of that thread, the waiting thread can continue or start its execution. The *join()* method is defined in *Thread* class as follows:

```
public final void join() throws InterruptedException;
public final synchronized void join(long millis) throws InterruptedException;
public final synchronized void join(long millis, int nanos) throws InterruptedExceptio
```

Calling the *join()* method will move the thread from the RUNNABLE (Running) to the WAITING state and calling the *join(int millis), join(int millis, int nanos)* method will move the thread from the RUNNABLE (Running) to the TIMED WAITING state.

sleep()

The ***sleep()*** method of the *Thread* class is used to pause the currently executing thread for a particular amount of time. After the specified time elapse the thread can continue its execution. The *sleep()* method is defined in *Thread* class as follows:

```
public static native void sleep(long millis) throws InterruptedException;
public static void sleep(long millis, int nanos) throws InterruptedException;
```

Calling the *sleep()* method will move the thread from the RUNNABLE (Running) to the TIMED WAITING state.

> See Also: Java Concurrency – Difference between yield() and join()

3.2. What is the difference between *notify()* and *notifyAll()*?

notify()	***notifyAll()***
Only one thread is notified among all the waiting threads.	All waiting threads will are notified.
The risk of infinite waiting and missing the execution of a thread is very high as there may be a chance that some thread does not get the notification.	The risk of infinite waiting and missing the execution of a thread is very low as each and every thread will get a notification.
CPU drain is very low hence performance is very high as only one waiting thread has to get the notification.	CPU drain is very high hence performance is very low as every waiting threads have to get the notification.
Only one waiting thread will be executed which got the notification.	Which thread will execute depends on the thread scheduler as every waiting thread got the notification.

> See Also: wait(), notify() and notifyAll() methods

3.3.What is the difference between *Callable* and *Runnable*?

Runnable	**Callable**
If a thread is not required to return some value after completing its execution then use *Runnable* interface	If a thread is required to return some value after completing its execution then use *Callable* interface
The *Runnable* interface is defined inside ***java.lang*** package since Java 1.0.	The *Callable* interface is defined inside ***java.util.concurrent*** package since Java

Runnable	Callable
	1.5.

The *Runnable* interface contains the *run()* method
 	
public interface Runnable {
public abstract void run();
}
	The *Callable* interface contains the *call()* method
	
	public interface Callable<V> {
	V call() throws Exception;
	}

We can pass the *Runnable* as a parameter to the *Thread* class constructor as *targetRunnable*	We can't pass the *Callable* as a parameter to the *Thread* class constructor
We can't call *invokeAll(), invokeAny()* methods of `ExecutorService` by passing the collection of *Runnable* as a parameter	We can call *invokeAll(), invokeAny()* methods of `ExecutorService` by passing the collection of *Callable* as a parameter

> See Also: Difference between Callable and Runnable

3.4. What is the difference between class-level lock and object-level lock?

The object-level lock is used when we want to synchronize the instance properties or critical sections of a class such that only one thread can execute *synchronized* methods and locks on the given instance.

We can achieve object-level locking using the *synchronized* keyword.

```
public synchronized void demoMethod(){...}

or

synchronized (this){...}
```

The class-level lock is used when we want to synchronize class-specific properties or critical sections (*static* members) such that only one thread can execute *synchronized* methods and blocks on any instance of that class in the runtime.

We can achieve object-level locking using the *synchronized* keyword with a *static* method.

```
public synchronized static void demoMethod(){...}

or

synchronized (MyClass.class){...}
```

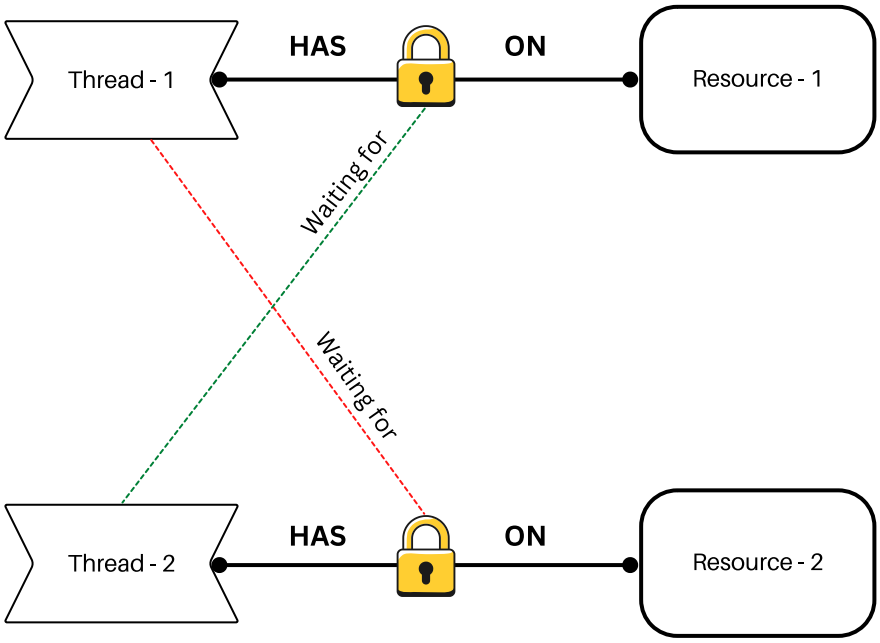
> Read More: Object level lock vs Class level lock

4. Deadlock, Livelock, and Race Condition

4.1. What is the deadlock and the main reason behind the deadlock?

A deadlock is a situation where two threads are holding the lock on some different resources, and both threads are waiting for each other to release the lock but no one is able to release to lock on the resources that it's holding, and both threads remain in the BLOCKED state and end up creating a circular dependency. Hence this infinite amount of waiting for each other to release the lock is nothing but a deadlock.

up first".



Thread synchronization is the main reason behind the deadlock situation. Improper use of *synchronized* methods and blocks can lead our program to a deadlock situation. Hence it's never recommended to use synchronization if there is no specific requirement.

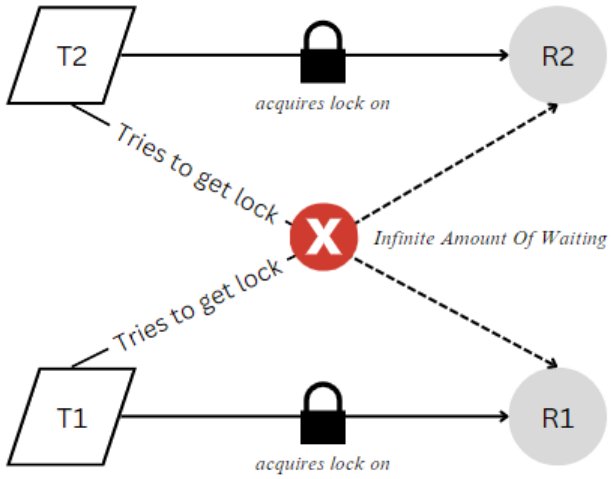
> See Also: How to Create a Deadlock and Solve in Java

4.2. What is the difference between Deadlock, Livelock, and Starvation?

Deadlock

A deadlock is a situation where two or more threads are waiting infinitely for each other to release the lock on the resource that they are holding.

Now thread **T1 needs** a lock on resource **R2**, and thread **T2 needs** a lock on resource **R1**. In this case, among T1 and T2 no thread makes the first move and releases the lock which they already have and they both get stuck in an infinite amount of waiting time saying **"No, you release first"**.

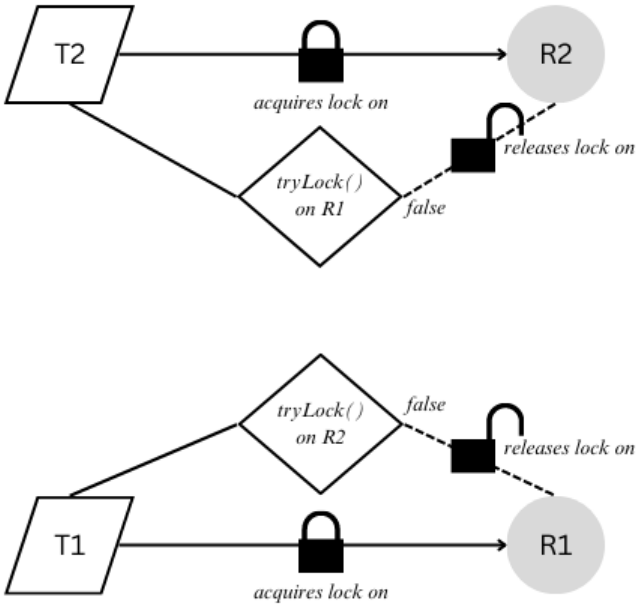


No, you release first

Livelock

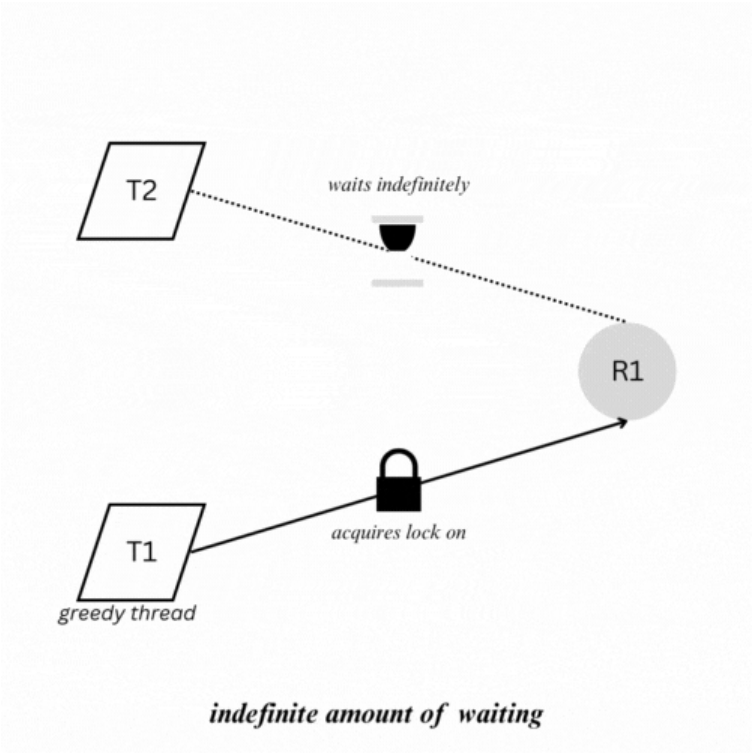
Livelock is a condition where two or more threads are continuously changing their state and tries to make progress but no one making any progress, they end up reaching where they started. It's like two people who meet face-to-face in a corridor, and both of them move aside to let the other pass and they end up making no progress.

Thread **T1 acquires** a lock on resource **R1**, and thread **T2 acquires** a lock on resource **R2**. Now thread **T1 needs** a lock on resource **R2**, and thread **T2 needs** a lock on resource **R1**. In this case, thread **T1 tries** to get the lock on **R2** if it **does not get it within some amount of time** it releases the lock which it's already having (i.e **releases the lock on R1**), and thread T2 does the same process if it **does not get a lock on R1 within some amount of time** it also releases the lock on R2. And after some time both T1 and T2 acquire the lock on R2 and R1 respectively and **end up reaching the situation where they started.**



Starvation

Starvation is a situation where a thread has to wait for an indefinite amount of time due to some high-priority threads taking longer time to execute or in case of shared resources are made unavailable for long periods of time by some greedy threads.



- In a deadlock, threads remain in the waiting state whereas, in a livelock threads continuously change their state but end up making no progress.
- In deadlock and livelock, no threads are making progress whereas, in case of starvation, high-priority or greedy thread is making progress.
- In a deadlock, there is no chance of getting the resource but in case of starvation, the thread will get the resource even after waiting for a long period of time
- In a deadlock, threads are not making any efforts to resolve the deadlock whereas in livelock threads are continuously making efforts to resolve it.
- Deadlock, livelock, and starvation all can decrease the overall efficiency of our application.

4.3. Write a program to demonstrate deadlock

Real-life example demonstrating the deadlock using synchronized methods.

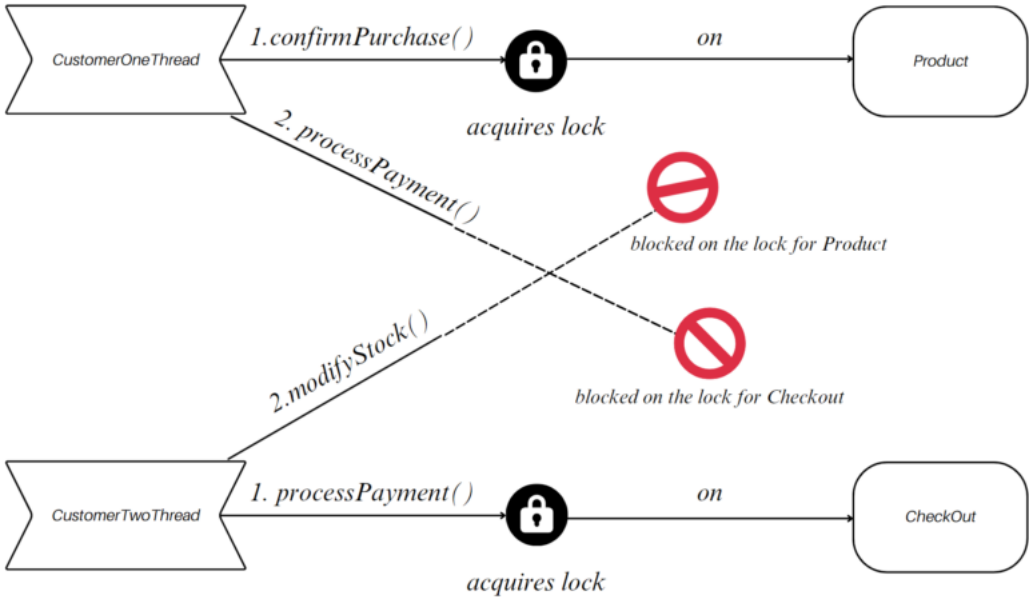
```
class Product {

    static Checkout checkout;

    public synchronized void confirmPurchase() {

        System.out.println(Thread.currentThread().getName() + " Confirming product pur

        try {
            System.out.println(Thread.currentThread().getName() + " Went away to anoth
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " Continuing process and

        checkout.processPayment();
```

```
    }

    public synchronized void modifyStock() {

        System.out.println(Thread.currentThread().getName() + " Modifying stock in pro

    }

}

class Checkout {

    static Product product;

    public synchronized void processPayment() {

        System.out.println(Thread.currentThread().getName() + " Started payment proces

        try {
            System.out.println(Thread.currentThread().getName() + " Taking some time t
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " Completed the payment

        product.modifyStock();
    }

}

class CustomerOne implements Runnable {

    Product product;
    Checkout checkout;

    public CustomerOne(Product product, Checkout checkout) {
        this.product = product;
        this.checkout = checkout;
    }

    public void run() {

        /* Customer One first buys the product and then process the payment*/
```

```
        product.confirmPurchase();
    }

}

class CustomerTwo implements Runnable {

    Product product;
    Checkout checkout;

    public CustomerTwo(Product product, Checkout checkout) {
        this.product = product;
        this.checkout = checkout;
    }

    public void run() {
        /*
         * Customer Two already added product to the cart and now directly
         * processing the payment.
         */

        checkout.processPayment();

    }

}

public class DeadLockRealLifeExample {

    public static void main(String[] args) {

        Product product = new Product();
        Checkout checkout = new Checkout();

        Product.checkout = checkout;
        Checkout.product = product;

        Thread customerOneThread = new Thread(new CustomerOne(product, checkout), "Cus

        Thread customerTwoThread = new Thread(new CustomerTwo(product, checkout), "Cus

        customerOneThread.start();
        customerTwoThread.start();

    }
```

```
}
```

In the above example, the real-life scenario of an e-commerce website is represented where two different customer threads are trying to process two synchronized methods and end up trapped in a deadlock.



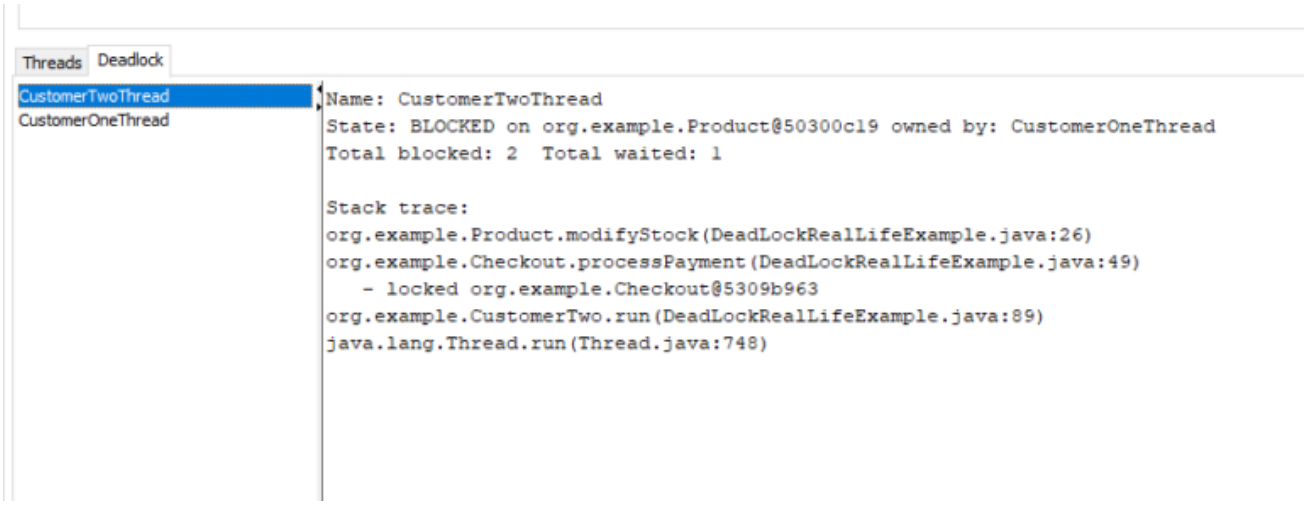
Let's understand how we are trapped in a deadlock in the above example. There are two customers in the above code, in which the first customer is confirming the product purchase so it is calling the *confirmPurchase()* method of the `Product` class, which is a *synchronized* method hence *CustomerOneThread* acquires the lock of the *Product* class. Now during the confirmation of the purchase, the first customer went away to some other website to cross-check the price. At that time the second customer logins and check out the products which are already added to the cart. So *CustomerTwoThread* calls the *processPayment()* method of the *CheckOut* class, which is again a *synchronized* method so *CustomerTwoThread* acquires the lock of the *CheckOut* class, and that payment takes some time to complete.

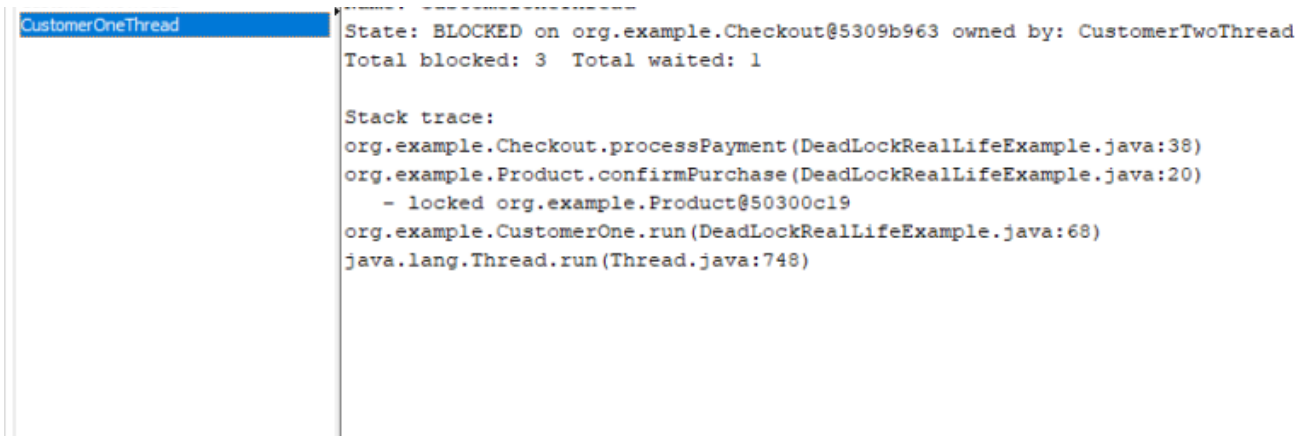
customer needs to process the payment, so it tries to call the *processPayment()* method of the *CheckOut* class but as it is a *synchronized* method, it requires the lock of the *CheckOut* class. But, the lock of the *CheckOut* class is already acquired by *CustomerTwoThread* so *CustomerOneThread* enters the *Blocked* state. Now after the *CustomerTwoThread* completed the payment, it tries to modify the product stock as he successfully bought the product, so it tries to call the *modifyStock()* method of the *Product* class which is also a synchronized method so, *CustomerTwoThread* requires the lock of the *Product* class, but that lock is already acquired by *CustomerOneThread* so, *CustomerTwoThread* also enters the *Blocked* state. And both are in the *Blocked* state waiting for each other to release the lock that they are holding on the class.

Output (it can be different in each run):

```
CustomerTwoThread Started payment process
CustomerOneThread Confirming product purchase
CustomerTwoThread Taking some time to verify payment with partner bank
CustomerOneThread Went away to another Site
CustomerTwoThread Completed the payment now need to modify the stock
CustomerOneThread Continuing process and now need to process the payment
......
......
......(infinite ammount of waiting)
```

Let's detect this deadlock in the *jconsole*.





4.4. What is race condition? how to find and fix it?

The race condition is a concurrency bug that may occur due to the concurrent execution of a critical section or shared resource by multiple threads without proper synchronization mechanisms. This may lead our program to a data inconsistency problem because the change in the sequence of execution of threads may produce a major change in the final result also.

There are two types of race conditions:

a. Read-Modify-Write

In this type of race condition, two or more threads **read the same value of a variable** and then modify the value. In this case, the new value of a variable somehow depends on the previous value.

For example, suppose two threads are trying to increment the value of a shared counter. If both threads read the current value of the variable and increment it independently, and then update in their turn. Then the final value of the counter will be incorrect because both threads are incrementing the value based on the same original value, rather than incrementing it based on each other's updates.

Let's understand this scenario by taking a real-world example. Suppose in a company the salary increment of an employee takes place in two phases. First associate hr level

increment by associate hr + increment by head hr.

```
public class ReadModifyWrite implements Runnable {

    static int salary = 100000;

    @Override
    public void run() {

        System.out.println(Thread.currentThread().getName() + " reading currentSalary"
        int currentSalary = salary;

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " incrementing currentSa
        salary = currentSalary += 10000;

    }

    public static void main(String[] args) throws InterruptedException {

        Thread.currentThread().setName("Head HR");

        Thread associateHrThread = new Thread(new ReadModifyWrite(), "Associate HR");
        associateHrThread.start();

        System.out.println(Thread.currentThread().getName() + " reading currentSalary"
        int currentSalary = salary;

        Thread.sleep(2000);

        System.out.println(Thread.currentThread().getName() + " incrementing currentSa
        salary = currentSalary += 20000;

        System.out.println("My Final Salary After Associate and Head Level HR Increame
    }
```

```
}
```

Output (it can be different in each run):

```
Head HR reading currentSalary
Associate HR reading currentSalary
Associate HR increamenting currentSalary
Head HR increamenting currentSalary
My Final Salary After Associate and Head Level HR Increment Is : 120000
```

In the above example, the final increment after associate and hr level should be 130000 but in the output, we are getting 120000 due to a race condition.

b. Check-Then-Act

In this type of race condition, two or more threads **check the same condition** and then act based on the result of the condition, which may also lead our program to a data inconsistency problem.

In the following example, both threads check the condition that whether *HashMap* contains the 101 key or not, and both get *true* as the result, and both try to put the value with the *101* key to the *HashMap*, and this generates data inconsistency in the *HashMap*.

```
import java.util.HashMap;

public class CheckThenAct implements Runnable {

    static HashMap<Integer, String> hashMap = new HashMap<>();

    @Override
    public void run() {
        if (!hashMap.containsKey(101)) {
            System.out.println(Thread.currentThread().getName() + " Checked that there
```

```
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            hashMap.put(101, "ValueBy" + Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) throws InterruptedException {

        Thread newThread = new Thread(new CheckThenAct(), "NewThread");
        newThread.start();

        if (!hashMap.containsKey(101)) {
            System.out.println(Thread.currentThread().getName() + " Checked that there
            Thread.sleep(2000);

            hashMap.put(101, "ValueBy" + Thread.currentThread().getName());
        }

        System.out.println(hashMap);
    }

}
```

Output (it can be different in each run):

```
main Checked that there is no Entry with 101 key in hashMap
NewThread Checked that there is no Entry with 101 key in hashMap
{101=ValueBymain}


-----------------------------------OR-----------------------------------------

main Checked that there is no Entry with 101 key in hashMap
NewThread Checked that there is no Entry with 101 key in hashMap
{101=ValueByNewThread}
```

only one thread at a time, and this can be done by proper use of synchronization.

Let's solve the problem that occurred in the Check-Then-Act race condition using *synchronized* blocks and class-level locking.

```
import java.util.HashMap;

public class CheckThenAct implements Runnable {

    static HashMap<Integer, String> hashMap = new HashMap<>();

    @Override
    public void run() {

        synchronized (CheckThenAct.class) {
            if (!hashMap.containsKey(101)) {
                System.out.println(
                        Thread.currentThread().getName() + " Checked that there is no

                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                hashMap.put(101, "ValueBy" + Thread.currentThread().getName());
            }
        }

    }

    public static void main(String[] args) throws InterruptedException {

        Thread newThread = new Thread(new CheckThenAct(), "NewThread");
        newThread.start();

        synchronized (CheckThenAct.class) {
            if (!hashMap.containsKey(101)) {
                System.out.println(
                        Thread.currentThread().getName() + " Checked that there is no
                Thread.sleep(2000);
```

```
                }
            }
            System.out.println(hashMap);
        }

    }
```

Output:

```
main Checked that there is no Entry with 101 key in hashMap
{101=ValueBymain}
```

In this case, the problem of data inconsistency will never raise as only one thread is adding the value in the *HashMap*.

5. Thread Safety

5.1. What is a Thread Safe class?

The term "thread safety" means different threads can access the same resource without exposing erroneous behaviors or producing unpredictable results.

To make the entire class thread safe we must ensure that the **internal state of an instance is accessed and modified by only one thread at a time**. There are multiple classes in Java that are already thread-safe. For example:

- StringBuffer
- Stack
- Vector

- Properties ...

We can create our own thread class also. There are multiple ways to achieve thread safety in our class among them, some are listed below.

- Using Synchronization
- *volatile* keyword
- Using Atomic classes
- Using final variables (i.e Immutable Implementation)
- Using ThreadLocal variables
- Using Reentrant Lock

A real-world scenario where a movie theater is having 4 different counters to book tickets and 6 customers are there, each trying to book 2 tickets for the same movie.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MovieTicketBooking {

    private final int totalSeat = 10;

    private int lastBookedSeat = 0;

    public synchronized String book(int numberOfSeats) {

        String response;

        if (lastBookedSeat == totalSeat) {
            response = "House Full! " + Thread.currentThread().getName();
        } else if (lastBookedSeat + numberOfSeats > totalSeat) {
            response = "Enough Seats Not Available " + Thread.currentThread().getName(
        } else {
```

```
            lastBookedSeat += numberOfSeats;
            response += lastBookedSeat + " Seats Booked " + Thread.currentThread().get
        }

        return response;
    }
}

class Customer {

    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(4); //4 threads mean

        MovieTicketBooking movieTicketBooking = new MovieTicketBooking();

        for (int i = 0; i < 6; i++) { //6 customers
            threadPool.submit(() -> {
                System.out.println(movieTicketBooking.book(2));
            });
        }
        threadPool.shutdown();
    }

}
```

Output (it can be different in each run):

```
1 to 2 Seats Booked pool-1-thread-1
7 to 8 Seats Booked pool-1-thread-2
5 to 6 Seats Booked pool-1-thread-3
3 to 4 Seats Booked pool-1-thread-4
9 to 10 Seats Booked pool-1-thread-4
House Full! pool-1-thread-4
```

In the above example `MovieTicketBooking` is the thread-safe class as every field is *private* and the method is also *synchronized*, If we remove the *synchronized* keyword from the method declaration then we will get unpredictable results.

The `ThreadLocal` class is present in the *"java.lang"* package and it's simply another way to achieve thread safety. It provides a way to declare thread-local **variables that can only be accessed and modified by the same thread**, by maintaining an individual and independently initialized copy of that variable for each thread. Even if two threads are executing the same code, and the code has a reference to the same `ThreadLocal` variable, the **two threads cannot see and modify each other's `ThreadLocal` variables**.

Once the thread enters into the DEAD or TERMINATED state, all its thread-local variables will become eligible for Garbage Collector.

Let's understand this by taking a real-world scenario where a chocolate box is having multiple chocolates, where 5 chocolates are reserved for each specific kids (i.e *KidLocal* chocolates), and In this case, two kids are eating chocolate from their own reserved chocolates.

```
public class ChocolateBox {
    private ThreadLocal<Integer> chocolates = new ThreadLocal<>() {
        @Override
        protected Integer initialValue() {
            return 5;
        }
    };


    public void eat() {
        Integer availableChocolate = chocolates.get();

        if (availableChocolate == 0) {
            System.out.println("Dear " + Thread.currentThread().getName() + " You've e
        } else {
            chocolates.set(--availableChocolate);
            System.out.println("Dear " + Thread.currentThread().getName() + " You ate
        }
    }

    public void availableChocolates() {
        System.out.println("Dear " + Thread.currentThread().getName() + " You have "
```

```
    }

}

class KidsEatingChocolate {

    public static void main(String[] args) {

        ChocolateBox chocolateBox = new ChocolateBox();

        Thread kid1 = new Thread(() -> {
            chocolateBox.eat();
            chocolateBox.eat();
            chocolateBox.eat();
            chocolateBox.availableChocolates();
        }, "KidOne");

        Thread kid2 = new Thread(() -> {
            chocolateBox.eat();
            chocolateBox.availableChocolates();
            chocolateBox.eat();
            chocolateBox.availableChocolates();
        }, "KidTwo");

        kid1.start();
        kid2.start();
    }
}
```

Output (it can be different in each run):

```
Dear KidTwo You ate one chocolate
Dear KidOne You ate one chocolate
Dear KidOne You ate one chocolate
Dear KidOne You ate one chocolate
Dear KidOne You have 2 Chocolates
Dear KidTwo You have 4 Chocolates
Dear KidTwo You ate one chocolate
Dear KidTwo You have 3 Chocolates
```

InheritableThreadLocal?

ThreadLocal	InheritableThreadLocal
The value of the thread-local variable is not available to any other thread.	The InheritableThreadLocal class is having its own method. 1. childValue() (Also gets all ThreadLocal class methods in inheritance)
The ThreadLocal class extends the Object class	The InheritableThreadLocal class extends the ThreadLocal class
The default initial value of the thread-local variable is the default value of the corresponding data type.	The default initial value of the thread-local variable is the value of the parent thread's thread-local variable.
The ThreadLocal class is having its own five methods. 1. get() 2. initialValue() 3. remove() 4. set() 5. withInitial()	The InheritableThreadLocal class is having its own method. 1. childValue() (Also inherits all ThreadLocal class methods in inheritance)

```
InheritableThreadLocal<Integer> threadLocal = new InheritableThreadLocal<>();
threadLocal.set(1000);

System.out.println(Thread.currentThread().getName() + " " + threadLocal.get()); // mai

Thread t = new Thread(() -> {
```

```
}, "ChildOfMain");
t.start();
```

Does the child thread have direct access to the parent thread's thread local variables?

No, the **child thread is not having any direct access to the parent thread's thread-local variable**. When we are creating a child thread at that time internally new copy of the thread-local variable is created for the child thread and that variable is initialized using the value of the parent thread's thread-local variable. After that initialization, if the value of the parent thread-specific thread-local variable is modified then that modification is not available to the child thread-specific thread-local variable.

```
public class InheritableThreadLocalDemo {

    public static void main(String[] args) throws Exception {

        InheritableThreadLocal<Integer> threadLocal = new InheritableThreadLocal<>();
        threadLocal.set(1000); // Setting parent thread specific thread local variable

        //Creating child thread
        Thread t = new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + " " + threadLocal.ge
            try {
                Thread.sleep(1000); //giving chance to parent thread to execute.
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " "
                    + threadLocal.get()); //Here child thread won't get modified value
        }, "ChildOfMain");
        t.start();

        Thread.sleep(500); //giving chance to child thread to execute.
        threadLocal.set(2004); // Modifying parent thread specific thread local variab
        System.out.println(Thread.currentThread().getName() + " " + threadLocal.get())
    }
```

```
}
```

Output (it can be different in each run):

```
ChildOfMain 1000
main 2004
ChildOfMain 1000
```

5.4. What is the *volatile* field? What does JVM guarantee for volatile fields?

In multithreaded applications, sometimes threads may copy variables from the main memory to the CPU cache to improve performance. And in the case of the multi-core processor, each thread may run on a different CPU. Hence threads may copy the variable in their respective CPU cache.

In the case of non-volatile variables, there is no guarantee that the JVM reads the data from the CPU cache or from the main memory and writes the data into the CPU cache or in the main memory. There raises the problem that **threads may not be able to see the latest value of the shared variable**, as the latest value is only modified in the cache memory of the respective previous thread's cache memory and not written back into the main memory, this problem is called a "**visibility problem**".

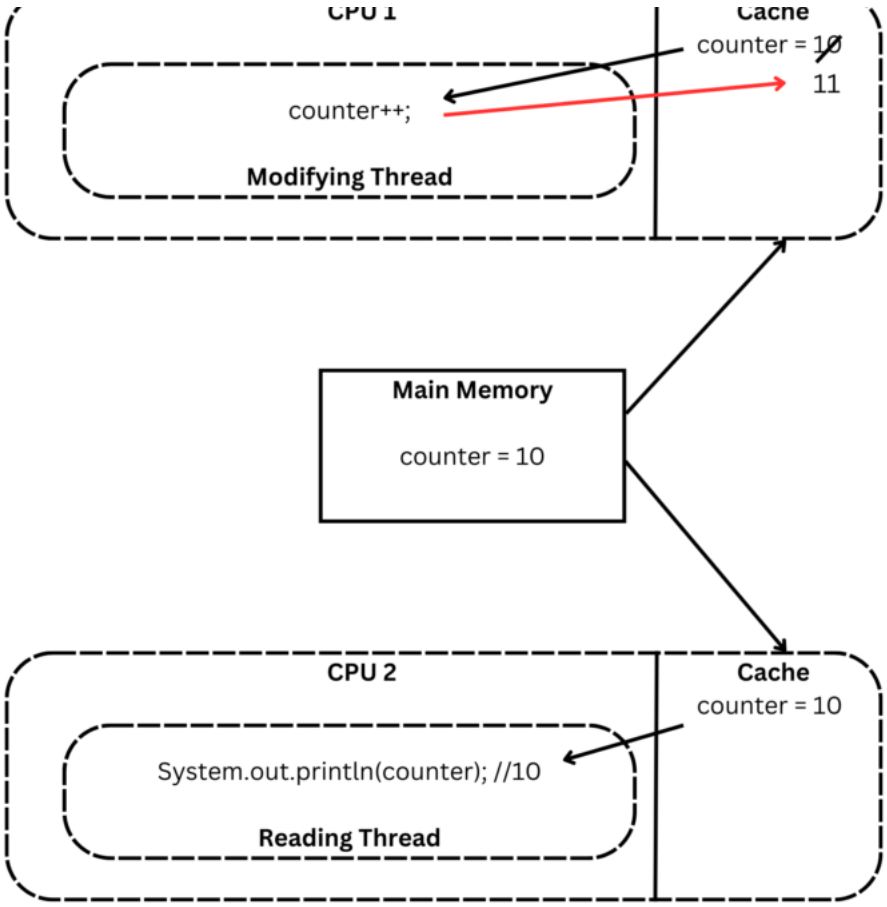
Let's take an example of a volatile field to understand this problem further.

```
class VisibilityProblem {

    private static int counter = 10;

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

```
            while (true) {
                if (counter == 11) {
                    System.out.println("Counter: " + counter);
                    break;
                }
            }
        });

        Thread modifyingThread = new Thread(() -> {
            System.out.print("Modifying Counter from " + counter);
            counter++;
            System.out.println(" To " + counter);
        });

        readingThread.start();
        modifyingThread.start();
    }

}
```

Output:

```
Modifying Counter from 10 To 11
...
...
...(infinite ammount of waiting)
```



Here comes the purpose of *volatile* fields, declaring a variable as *volatile*, **JVM gives volatile visibility guarantee** that all modifications to that variable will be **written back to the main memory immediately**, and **read** operations will also be made from the **main memory directly**.

In the above example if we mark the shared variable *counter* as *volatile* then the output will be:

```
Modifying Counter from 10 To 11
Counter: 11
```

JVM guarantees full visibility for volatile variables

memory all non-volatile variables' latest value will also be written to the main memory.

```
class FullVisibilityGuarantee {

    private static int rollNo;
    private static String name = "";
    private volatile static int marks;


    public static void main(String[] args) {

        Thread studentThread = new Thread(() -> {
            while (true) {
                if (marks > 90) {
                    System.out.println("Congratulation  " + rollNo + "_" + name + " yo
                    System.out.println("Congratulation You've Passes The Exam!");
                    break;
                }
            }
        });

        Thread teacherThread = new Thread(() -> {
            name = name + "Chandresh";
            rollNo = rollNo + 101;
            marks = marks + 92;

            System.out.println("Student " + rollNo + "_" + name + " got " + marks + "
        });

        studentThread.start();
        teacherThread.start();

    }

}
```

Output:

```
Congratulation  101_Chandresh you got 92 marks
Congratulation You've Passes The Exam!
Student 101_Chandresh got 92 marks out of 100
```

In the above program writing value of the volatile variable *marks*, the values of *name* and *rollNo* are also written to the main memory.

5.5. What is the atomic operation? What are atomic classes?

Atomic operations always execute together, there is no chance of partial completion of that operation. Atomic operations will either be completely executed or nothing executed. There is no term like partial execution of the atomic operation. It's the same as the **A(atomicity)** in the database's ACID property.

```
int i;
i = 1;
```

The above operation is atomic as simply 1 is assigned to the variable *i*. Either 1 will be assigned to that variable or it will remain uninitialized.

```
int i = 4;
i++;
```

The above operation looks atomic but it's not, it is divided into three parts,

- Read i
- Increment i
- Assign the incremented value to i

define its atomicity in a single-threaded application but in the case of a multi-threaded application, it may happen that without completing the entire operation thread may be context switched.

In multi-threaded applications due to this type of data inconsistency problem java defines some atomic classes in the ***java.util.concurrent.atomic*** package which provides wait-free and lock-free atomic operations on the variables. There are multiple atomic variable classes like,

- *AtomicBoolean*
- *AtomicInteger*
- *AtomicLong*
- *AtomicReference*
- *AtomicIntegerArray*
- AtomicLongArray

Let's solve the visibility problem using the `AtomicInteger`.

```
import java.util.concurrent.atomic.AtomicInteger;

class VisibilityProblemSolved {

    static AtomicInteger counter = new AtomicInteger(10);

    public static void main(String[] args) {

        Thread readingThread = new Thread(() -> {
            while (true) {
                if (counter.get() == 11) {
                    System.out.println("Counter: " + counter.get());
                    break;
                }
            }
```

```
        Thread modifyingThread = new Thread(() -> {
            System.out.print("Modifying Counter from " + counter.get());
            System.out.println(" To " + counter.incrementAndGet()); // incrementAndGet
        });

        readingThread.start();
        modifyingThread.start();
    }

}
```

Output:

```
Modifying Counter from 10 To 11
Counter: 11
```

In the above program, we will never get stuck in infinite waiting of thread as we are performing atomic operations.

6. Executor Framework

6.1. What is ExecutorService? Explain different implementations.

The *ExecutorService* is an interface present inside ***java.util.concurrent*** package, which allows us to submit tasks to be executed by threads asynchronously. *ExecutorService* creates and maintains a pool of reusable threads to execute the submitted tasks.

There are different implementations of the *ExecutorService* interface present in the same *java.util.concurrent* package,

- *ThreadPoolExecuter*

- *ForkJoinPool*
- *AbstractExexuterService* (abstract class)

ThreadPoolExecuter is used to execute the given *Callable* or *Runnable* tasks. There are two different ways to instantiate *ThreadPoolExecuter*,

1. By directly creating the object of *ThreadPoolExecuter*.

```
ExecutorService executorService = new ThreadPoolExecutor(1, 1,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
```

It creates a pool of thread with corePoolSize 1, maxPoolSize 1, and a keepAliveTime of 0ms (thread in the pool would stay alive unless explicitly closed) and an unbounded LinkedBlockingQueue.

Note: *ThreadPoolExecutor* maintains a queue in order to manage the submitted tasks when there is more number of tasks than the threads in the thread-pool.

2. By using factory methods of the *Executors* class.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
// Creates an Executor that uses a single worker thread operating off an unbounded que

ExecutorService executorService = Executors.newFixedThreadPool(10);
// Creates a thread pool that reuses 10 threads operating off a shared unbounded queue

ExecutorService executorService = Executors.newCachedThreadPool();
// Creates a thread pool that creates new threads as needed, but will reuse previously
```

There are different methods to delegate tasks for execution to the *ExecutorService*.

- Future submit(Runnable task)
- Future submit(Callable task)
- List<Future> invokeAll(Collection callableTasks)
- Object invokeAny(Collection callableTasks)

ScheduledThreadPoolExecuter is used to schedule tasks to run after a certain delay or to execute tasks repeatedly with a fixed interval of time between each execution.

6.2. Difference between shutdown(), shutdownNow() and awaitTermination()

ExecutorService interface provides 3 methods *shutdown()*, *shutdownNow()* and *awaitTermination()* for controlling the termination of tasks submitted to executors.

- The **shutdown() initiates an orderly shutdown** in which previously submitted tasks are executed, but no new tasks will be accepted.
- The **shutdownNow(), forcibly, attempts to stop all actively executing tasks**, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
- The **awaitTermination(long timeout, TimeUnit unit) blocks until all tasks have completed execution after a shutdown request**, or the timeout occurs, or the current thread is interrupted, whichever happens first. *Remember that* `awaitTermination()` *is invoked after a* `shutdown()` *request.*

> See Also: How to Shutdown a Java ExecutorService

ExecutorService interface provides 2 methods *submit()* and *execute()* for submitting the new tasks to thread pool:

submit()	execute()
Declared in the ExecutorService interface	Declared in the Executor interface
Can accept both *Runnable* and *Callable*	Can accept only *Runnable*
The return type is the *Future* object for checking if the task is completed, and for obtaining the result of the task or to cancel it.	The return type is *void*

6.4. How to schedule a task to run after a specific interval?

Java defines the *ScheduledExecutorService* interface that can schedule a task to run after a certain delay or execute a task repeatedly with a fixed interval of time between each execution.

For the purpose of running a task after a specific interval, the *ScheduledExecutorService* interface defines two different methods,

- ***scheduleWithFixedDelay (Runnable command, long initialDelay, long period, TimeUnit unit)***
  In the case of *scheduleWithFixedDelay()* the second execution will only start after the given delay time since the first execution was completed. The second execution will never start without the completion of the first execution. There is a guaranteed fixed delay between two executions of the task.

TimeUnit unit)

In the case of *scheduleAtFixedRate()* the second execution will start after the given delay time since the first execution started. If the first execution takes longer time to complete its execution than the given delay between the two executions, then the second execution will only start after the first execution ends.

Let's take an example that schedules a task using the *scheduleAtFixRate()* method with 5 seconds of the initial delay and 5 seconds of the time period between two execution.

```
import java.time.LocalDateTime;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledThreadPool {
    public static void main(String[] args) throws InterruptedException {
        ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThre

        Runnable runnableJob = () -> {
            System.out.println("Started Execution On:" + LocalDateTime.now());
            System.out.println("Hello World");
            System.out.println("Completed Execution On:" + LocalDateTime.now());
            System.out.println("\n");
        };

        scheduledExecutorService.scheduleAtFixedRate(runnableJob, 5, 5, TimeUnit.SECON
    }
}
```

The program executes at 15:13:39 and the after 5 seconds at 15:13:44.

```
Started Execution On:2023-03-17T15:13:39.257095491
Hello World
Completed Execution On:2023-03-17T15:13:39.265390688
```

```
Started Execution On:2023-03-17T15:13:44.212154070
Hello World
Completed Execution On:2023-03-17T15:13:44.212430386

...
...
```

7. Concurrency Classes

7.1. What is the *BlockingQueue* interface? How does it work?

BlockingQueue is a thread-safe version of `Queue`. It provides thread safety by blocking the thread trying to enqueue an element into a full queue until there is some space in the queue, and blocking the thread trying to dequeue one or more elements from an empty queue until there is an element inside the queue.

The `BlockingQueue` interface has 4 different sets of methods for inserting, removing, and examining the elements in the queue, one throws an exception, the second returns a special value (either null or true or false), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit.

	Throws Exception	***Special Value***	***Blocks***	***Times Out***
Insert	*add(e)*	*offer(e)*	*put(e)*	*offer(e, time, unit)*
Remove	*remove()*	*poll(), remove(e)*	*take()*	*poll(time, unit)*
Examine	*element()*	*peek(), contains(e)*	N/A	N/A

```
int drainTo(Collection<? super E> c);
/*removes all elements from the queue and adds them to the given collection and return
the number of elements added to the collection*/

int drainTo(Collection<? super E> c, int maxElements);
/*removes at most the given number of available elements elements from the queue and a
```

There are two types of blocking queues:

a. Unbounded Queue

In the unbounded blocking queue, the capacity is set to `Integer.MAX_VALUE`. In the case of an unbounded blocking queue while enqueuing elements into the queue thread will never be blocked as it could grow to a very large size. But dequeuing elements from an empty queue will block the thread.

```
BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>();
```

b. Bounded Queue

In the bounded queue, the capacity is defined at the time of object creation, and in this case while enqueuing elements into the queue the thread can be blocked in case of elements in the queue reach the given capacity.

```
BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>(5);
```

Since `BlockingQueue` is an interface, we need to use one of its implementations in order to use it. The `java.util.concurrent` packag has the following implementations of

- ArrayBlockingQueue
- LinkedBlockingQueue
- LinkedBlockingDeque
- DelayQueue
- PriorityBlockingQueue
- SynchronousQueue
- LinkedTransferQueue

7.2. What are the *Callable* and *Future*?

The *Callable* interface provides a way to create a thread that can return some computed result after completing the execution. It defines only one abstract method,

```
V call() throws Exception();
```

Hence it is also a Functional interface like *Runnable*. We can't create a new thread by passing *Callable* to the *Thread* class constructor as target runnable. There is only one way to use *Callable* is *ExecutorService*.

Using the ***submit()*** method of the *ExecutorService* we can execute the callable task, which returns the *Future* object.

```
<T> Future<T> submit(Callable<T> task);
```

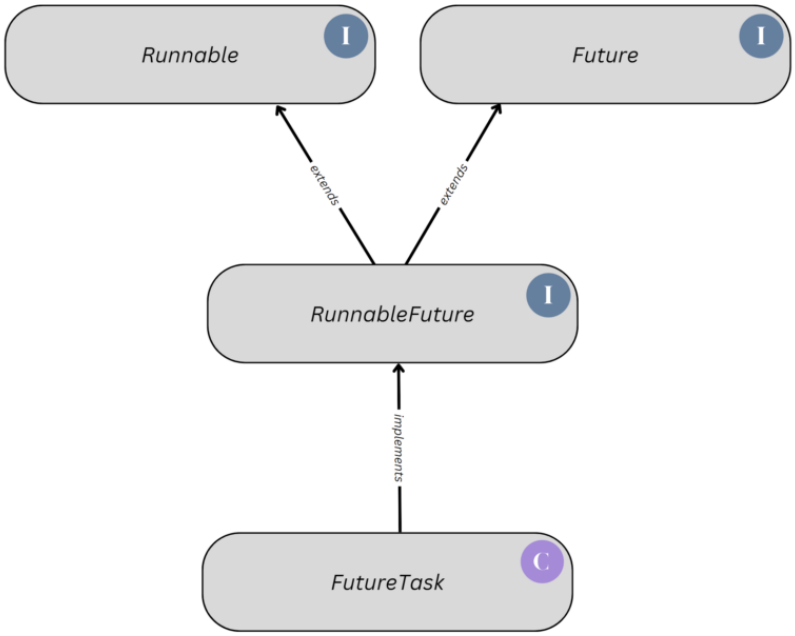
completed and to check the status of the task.

```
V get() throws InterruptedException, ExecutionException
//Waits if necessary for the computation to complete, and then retrieves its result.

V get(long timeout, TimeUnit unit);
//Waits if necessary for at most the given time for the computation to complete, and t

boolean isDone();
//Returns true if the task is completed.

boolean cancel(boolean mayInterruptIfRunning);
//Attempts to cancel execution of the task.

boolean isCancelled();
//Returns true if the task was cancelled before it completed normally.
```

> See Also: Java Callable Future Example

7.3. What is *FutureTask* class?

The ***FutureTask*** provides the concrete implementation of *Future*, *RunnableFuture*, and *Runnable* interfaces.



`Future` and `FutureTask` in Java allows you to write asynchronous code. The `FutureTask` class provides a base implementation of `Future`, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation. As it implements `Runnable` we can submit it to `ExecutorService` for execution. Calling *get()* methods on this class's object blocks the calling thread until the computation completes.

When calling *submit()* method of `ExecutorService`, it internally creates the object of the `FutureTask` with the given `Runnable` or `Callable` and returns it.

Let's understand this concept by an example.

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

public class Main {
    public static void main(String[] args) throws ExecutionException, InterruptedExcep

        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        Callable<Integer> sumOfTenNumbers = () -> { //asynchronous task to find sum of
```

```
            for (int i = 0; i < 10; i++) {
                Thread.sleep(500);
                sum += i;
            }
            return sum;
        };

        Future<Integer> result = threadPool.submit(sumOfTenNumbers);
        System.out.println(result instanceof FutureTask);//true i.e. submit() internal

        while (true) {
            if (result.isDone()) { //checking whether task is done
                System.out.println("Result : " + result.get());

                threadPool.shutdown();
                break;
            } else {
                System.out.println("Waiting For Result...");
                Thread.sleep(1000); //doing some other task instead of blocking
            }
        }
    }
}
```

In the above code, we are calling *submit(Callable<T> task)* method of the *ExecuterService* by passing the *Callable<Integer>* task that performs and returns the sum of the first 10 numbers. The *submit()* method returns the instance of *FutureTask*, which we are later using to check whether the task is done or not. If the task is done then we are getting the result, and if not then we are performing alternate activities instead of waiting on the result.

Output (it can be different in each run):

```
true
Waiting For Result...
Waiting For Result...
Waiting For Result...
Waiting For Result...
Waiting For Result...
```

```
Result : 45
```

7.4. Explain *CountDownLatch* and *CyclicBarrier*

The *CountDownLatch* class allows one or more threads to wait for some other thread to complete a certain number of tasks. The *CountDownLatch* object is initialized with an *int count*, the thread which calls *await()* method on it immediately enters into the WAITING state and waits for that count to become 0. As soon as the count becomes 0 the waiting thread enters into the RUNNABLE (Ready to Run) state.

CountDownLatch class overloads the *await()* method as *await(long timeout, TimeUnit unit)* to accept a timeout, the thread which calls the await method by passing a timeout, immediately enters into the TIMED WAITING state and either the count becomes 0 or the given timeout elapses whatever happens first, the waiting thread enters to the RUNNABLE (Ready to Run) state.

As the *await()* method puts the calling thread into the WAITING state, it throws the *InterruptedException*. So it's compulsory to either handle the exception in catch block or the method must explicitly declare it using throws declaration.

Now, suppose we want to stop a scheduled task to be stooped after exactly two executions that are running continuously after a specific interval of time, Let's solve it using *CountDownLatch*.

We can initialize the *CountDownLatch* of 2 by passing 2 as the parameter to the *CoutDownLatch* constructor, then let the main thread waits on that *CountDownLatch* to become 0 before shutting down the *ScheduledExecutorService*. Every time the scheduled task executes we can *countDown()* the *CountDownLatch*, which will decrement the counter value by 1. So exactly after two execution of the scheduled task, the *ScheduledExecutorService* will be *shutDown()*.

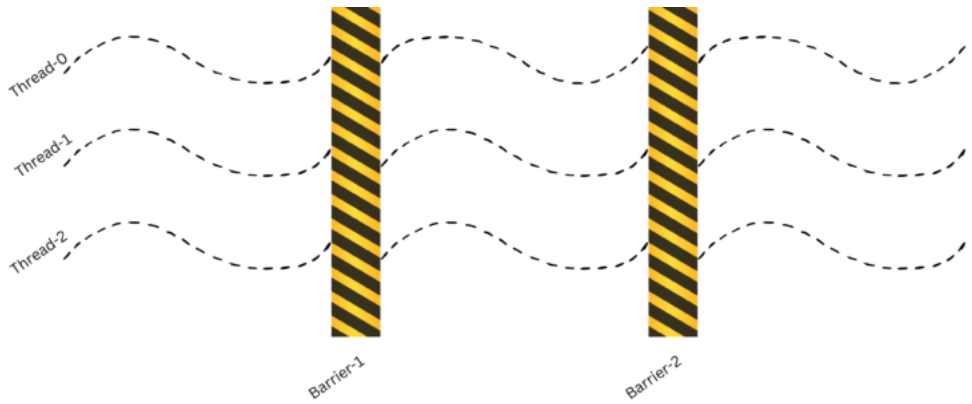
```
import java.time.LocalDateTime;
import java.util.concurrent.*;

public class ScheduledThreadPool {

    public static void main(String[] args) throws InterruptedException, ExecutionExcep

        ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThre

        CountDownLatch count = new CountDownLatch(2);

        Runnable runnableJob = () -> {
            System.out.println("Started Execution On:" + LocalDateTime.now());
            for (int i = 0; i < 5; i++) {
                System.out.println("Hello World : " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println("Completed Execution On:" + LocalDateTime.now());
            count.countDown();
            System.out.println("\n");
        };

        scheduledExecutorService.scheduleAtFixedRate(runnableJob, 5, 5, TimeUnit.SECON
        count.await();
        scheduledExecutorService.shutdown();

    }
}
```

Output:

```
Started Execution On:2023-03-17T18:20:30.650018020
Hello World : 0
Hello World : 1
Hello World : 2
Hello World : 3
Hello World : 4
```

```
Started Execution On:2023-03-17T18:20:35.652813671
Hello World : 0
Hello World : 1
Hello World : 2
Hello World : 3
Hello World : 4
Completed Execution On:2023-03-17T18:20:40.654577372
```

The ***CyclicBarrier*** class allows threads to wait for each other at the given barrier to let every thread complete its execution until that barrier. No thread can continue further until all threads reach that barrier. Hence it provides one kind of synchronization mechanism for threads.



The *CyclicBarrier* object is initialized with an *int count*, which defines how many threads have to reach that barrier to release all threads waiting at the barrier. Once that number of threads reaches that barrier all threads can continue their execution.

The thread can wait at the specific barrier by calling *await()* method on that specific *CyclicBarrier* object. *await()* method in *CyclicBarrier* class is overloaded as *await(long timeout, TimeUnit unit)* which allows threads to specify the waiting time at the barrier. If the given number of threads does not reach that barrier until the specified time, then the waiting thread which specified the time throws *TimeoutException* and brokes the barrier, and continues its execution. In this case, the barrier is broken hence all threads waiting at

execution.

At the time of initializing the *CyclicBarrier* object, we can pass a *Runnable barrierAction* also, which defines the job to be performed once the last thread arrives at that barrier. And the last arriving thread is responsible for performing that *Runnable* action.

Let's understand both *CountDownLatch* and *CyclicBarrier* concepts using an example.

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

public class CollectingDataFromDifferentFiles {

    public static void main(String[] args) throws Exception {

        CountDownLatch count = new CountDownLatch(2);

        Runnable mergeJob = () -> {
            System.out.println("\n");
            System.out.println("Merging Data Into A Final File");
            System.out.println("Task Performed By:" + Thread.currentThread().getName()
            count.countDown();
        };

        CyclicBarrier firstFiveFileBarrier = new CyclicBarrier(3, mergeJob);
        CyclicBarrier endBarrier = new CyclicBarrier(3, mergeJob);

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        executorService.submit(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    System.out.println(Thread.currentThread().getName() + " Getting Da

                    if (i == 5) {
                        firstFiveFileBarrier.await();
                    }
                }
                endBarrier.await();
            } catch (InterruptedException | BrokenBarrierException exception) {
                System.out.println(Thread.currentThread().getName() + "...." + excepti
            }
```

```
        executorService.submit(() -> {
            try {
                for (int i = 11; i <= 20; i++) {
                    System.out.println(Thread.currentThread().getName() + " Getting Da

                    if (i == 15) {
                        firstFiveFileBarrier.await();
                    }
                }
                endBarrier.await();
            } catch (InterruptedException | BrokenBarrierException exception) {
                System.out.println(Thread.currentThread().getName() + "...." + excepti
            }
        });

        executorService.submit(() -> {
            try {
                for (int i = 21; i <= 30; i++) {
                    System.out.println(Thread.currentThread().getName() + " Getting Da

                    if (i == 25) {
                        firstFiveFileBarrier.await();
                    }
                }
                endBarrier.await();
            } catch (InterruptedException | BrokenBarrierException exception) {
                System.out.println(Thread.currentThread().getName() + "...." + excepti
            }
        });

        count.await();
        executorService.shutdown();

    }

}
```

In the above program, our main aim is to collect data from 30 different files and merge them into one file. For this purpose, we have created a thread pool of 3 threads, so 1

will wait for each other to complete reading from the first 5 files.

After the last thread arrives we've also defined a *Runnable* job to merge that data to the final file which will be performed by the last arriving thread. At the second barrier also the last arriving thread will perform the merge job.

Output (it can be different in each run):

```
pool-1-thread-1 Getting Data From File:1
pool-1-thread-1 Getting Data From File:2
pool-1-thread-2 Getting Data From File:11
pool-1-thread-3 Getting Data From File:21
pool-1-thread-2 Getting Data From File:12
pool-1-thread-2 Getting Data From File:13
pool-1-thread-2 Getting Data From File:14
pool-1-thread-1 Getting Data From File:3
pool-1-thread-2 Getting Data From File:15
pool-1-thread-3 Getting Data From File:22
pool-1-thread-3 Getting Data From File:23
pool-1-thread-1 Getting Data From File:4
pool-1-thread-3 Getting Data From File:24
pool-1-thread-1 Getting Data From File:5
pool-1-thread-3 Getting Data From File:25


Merging Data Into A Final File
Task Performed By:pool-1-thread-3


pool-1-thread-3 Getting Data From File:26
pool-1-thread-2 Getting Data From File:16
pool-1-thread-2 Getting Data From File:17
pool-1-thread-1 Getting Data From File:6
pool-1-thread-3 Getting Data From File:27
pool-1-thread-1 Getting Data From File:7
pool-1-thread-1 Getting Data From File:8
pool-1-thread-2 Getting Data From File:18
pool-1-thread-1 Getting Data From File:9
pool-1-thread-3 Getting Data From File:28
pool-1-thread-1 Getting Data From File:10
pool-1-thread-2 Getting Data From File:19
```

```
pool-1-thread-2 Getting Data From File:20
pool-1-thread-3 Getting Data From File:30


Merging Data Into A Final File
Task Performed By:pool-1-thread-3
```

7.5. Explain Semaphore and Reentrantlock

Semaphore

The `Semaphore` provides the thread synchronization mechanism. The `Semaphore` is a counting semaphore that **internally works on the concept of permits**. The `Semaphore` class object is initialized with an int number, defining the set of permits.

```
Semaphore countingSemaphore = new Semaphore(5); // at a time at most 5 threads can ent
```

It has two main methods,

```
public void acquire();
/*Acquires a permit, if one is available and returns immediately, and reduce the numbe

public void release();
/*Releases a permit, and increase the number of available permits by one.*/
```

Note: If the permits counter value reaches zero, it means all the shared resources are in use by other threads, then the *`acquire()`* method blocks the current *Thread* until a permit is available to the *Semaphore*.

and provides simple mutual exclusion that allows only one thread to access the shared resource (i.e. enter a critical section) at a time.

```
Semaphore binarySemaphore = new Semaphore(1);

// Acquiring semaphore
binarySemaphore.acquire();

// Printing Available Permits
System.out.println(binarySemaphore.availablePermits())      // 0

// Releasing semaphore
binarySemaphore.release();

// Printing Available Permits
System.out.println(binarySemaphore.availablePermits())      // 1
```

Reentrantlock

The `ReentrantLock` implements the `Lock` interface, which provides a mutual exclusion mechanism that allows the same thread to reenter into a lock on the same resource (multiple times) without any deadlock problem.

It has two methods:

```
/*Acquires the lock if it is not held by another thread and returns immediately, setti
public void lock();


/*Attempts to release this lock.
If the current thread is the holder of this lock then the hold count is decremented. I
public void unlock();
```

Note: The lock on the resource will be released only when a hold count reaches to 0.

- If you need a simple mutual exclusion mechanism, there is no specific reason to use a binary semaphore over a ReentrantLock. But, if your requirement is that more than one thread (but a limited number) can enter a critical section, ReentrantLock is not helpful, whereas Semaphores can fulfill this requirement.
- Semaphores may also be released by another thread, as it **uses a non-ownership release mechanism**. According to Java docs, such behavior may be applicable in some specialized contexts like *Deadlock* recovery. But, *Deadlock* Recovery is a bit difficult in the case of *Reentrant Lock* as it **uses ownership of a *Thread* to a resource by physically locking it** and only the owner *Thread* can unlock that resource. If the non-owner attempts to unlock the lock, it will immediately raise *IllegalMonitorStateException*.
- Semaphores are not reentrant in nature, i.e. you can't acquire a semaphore for the second time in the same thread. attempting to do so will lead to a *deadlock* problem. But, ReentrantLock allows a *Thread* to lock a particular resource multiple times using `lock()` method, without any deadlock problem.

> See Also: Java Semaphore vs ReentrantLock

7.6. What are Concurrent Collection Classes?

Java contains a very rich API of collection classes that are non-synchronized in nature hence in the multi-threaded environment they are not thread-safe, hence we are responsible for taking care of thread safety.

To properly use these collection classes in the multi-threaded environment, we have to make these collection classes *synchronized*. To do so we can get the synchronized version of a particular collection by using `synchronizedCollection()` method of the `Collections` class.

```
List is = Collections.synchronizedList(Arrays.asList(1, 2, 3, 4, 5));  //to get the sy
```

But in this case, there is also a problem that the synchronizing of these collections is achieved by serializing all access to the collection's state. i.e. locking the entire collection object which may affect the overall performance of the application.

To solve this issue java introduced **concurrent collection classes**, packaged into ***java.util.concurrent*** package from Java version 5. These concurrent collection classes are a set of collections APIs that are designed and optimized specifically for synchronized multithreaded access. Concurrent collections are **much more performant than synchronized collections**.

For example, `ConcurrentHashMap` is the class that we need when we want to use a synchronized hash-based `Map` implementation. Similarly, if we want a thread-safe `List`, we can actually use the `CopyOnWriteArrayList` class, if we want a thread-safe `HashSet`, we can actually use the `CopyOnWriteArraySet`. There are plenty of like this.

- BlockingQueue
- ArrayBlockingQueue
- LinkedBlockingQueue
- LinkedBlockingDeque
- LinkedTransferQueue
- PriorityBlockingQueue
- DelayQueue
- SynchronousQueue
- ConcurrentHashMap
- ConcurrentSkipListMap

- CopyOnWriteArraySet ...

8. Programming Exercises

8.1. How many threads are there in the below program after Line 1?

```
class MyThread extends Thread{
    public void run(){
        System.out.println("Child Thread");
    }
}

public class Test{
    public static void main(String[] args){
        MyThread t = new MyThread();
        t.run(); //------------------- Line1
        System.out.println("Main Thread");
    }
}
```

There is only **one thread** in the above program after executing line 1.

As in the above code, we are calling the *run()* method instead of the *start()* method on the `Thread` class object, no new thread will be created. The *start()* method present in the `Thread` class is mainly responsible to spawn a new thread and **register it with the thread scheduler**. Hence in the above case, the *run()* method will be executed by the *main* thread just like a normal method call, and in this case, we will always get the fixed output.

```
Main Thread
Child Thread
```

An interrupt is a kind of signal to the thread that it should stop doing what it is doing and perform some other activity, which depends upon us what we want the thread to perform. A thread can interrupt another thread **by calling the *interrupt()*** method on that `Thread` object which is to be interrupted.

The interrupt mechanism is internally implemented using an internal flag known as the interrupt status. When the new thread is spawned initially the interrupt status is set to *false*. When any thread interrupts that thread the interrupt status is set to *true*, which means that the thread has to perform some other activity and clear the interrupt status by again setting its value to *false*.

The `Thread` class defines three methods to deal with interrupt status.

```
public static boolean interrupted();
/*Tests whether the current thread has been interrupted, and if it's interrupted then


public boolean isInterrupted(); //Tests whether this thread has been interrupted only.

public void interrupt(); //Interrupts this thread. (the thread can interrupt itself al
```

Thread blocking methods like *wait(), join(), sleep(), …* already defines a way to handle the interrupt which is by throwing the checked exception `InterrupterException`. Hence during the thread is in any of the above method's corresponding block state, if we call the *interrupt()* method on its object, or after calling the *interrupt()* method if the thread enters that blocking state any time during its life cycle, then we will immediately get the *java.lang.InterruptedException*.

```
class ChildThread extends Thread {
    @Override
    public void run() {
```

```
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        System.out.println("Done! The Sum Is: " + sum + ". Now Going To Sleep!");

        try {
            Thread.sleep(1000);
        } catch (InterruptedException exception) {
            System.out.println("Got Interrupted! " + exception.getMessage());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ChildThread childThread = new ChildThread();
        childThread.start();

        System.out.println("Let's Interrupt Child Thread!");
        childThread.interrupt();
    }
}
```

Output (it can be different in each run):

```
Let's Interrupt Child Thread!
Performing Sum Of 100 Numbers
Done! The Sum Is: 4950. Now Going To Sleep!
Got Interrupted! sleep interrupted
```

What if the thread never enters the blocking state in its entire life cycle? In that case, we have to periodically check the interrupt status using the static *interrupted()* method or instance *isInterrupted()* method and provide the way to handle that interrupt.

```
class ChildThread extends Thread {
    @Override
```

```
        System.out.println( "Performing Sum Of 100 Numbers" );
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            if (Thread.interrupted()) {
                System.out.println("Got Interrupted! ");
                break;
            } else {
                sum += i;
            }
        }
        System.out.println("The Sum is: " + sum);
    }
}

public class Main {
    public static void main(String[] args) {
        ChildThread childThread = new ChildThread();
        childThread.start();

        System.out.println("Let's Interrupt Child Thread!");
        childThread.interrupt();
    }
}
```

Output (it can be different in each run):

```
Let's Interrupt Child Thread!
Performing Sum Of 100 Numbers
Got Interrupted!
The Sum is: 0
```

In the above code instead of *break*, we can also throw `InterruptedException` manually. Hence, these are the ways to interrupt a thread and handle that interrupt.

What if the thread never checks for interrupt status and never enters into a blocking state in its entire life cycle? In that case, the interrupt to that thread will be wasted and only the

interrupt status. So it's never guaranteed that we can always interrupt a thread.

8.3 How to pass the parameter to the thread?

We can pass the parameter to Java threads to dynamic the behavior of threads depending upon the parameter value. There are multiple ways to pass parameters to the thread let's discuss it one by one.

Using Parameterized Constructor

We can create a parameterized constructor of the class which either `implements Runnable` or `extends Thread` to accept the parameters that we want to use in the *run()* method of that class.

```
class EchoThread implements Runnable { //OR extends Thread
    private String message;

    EchoThread(String message) {
        this.message = message;
    }

    @Override
    public void run() {
        System.out.println("Echo: " + message); //Echo: Hello New Thread!
    }
}

public class Main {
    public static void main(String[] args) {
        new Thread(new EchoThread("Hello New Thread!")).start();
    }
}
```

Using Setter Method

value of that variable from outside of the class and then we can use it in the *run()* method.

```
class EchoThread implements Runnable { //OR extends Thread
    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public void run() {
        System.out.println("Echo: " + message);  //Echo: Hello New Thread!
    }
}

public class Main {
    public static void main(String[] args) {
        EchoThread echo = new EchoThread();
        echo.setMessage("Hello New Thread!");

        new Thread(echo).start();
    }
}
```

Note: In this case, we have to compulsory initialize all variables using the setter methods of each, otherwise we may get the `NullPointerException`.

Using Effectively Final Variables

We can access the parent class variable directly inside the inner class or lambda function, but that variable should be effectively final.

```
public class Main {
    public static void main(String[] args) {
        String message = "Hello Child Thread!";

        //Lambda Function
```

```
            System.out.println("Echo: " + message + " By " + Thread.currentThread().ge
            //Echo: Hello Child Thread! By Thread-0
        };
        Thread subTask = new Thread(subTaskWithLambda);
        subTask.start();

        //Anonymous Inner Class
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Echo: " + message + " By " + Thread.currentThread(
                //Echo: Hello Child Thread! By Thread-1
            }
        }).start();
    }
}
```

Using Static Variables

we can declare a static variable inside the class which we will be directly assigned using the class name from outside of the class. But this way is not recommended as for every instance of the thread there will be only one copy of that variable so there is a very high chance of the data inconsistency problem.

```
class EchoThread implements Runnable { //OR extends Thread
    static String message;

    @Override
    public void run() {
        System.out.println("Echo: " + message);  //Echo: High chance of Data Inconsist
    }
}

public class Main {
    public static void main(String[] args) {
        EchoThread.message = "High chance of Data Inconsistency Problem!";
        new Thread(new EchoThread()).start();
```

```
}
```

8.4 What is Java Thread Dump, How can we get Java Thread dump of a program?

A thread dump is a snapshot of the state of all the threads of a Java process. We can also say that a thread dump is a way of finding out what every thread in the JVM is doing at a particular point in time. That snapshot contains the state of each thread in the form of the stack trace of each, and it is useful for diagnosing problems that occurred in our Java application. It is highly recommended to take more than 1 thread dump while diagnosing any problem.

There are multiple ways to take the thread dump of any java program. Let's discuss them one by one.

jstack

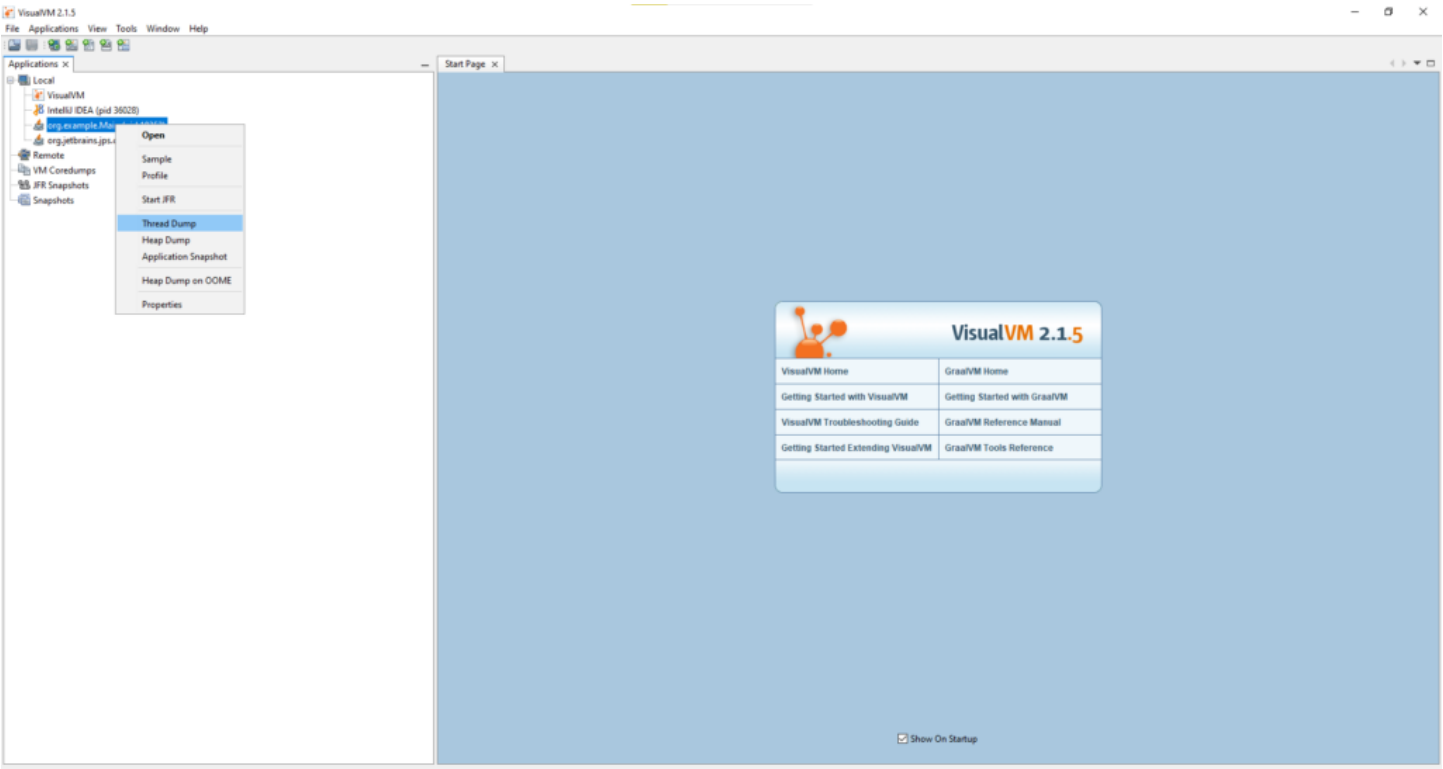
jstack is a command line tool to capture thread dumps. *jstack* tool is present inside the JDK_HOME/bin folder as a part of JDK since Java version 5.

```
jstack -l  <pid>
# this command prints the entire thread dump to the command line console.
# we can find all java process pids using jps command.

jstack -l  <pid> > <file-path>
# this command capture the thread dump in the given file.
```

jvisualvm

Java VisualVM is a graphical user interface tool that lets us monitor, troubleshoot, and profile Java applications. From JDK 9, Visual VM isn't included in the Oracle JDK and Open JDK distributions so we have to download it from the official website.

your machine. Right-click on the process you want to take the thread dump and then click the Thread Dump option present in the menu.



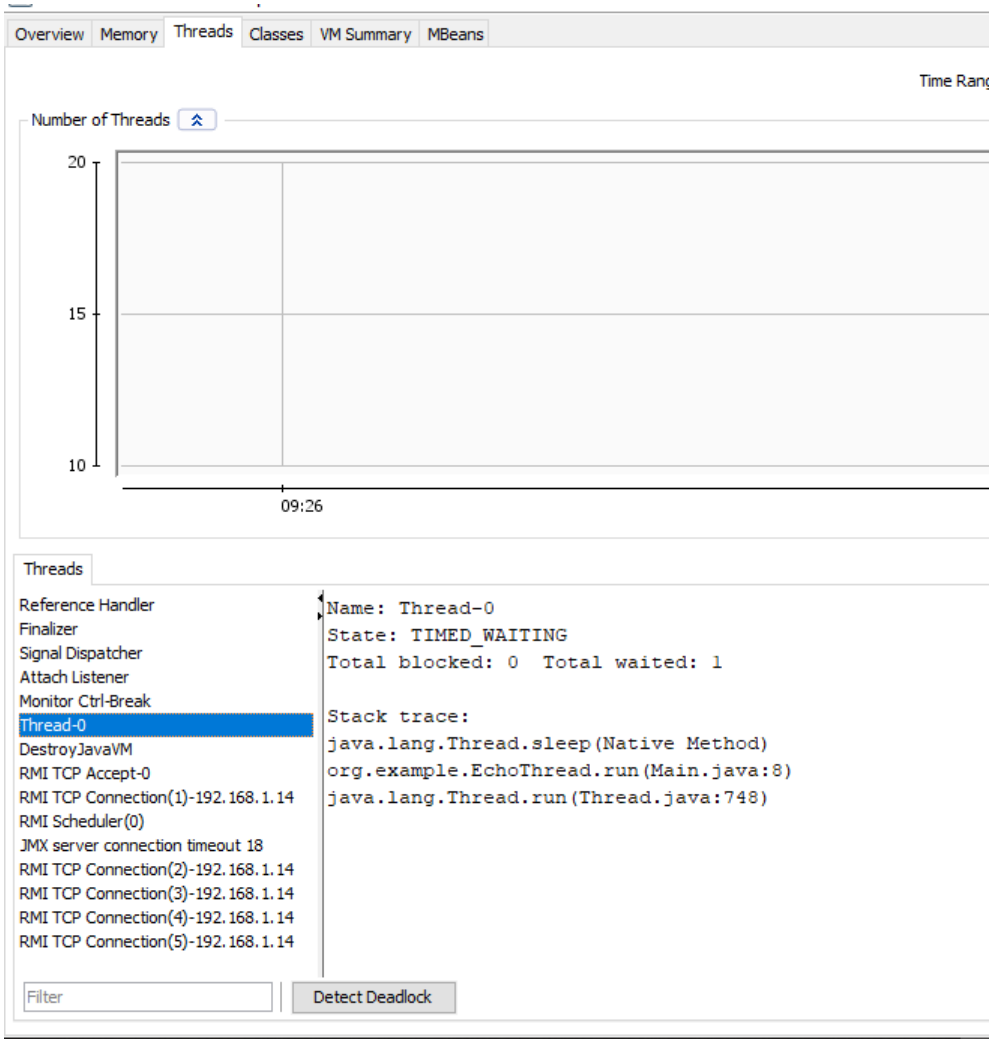
jcmd

The *jcmd* tool was introduced in Oracle JDK 7, and it internally works by sending the commands requests to the JVM. There are many commands with various capabilities like getting thread dump, getting heap dump, and getting all java process ids, …

```
jcmd <pid> Thread.print > <file-path>
# we can find all java process pids using jcmd command
```

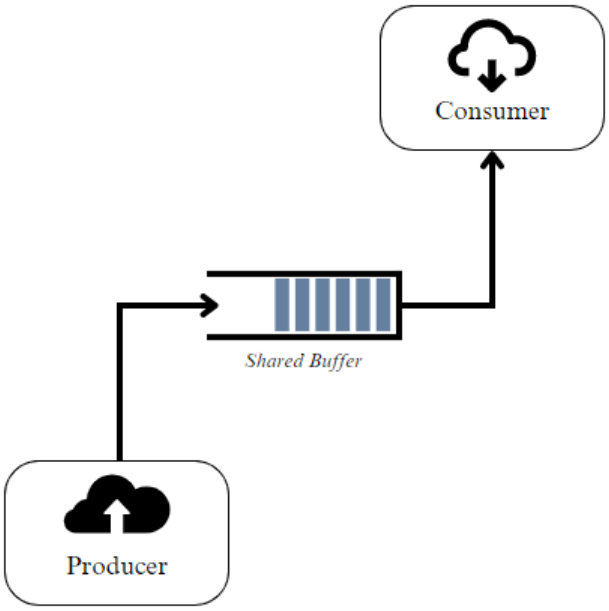
jconsole

The *jconsole* tool is a JMX-compliant graphical tool for monitoring a Java virtual machine. In the *jconsole*, we can inspect the individual stack trace of each thread.



8.5 How to solve the producer-consumer problem?

The producer-consumer problem is a very popular synchronization problem. In the producer-consumer problem, there are two threads, one is *ProducerThread* whose job is to continuously produce the items and the second is *ConsumerThread* whose job is to continuously consume the items. The producers and consumers share the same memory buffer that is of fixed size in which the producer put the produced item and from which the consumer consumes the items.



Then What's The Problem?

There are multiple problems with it.

1. The producer should produce data only when the buffer is not full. Otherwise, it can lead our program to buffer overflow.
2. The consumer should consume data only when the buffer is not empty. Otherwise, it can lead our program to buffer underflow.
3. The producer and consumer should not access the buffer at the same time. Otherwise, there is a very high chance of a data-inconsistency problem.

Then What Can Be The Solution?

- Synchronization on the shared buffer while producing and consuming items.
- When the buffer is empty the consumer should wait until the producer produces and add something to the buffer.
- When the buffer is full the producer should wait until the consumer consumes something from the buffer.

BlockingQueue, which you can find on Producer Consumer Problem Using BlockingQueue.

9. Conclusion

The Java concurrency interview questions guide lists some important and tricky questions to help refresh the basic concepts and some advanced concepts related to threading in Java. These should help you in attending the interview more confidently.

Happy Learning!

DISCOVER MORE

Related Articles And Resources

- Java String Interview Questions
- Java Interview Questions and Answers
- Java HashMap Interview Questions
- Java Collections Interview Questions
- Real Java Interview Questions asked in Oracle
- Java OOP Interview Questions with Answers

Leave a Comment

Name *

Email *

Website

Post Comment

Search ...

Weekly Newsletter

Stay Up-to-Date with Our Weekly Updates. Right into Your Inbox.

Email Address

Subscribe

About Us

HowToDoInJava provides tutorials and how-to guides on Java and related technologies.

It also shares the best practices, algorithms & solutions and frequently asked interview questions.

Tutorial Series

Regex

Maven

Logging

TypeScript

Python

Meta Links

About Us

Advertise

Contact Us

Privacy Policy

Our Blogs

REST API Tutorial