



Multithreading in Java

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Multitasking

Multitasking allows several activities to occur concurrently on the computer.

- Process-based multitasking
- Thread-based multitasking



3.51 / 1:58:09 • Multitasking >

Process Based Multitasking :

Allows processes (i.e programs) to run concurrently on the computer.
Eg: Running the Ms Paint while also working with the word processor.

Thread Based Multitasking

Allows parts of the same program to run concurrently on the computer.
Eg: Ms Word that is printing and formatting text at the same time.

Threads vs Process :

- Two threads share the same address space
- Context switching between threads is usually less expensive than between processes.
- The cost of communication between threads is relatively low.



Why MultiThreading?

- In a single-threaded environment, only one task at a time can be performed.
- CPU cycles are wasted, for example, when waiting for user input.
- Multitasking allows idle CPU time to be put to good use.

Threads :

- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).
- They also share the process running the program.



Threads :

- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).
- They also share the process running the program.



The Main Thread :

- When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the main thread.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.
- All other threads, called child threads, are spawned from the main thread.



Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

The Main Thread :

- When a standalone application is run, a user thread is created to execute the main() method of the application. This is called the main thread.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.
- All other threads, called child threads, are spawned from the main thread.
- The main() method can then finish, but the program will keep running until all user threads have completed.
- The runtime environment distinguishes between user threads and daemon threads.



15:50 / 1:58:09 • The Main Thread >

The Main Thread :

- Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.
- As long as a user thread is alive, the JVM does not terminate.
- A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program.



15:50 / 1:58:09 • The Main Thread >

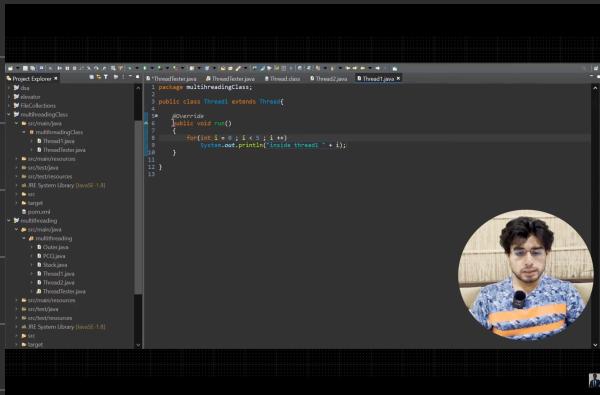
Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Thread Creation :

- A thread in Java is represented by an object of the Thread class.
- Creating threads is achieved in one of two ways:
 1. Implementing the java.lang.Runnable interface
 2. Extending the java.lang.Thread class

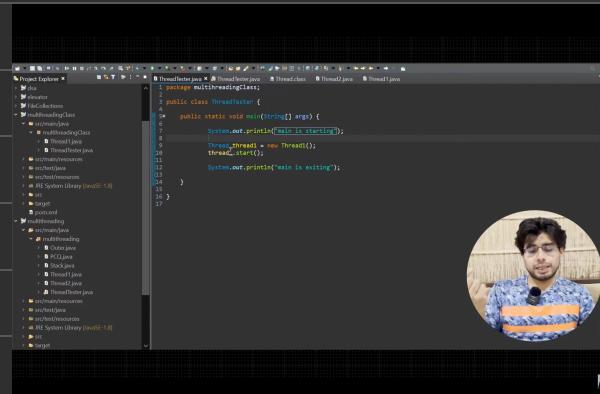


15:49 / 1:58:09 • Thread Creation in Java >



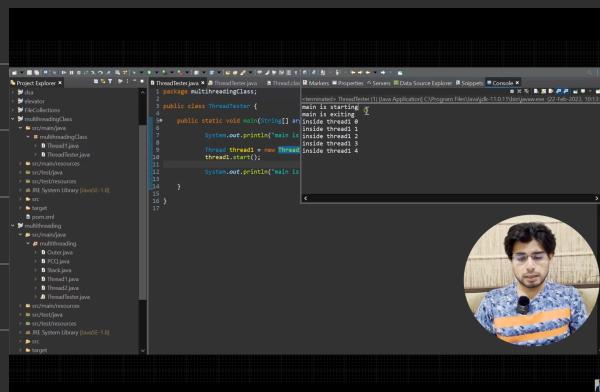
Project Explorer

```
package multithreadingClass;
public class Thread extends Thread{
    @Override
    public void run(){
        for(int i=0; i<5; i++){
            System.out.println("Inside thread " + i);
        }
    }
}
```



Project Explorer

```
package multithreadingClass;
public class ThreadTester {
    public static void main(String[] args) {
        System.out.println("Main is starting");
        Thread t = new Thread();
        t.start();
        System.out.println("Main is exiting");
    }
}
```



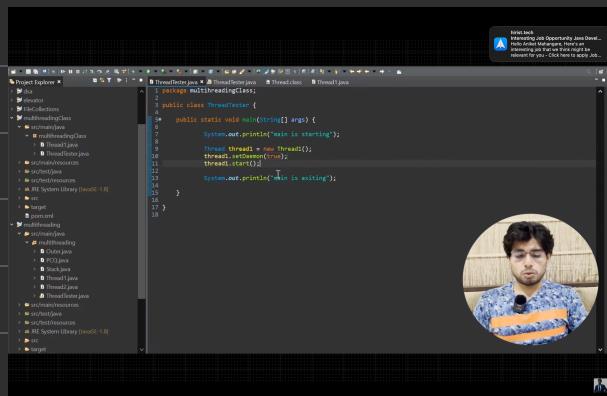
Project Explorer

```
package multithreadingClass;
public class ThreadTester {
    public static void main(String[] args) {
        System.out.println("Main is starting");
        Thread t = new Thread();
        t.start();
        System.out.println("Main is exiting");
    }
}
```

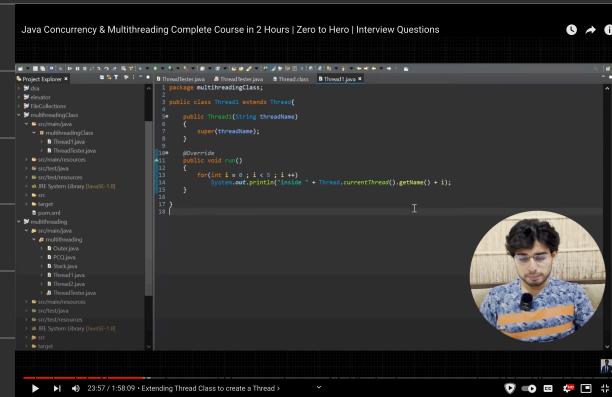


no order
guaranteed.

Note: program will continue running until all user threads are completed, if the daemon thread is running, then the program may or may not terminate.



```
package multithreadingClass;
public class ThreadTester {
    public static void main(String[] args) {
        System.out.println("main is starting");
        Thread thread = new Thread();
        thread.setName("new Thread()");
        thread.start();
        System.out.println("main is exiting");
    }
}
```



```
package multithreadingClass;
public class ThreadTester extends Thread{
    public ThreadTester(String threadName)
    {
        super(threadName);
    }
    @Override
    public void run()
    {
        for(int i = 1; i < 5; i++)
            System.out.println("Inside " + Thread.currentThread().getThreadName() + " " + i);
    }
}
```

Extending Thread class.

```
build.gradle (multithreading.java) Main.java ExtendingThreadClass.java
```

```
1 package org.nsk;
2
3 class ThreadClass extends Thread {
4     public ThreadClass(String name) {
5         super(name);
6     }
7
8     @Override
9     public void run() {
10        for (int i = 0; i < 5; i++) {
11            System.out.println("inside thread: " + Thread.currentThread().getname() + " " + i);
12        }
13    }
14
15    public class ExtendingThreadClass {
16        public static void main(String[] args) {
17            System.out.println("starting main thread");
18            Thread thread = new ThreadClass("name: " + "Thread1");
19            thread.start();
20            System.out.println("exiting main thread");
21        }
22    }
23}
```

```
> Task :ExtendingThreadClass.main()
starting main thread
exiting main thread
inside thread: Thread1 0
inside thread: Thread1 1
inside thread: Thread1 2
inside thread: Thread1 3
inside thread: Thread1 4
```

User Thread

```
build.gradle (multithreading.java) Main.java ExtendingThreadClass.java
```

```
1 package org.nsk;
2
3 class ThreadClass extends Thread {
4     public ThreadClass(String name) {
5         super(name);
6     }
7
8     @Override
9     public void run() {
10        for (int i = 0; i < 5; i++) {
11            System.out.println("inside thread: " + Thread.currentThread().getname() + " " + i);
12        }
13    }
14
15    public class ExtendingThreadClass {
16        public static void main(String[] args) {
17            System.out.println("starting main thread");
18            Thread thread = new ThreadClass("name: " + "Thread1");
19            thread.setDaemon(true);
20            thread.start();
21            System.out.println("exiting main thread");
22        }
23    }
24}
```

```
6:09:59 pm: Executing ':ExtendingThreadClass.main()'...
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :ExtendingThreadClass.main()
starting main thread
exiting main thread
```

Daemon Thread

Implementing
Runnable
interface.

Better than
extending as
Java supports
to extend only
one class.

The screenshot shows an IDE interface with several tabs: build.gradle, Main.java, ExtendingThreadClass.java, and ImplementingRunnableInterface.java. The ImplementingRunnableInterface.java tab is active, displaying the following code:

```
1 package org.nik;
2
3 public class ThreadTwo implements Runnable {
4
5     @Override
6     public void run() {
7         for (int i = 0; i < 5; i++) {
8             System.out.println("inside thread: " + Thread.currentThread().getName() + ", " + i);
9         }
10    }
11
12    public class ImplementingRunnableInterface {
13        public static void main(String[] args) {
14            System.out.println("starting main thread");
15            Thread t1 = new Thread(new ThreadTwo(), "threadTwo");
16            t1.start();
17            System.out.println("Exiting main thread");
18        }
19    }
20
21 }
```

The output window below the code shows the execution results:

```
starting main thread
exiting main thread
inside thread: threadTwo, 0
inside thread: threadTwo, 1
inside thread: threadTwo, 2
inside thread: threadTwo, 3
inside thread: threadTwo, 4
```

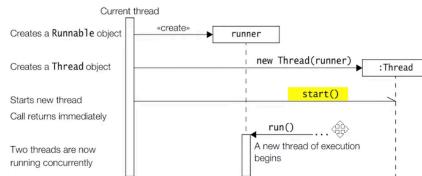
The screenshot shows an IDE interface with several tabs: gradle (multithreading-java), Main.java, ExtendingThreadClass.java, and ImplementingRunnableInterface.java. The ImplementingRunnableInterface.java tab is active, displaying the following code:

```
1 package org.nik;
2
3 public class ThreadWithRunnableLambda {
4
5     public static void main(String[] args) {
6         System.out.println("Starting main thread");
7         Thread t3 = new Thread(() -> {
8             for (int i = 0; i < 5; i++) {
9                 System.out.println("inside thread: " + Thread.currentThread().getName() + ": " + i);
10            }
11        }, "thread3");
12        t3.start();
13        System.out.println("Exiting main thread");
14    }
15 }
```

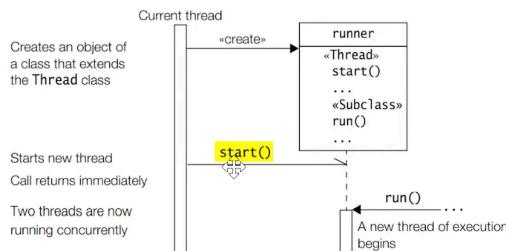
The output window below the code shows the execution results:

```
starting main thread
exiting main thread
inside thread: thread3: 0
inside thread: thread3: 1
inside thread: thread3: 2
inside thread: thread3: 3
inside thread: thread3: 4
```

Implementing Runnable Interface



Extending Thread Class

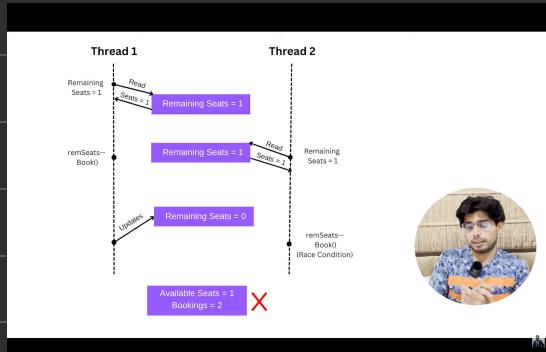


Synchronization :

- Threads share the same memory space, i.e. they can share resources (objects)
- However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource.



Race Conditions



```
public class SafeStack {  
    int[] arr; 5 usages  
    int stackTop; 7 usages  
    final Object lock; 3 usages  
  
    public SafeStack(int capacity) { 1 usage  
        this.arr = new int[capacity];  
        this.stackTop = -1;  
        this.lock = new Object();  
    }  
  
    public boolean push(int value) { 1 usage  
        synchronized (lock) {  
            if (isFull()) return false;  
            arr[++stackTop] = value;  
            return true;  
        }  
    }  
  
    // public synchronized int pop() --> this is also valid,  
    // it uses the current object of SafeStack class as lock  
    // every object in java can be used as lock, except primitives like int  
    public int pop() { 1 usage  
        synchronized (lock) { // using same lock as above  
            if (isEmpty()) return Integer.MIN_VALUE;  
            int returnVal = arr[stackTop];  
            arr[stackTop] = Integer.MIN_VALUE;  
            stackTop--;  
            return returnVal;  
        }  
    }  
  
    public boolean isEmpty() { 1 usage  
        return stackTop == -1;  
    }  
  
    public boolean isFull() { 1 usage  
        return stackTop == arr.length - 1;  
    }  
}
```

Thread Safe
Stack class

```

public static void main(String[] args) {
    System.out.println("Starting main thread");
    SafeStack stack = new SafeStack(capacity: 10);

    Thread pusherThread = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
            System.out.println(stack.push(i));
        }
    }, name: "pusher");

    Thread popperThread = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
            System.out.println(stack.pop());
        }
    }, name: "popper");

    pusherThread.start();
    popperThread.start();
}

System.out.println("Exiting main thread");
}

```

Testing Thread safe stack class.

```

1 package singleton;
2
3 public class TVSet {
4
5     private static volatile TVSet tvSetInstance = null;
6
7     private TVSet() {
8         System.out.println("TV Set instantiated");
9     }
10
11    public static TVSet getTVSetInstance() {
12        if(tvSetInstance == null) { // optimisation
13            synchronized(TVSet.class) { // t2
14                if(tvSetInstance == null) // double checking
15                    tvSetInstance = new TVSet();
16            }
17        }
18        // heavy work done here
19        return tvSetInstance;
20    }
21
22
23 // time 0 - t1 , t2
24 // time 5 - t5 , t6 , t7.....
25

```

for static method,
we can make a
method **synchronized**

Singleton pattern
with **volatile** keyword
& **synchronized** block.

Notice that, we are
using **TVSet.class**
here, as a lock.

← ↑
which does the same

Synchronized Methods :

- While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.
- This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object.
- Such a method can invoke other synchronized methods of the object without being blocked.
- The non-synchronized methods of the object can always be called at any time by any thread.



Rules of Synchronization :

- A thread must acquire the object lock associated with a shared resource before it can enter the shared resource.
- The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with it.
- If a thread cannot immediately acquire the object lock, it is blocked, i.e., it must wait for the lock to become available.
- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can try to acquire the lock in order to gain access to the shared resource.



Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Rules of Synchronization :

- It should be made clear that programs should not make any assumptions about the order in which threads are granted ownership of a lock.

1:04:36 / 1:58:09 • Rules of Synchronization >

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Static Synchronized Methods :

- A thread acquiring the lock of a class to execute a static synchronized method has no effect on any thread acquiring the lock on any object of the class to execute a synchronized instance method.
- In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.
- A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.

1:05:51 / 1:58:09 • Rules of Synchronization >

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Race Condition :

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



1:06:12 / 1:58:09 • Race Condition >

Synchronized Blocks :

- Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.
- The general form of the synchronized statement is as follows:
synchronized (object ref expression) { <code block> }
- The object ref expression must evaluate to a non-null reference value, otherwise a NullPointerException is thrown.



Summary :

A thread can hold a lock on an object :

- By executing a synchronized instance method of the object. (this)
- By executing the body of a synchronized block that synchronizes on the object. (this)
- By executing a synchronized static method of a class or a block inside a static method (in which case, the object is the Class object representing the class in the JVM)



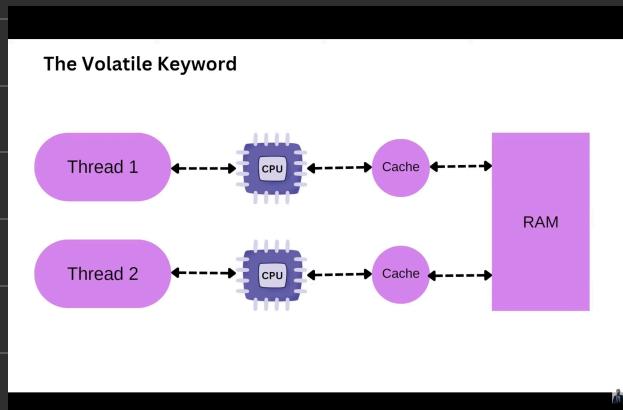
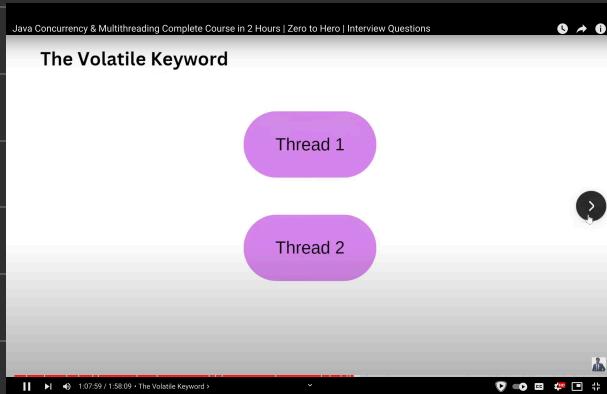
Thread safety :

It's the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads. Eg StringBuffer.

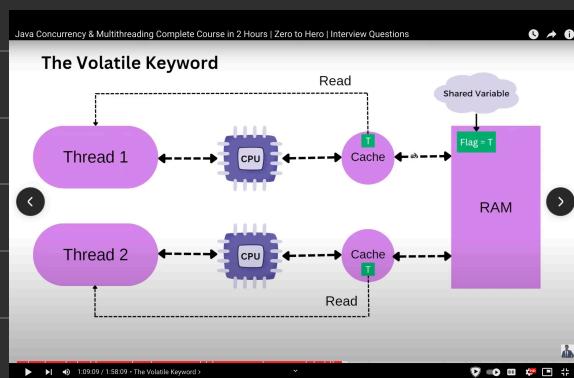
- *StringBuffer* is ThreadSafe
- *StringBuilder* is not ThreadSafe.



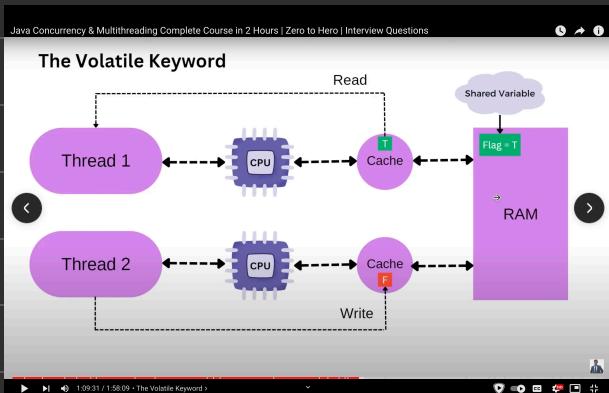
Volatile keyword



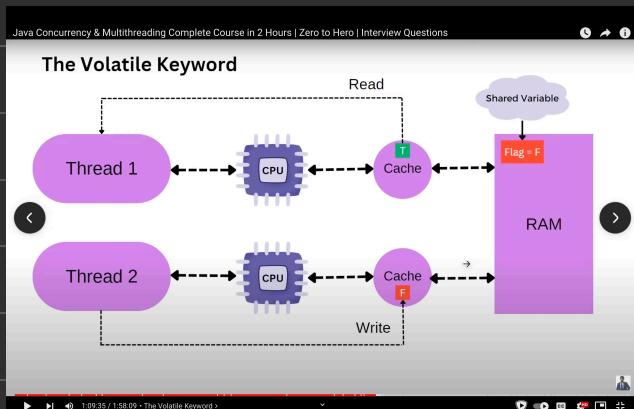
Threads store data in cache at times, as it's more efficient



Threads read value of shared variable from cache

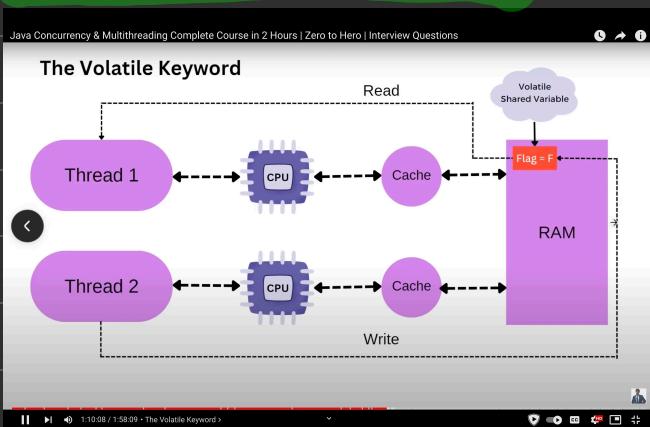


Thread 2 updates the value to false, which initially gets updated in its cache. → T1 reads value T

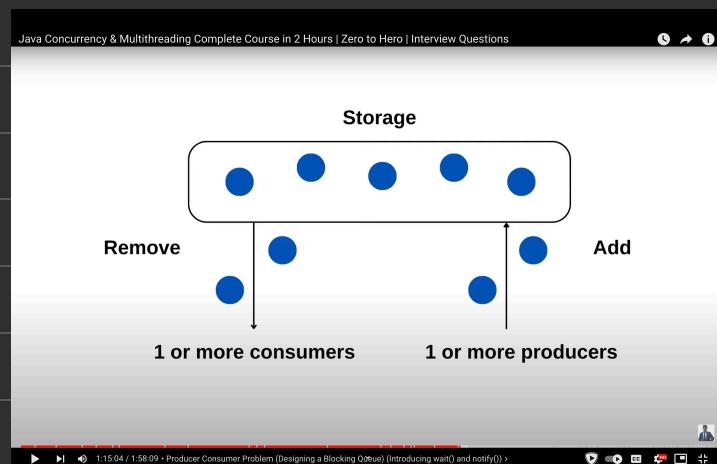
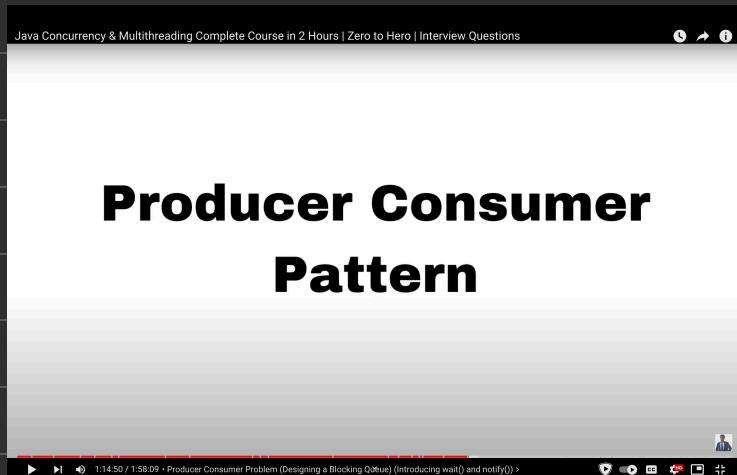


After some time, the write is reflected in main memory. → T1 reads value F

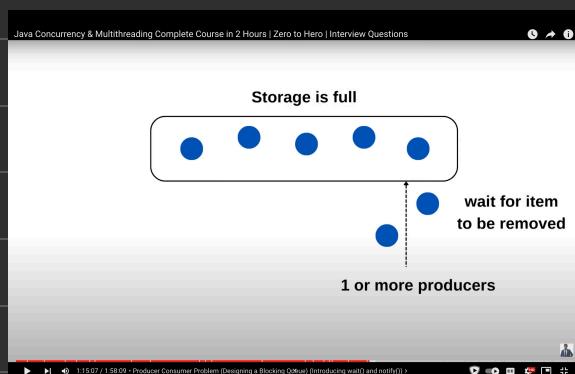
Should be used in Singleton Pattern



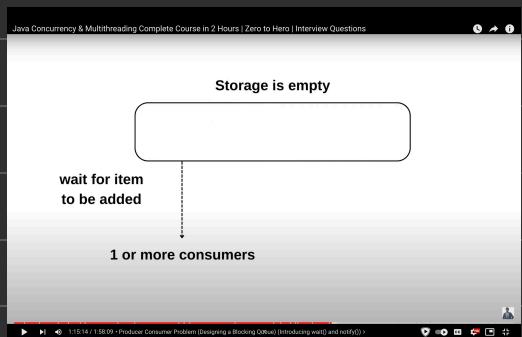
Volatile keyword ensures that the shared variable is stored only in RAM & not cached, which makes reads & writes consistent.



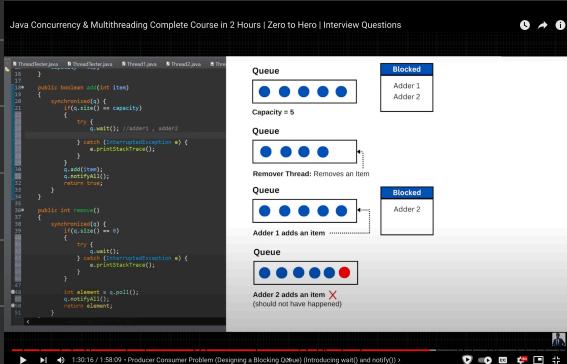
Publishers are trying to publish messages to queue, whilst consumers are trying to consume messages from queue.



If storage is full, publishers can't publish to queue.



If queue is empty, then
Consumers can't consume
from the queue.



Problem →
because of if

Solution:

Use while

```

public boolean add(int item)
{
    synchronized(q) {
        while(q.size() == capacity)
        {
            try {
                q.wait(); //Adder1 , adder2
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        q.add(item);
        q.notifyAll();
        return true;
    }
}

public int remove()
{
    synchronized(q) {
        while(q.size() == 0)
        {
            try {
                q.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        int element = q.poll();
        q.notifyAll();
        return element;
    }
}

```

```

package org.nik;

import java.util.LinkedList;

public class BlockingQueue {
    private final LinkedList<Integer> queue; // 1 usages
    private final int capacity; // 2 usages

    public BlockingQueue(int capacity) { // 1 usage
        this.queue = new LinkedList<>();
        this.capacity = capacity;
    }

    public void put(int item) { // 1 usage
        synchronized (queue) {
            while (queue.size() == capacity) {
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    System.out.println(e.getMessage());
                }
            }
            queue.add(item);
            queue.notifyAll();
        }
    }

    public int remove() { // 1 usage
        synchronized (queue) {
            while (queue.isEmpty()) {
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    System.out.println(e.getMessage());
                }
            }
            int top = queue.poll();
            queue.notifyAll();
            return top;
        }
    }
}

```

Thread safe blocking queue.

```

public static void main(String[] args) {
    System.out.println("starting main thread");
    BlockingQueue blockingQueue = new BlockingQueue(capacity: 10);

    Thread publisherThread = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            blockingQueue.put(i);
        }
    }, name: "publisher");

    Thread consumerThread = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            int top = blockingQueue.remove();
            System.out.println(top);
        }
    }, name: "consumer");

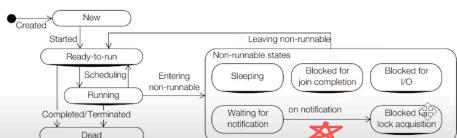
    publisherThread.start();
    consumerThread.start();
    System.out.println("exiting main thread");
}

```

Publisher &
consumer threads
working on
blocking queue.
↓
can use multiple
threads here.

Thread Transitions

Thread States



Sleeping thread
doesn't release
lock

|| ▶ ⏴ 1:33:55 / 1:58:09 Thread States and Thread Transitions ▶

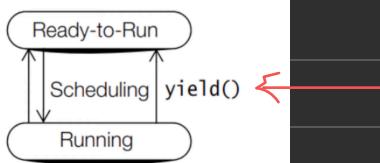
Thread States		
Constant in the Thread.State enum type	State in Figure 13.3	Description of the thread
NEW	New	Created but not yet started.
RUNNABLE	Runnable	Executing in the JVM.
BLOCKED	Blocked for lock acquisition	Blocked while waiting for a lock.
WAITING	Waiting for notify, Blocked for join completion	Waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING	Sleeping, Waiting for notify, Blocked for join completion	Waiting for another thread to perform an action for up to a specified time.
TERMINATED	Dead	Completed execution.

```

Thread thread3 = new Thread(() -> {
    try {
        thread3.sleep(1);
        for(int i = 10000; i > 0; i--);
    } catch (InterruptedException e){
        e.printStackTrace();
    }, "States");
thread3.start();
while(true)
{
    Thread.State state = thread3.getState();
    System.out.println(state);
    if (state == Thread.State.TERMINATED) break;
}
  
```

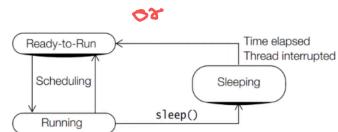
TIMED_WAITING
 TIMED_WAITING
 TIMED_WAITING
 TIMED_WAITING
 TIMED_WAITING
 TIMED_WAITING
 TIMED_WAITING
 RUNNABLE
 RUNNABLE
 RUNNABLE
 TERMINATED

Running and Yielding :

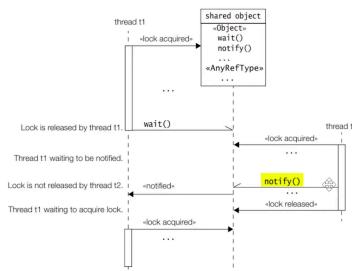
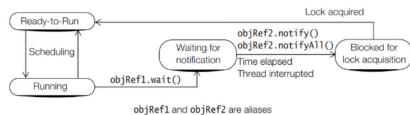


This is just an advice to the JVM, and it doesn't guarantee that the thread gets into the ready to run state immediately

Sleeping and Waking Up :



Waiting and Notifying :

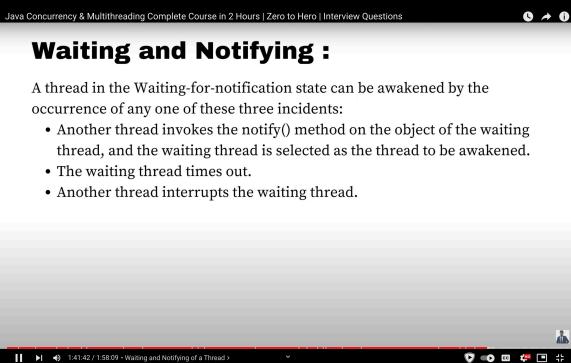


Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Waiting and Notifying :

A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

- Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
- The waiting thread times out.
- Another thread interrupts the waiting thread.



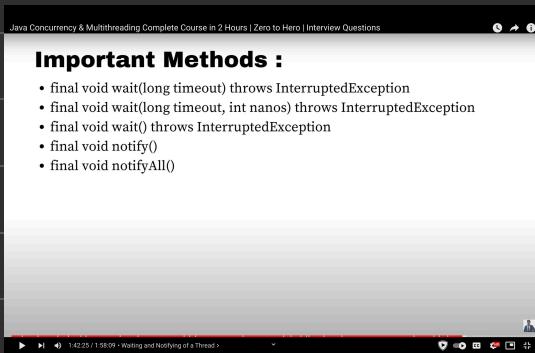
Notify :

- Invoking the notify() method on an object wakes up a single thread that is waiting for the lock of this object.
- The selection of a thread to awaken is dependent on the thread policies implemented by the JVM.
- On being notified, a waiting thread first transits to the Blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready-to-run state.
- The thread is also removed from the wait set of the object.

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Important Methods :

- final void wait(long timeout) throws InterruptedException
- final void wait(long timeout, int nanos) throws InterruptedException
- final void wait() throws InterruptedException
- final void notify()
- final void notifyAll()



Timed Out :

- The wait() call specified the time the thread should wait before being timed out, if it was not awakened by being notified.
- The awakened thread competes in the usual manner to execute again.
- The awakened thread has no way of knowing whether it was timed out or woken up by one of the notification methods.

In timed out, thread
doesn't know if it got
timed out or notified
by other thread.

Interrupted :

- Another thread invoked the interrupt() method on the waiting thread.
- The awakened thread is enabled , but the return from the wait() call will result in an InterruptedException if and when the awakened thread finally gets a chance to run.
- The code invoking the wait() method must be prepared to handle this checked exception.

In interrupted, a thread
knows it got interrupted.

```

01 // ThreadDemo.java
02 // ThreadDemo
03 // ThreadTypes
04 // ThreadDead
05 // ThreadJoin
06 // System.out.println("Popped : " + stack.pop());
07 // }
08 // Thread thread = new Thread();
09 // try {
10 //     Thread.sleep(1);
11 //     System.out.println("Sleeping : " + q);
12 // } catch (InterruptedException e) {
13 //     e.printStackTrace();
14 // }
15 // }
16 // States();
17 // }
18 // thread.start();
19 // while(true) {
20 //     if (Thread.currentThread().isInterrupted()) {
21 //         System.out.println("Status : " + Thread.currentThread().getStackTrace());
22 //         if (state == Thread.State.RETURNED) break;
23 //     }
24 // }
25 // Thread thread = new Thread();
26 // System.out.println(Thread.currentThread());
27 // }
28 // ThreadDemo();
29 // }
30 // thread.start();
31 // try {
32 //     thread.join();
33 // } catch (InterruptedException e) {
34 //     e.printStackTrace();
35 // }
36 // System.out.println("main is exiting");
37 // }
38 // 
```



join() waits for the
child thread to get completed.
↓
join(time) also accepts time
param, which says, parent thread
will wait either till child
thread is completed or time
mentioned is passed (min).

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

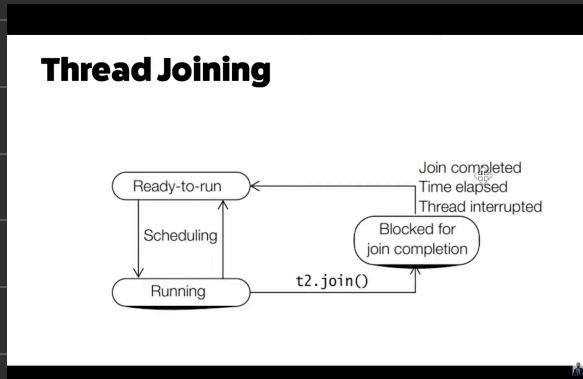
```

71 // ThreadJoiner.java
72 // Main thread prints "Popped: " + stack.pop();
73 // System.out.println("Popped: " + stack.pop());
74 // ...
75 // Thread thread = new Thread(() -> {
76 //     try {
77 //         Thread.sleep(1);
78 //         for(int i = 10000; i > 0; i--) {
79 //             stack.push(i);
80 //             if(i % 1000 == 0) {
81 //                 o.printStack();
82 //             }
83 //         }
84 //     } catch(InterruptedException e) {
85 //         e.printStackTrace();
86 //     }
87 // });
88 // thread.start();
89 // while(true) {
90 //     if(Thread.State.DORMANT == thread.getState()) {
91 //         if(state == Thread.State.TERMINATED) break;
92 //     }
93 //     Thread thread2 = new Thread(() -> {
94 //         System.out.println("Thread: " + currentThread());
95 //     });
96 //     thread2.start();
97 //     try {
98 //         thread.join();
99 //     } catch(InterruptedException e) {
100 //         e.printStackTrace();
101 //     }
102 //     o.printStack();
103 // }
104 // System.out.println("main is exiting");

```

1:46:44 / 58:09 • Thread Joining

without `join()` method,
main thread continues to
execute, whilst child thread is
being executed.



Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions

Thread Priorities :

- Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled.
- The thread scheduler can use thread priorities to determine which thread gets to run.

1:48:26 / 58:09 • Thread Priority >

Thread Priorities :

- Priorities are integer values from 1 (lowest priority given by the constant Thread.MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).
- A thread inherits the priority of its parent thread.
- The priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class.
- The setPriority() method is an advisory method, meaning that it provides a hint from the program to the JVM, which the JVM is in no way obliged to honor.

thread.getPriority();

thread.setPriority(6);

Thread Scheduler : *→ to handle threads of varying priorities.*

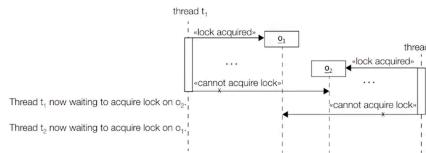
- Schedulers in JVM implementations usually employ one of the two following strategies
 1. Preemptive scheduling.
 2. Time-Sliced or Round-Robin scheduling.

Thread schedulers are platform dependent, so, it's unpredictable, how it'll behave on a platform.

Deadlocks :

- A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.
- Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state.
- The threads are said to be deadlocked.

DEADLOCKS



```
package org.nik;

public class Deadlock {
    public static void main(String[] args) {
        System.out.println("main is starting");

        Object lockOne = new Object();
        Object lockTwo = new Object();

        Thread t1 = new Thread(() -> {
            synchronized (lockOne) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                synchronized (lockTwo) {
                    System.out.println("lock acquired");
                }
            }
        }, "t1");

        Thread t2 = new Thread(() -> {
            synchronized (lockTwo) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                synchronized (lockOne) {
                    System.out.println("lock acquired");
                }
            }
        }, "t2");

        t1.start();
        t2.start();
        System.out.println("main is exiting ");
    }
}
```

```
multithreading-java [:Deadlock.main()]: Building: 'De... 8 sec
  :compileJava 33 ms
  :processResources
  :classes
  :Deadlock.main() 8 sec
```

Code for Deadlock.

Reentrant Lock

```
package org.nik;

import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample { new *
    private final ReentrantLock lock; 3 usages
    private int count; 3 usages

    public ReentrantLockExample() { 1 usage new *
        lock = new ReentrantLock();
        count = 0;
    }

    public void increment() { 1 usage new *
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) { new *
        ReentrantLockExample example = new ReentrantLockExample();
        example.increment();
        System.out.println(example.count);
    }
}
```

```
package org.nik;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockWithTimeoutExample { new *
    private final ReentrantLock lock; 3 usages
    int count; 2 usages

    public ReentrantLockWithTimeoutExample() { 1 usage new *
        lock = new ReentrantLock();
        count = 0;
    }

    public boolean increment() throws InterruptedException { 1 usage new *
        boolean isLockAcquired = lock.tryLock( 1, TimeUnit.SECONDS );
        if (isLockAcquired) {
            try {
                count++;
                Thread.sleep( millis: 5000 );
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            } finally {
                lock.unlock();
            }
            return true;
        }
        return false;
    }

    public static void main(String[] args) { new *
        ReentrantLockWithTimeoutExample example = new ReentrantLockWithTimeoutExample();

        Runnable runnable = () -> {
            try {
                System.out.println(example.increment());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        };

        Thread t1 = new Thread(runnable, name: "t1");
        Thread t2 = new Thread(runnable, name: "t2");
        t1.start();
        t2.start();
    }
}
```

Example with timeouts

```
> Task :ReentrantLockWithTimeoutExample.main()
false
true
```

```
package org.nik;

import java.util.HashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReentrantReadWriteLockHashMapExample { no usages new*
    private final ReentrantReadWriteLock lock; 3 usages
    private final HashMap<Integer, Integer> map; 5 usages
    private final ReentrantReadWriteLock.WriteLock writeLock; 5 usages
    private final ReentrantReadWriteLock.ReadLock readLock; 5 usages

    public ReentrantReadWriteLockHashMapExample() { no usages new*
        lock = new ReentrantReadWriteLock();
        map = new HashMap<>();
        writeLock = lock.writeLock();
        readLock = lock.readLock();
    }

    // If no threads are reading or writing, only one thread can acquire the write lock.
    public void put(int key, int value) { no usages new*
        try {
            writeLock.lock();
            map.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }

    // If no threads are reading or writing, only one thread can acquire the write lock.
    public int remove(int key) { no usages new*
        try {
            writeLock.lock();
            return map.remove(key);
        } finally {
            writeLock.unlock();
        }
    }

    // multiple threads can read the data if no write is happening
    public int get(int key) { no usages new*
        try {
            readLock.lock();
            return map.get(key);
        } finally {
            readLock.unlock();
        }
    }

    // multiple threads can read the data if no write is happening
    public boolean containsKey(int key) { no usages new*
        try {
            readLock.lock();
            return map.containsKey(key);
        } finally {
            readLock.unlock();
        }
    }
}
```

ReadWriteLock

Conditions

```
package org.nik;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class SafeStackWithCondition { new *
    private final int[] stack; 5 usages
    private int idx; 7 usages
    private final ReentrantLock lock; 7 usages
    private final Condition stackFullCondition; 3 usages
    private final Condition stackEmptyCondition; 3 usages

    public SafeStackWithCondition(int capacity) { 1 usage new +
        stack = new int[capacity];
        idx = -1;
        lock = new ReentrantLock();
        stackFullCondition = lock.newCondition();
        stackEmptyCondition = lock.newCondition();
    }

    public void push(int value) { 1 usage new *
        try {
            lock.lock();
            while (idx == stack.length - 1) {
                stackFullCondition.await();
            }
            stack[++idx] = value;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            stackEmptyCondition.signalAll();
            lock.unlock();
        }
    }

    public int pop() { 1 usage new *
        try {
            lock.lock();
            while (idx == -1) {
                stackEmptyCondition.await();
            }
            int value = stack[idx];
            stack[idx] = Integer.MIN_VALUE;
            idx--;
            return value;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
            return Integer.MIN_VALUE;
        } finally {
            stackFullCondition.signalAll();
            lock.unlock();
        }
    }
}
```

```
public static void main(String[] args) { new +
    SafeStackWithCondition stack = new SafeStackWithCondition( capacity: 10);

    Thread threadPusher = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            stack.push(i);
        }
    }, "pusher");

    Thread threadPopper = new Thread(() -> {
        for (int i = 0; i < 100; i++) {
            System.out.println(stack.pop());
        }
    }, "popper");

    threadPusher.start();
    threadPopper.start();
}
```

9
8
7
6
5
4
3
2
1
0
14
20
21
22
23
24

Response

ackWithCondition

ThreadPool Executor

```
package org.nik;

import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class ThreadPoolExecutorExample { new *
    public static void main(String[] args) { new *
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool( nThreads: 2);
        Runnable runnable = () -> {
            try {
                Thread.sleep( millis: 3000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        };
        executor.execute(runnable);
        executor.execute(runnable);
        executor.execute(runnable);

        if (executor.getPoolSize() == 2) {
            System.out.println("pool size is: " + executor.getPoolSize());
        }

        if (executor.getQueue().size() == 1) {
            System.out.println("queue size is: " + executor.getQueue().size());
        }
    executor.shutdown();
}
}
```

```
> Task :ThreadPoolExecutorExample.main()
pool size is: 2
queue size is: 1
```

ThreadPool with for loop

```
package org.nik;

import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class ThreadPoolExecutorForLoopExample { new *
    public static void main(String[] args) { new *
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool( nThreads: 5);
        Runnable runnable = () -> {
            try {
                System.out.println("started thread: " + Thread.currentThread().getName());
                Thread.sleep( millis: 3000);
                System.out.println("completed thread: " + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        };
        for (int i = 0; i < 10; i++) {
            executor.execute(runnable);
        }
    executor.shutdown();
}
}
```

```
> Task :ThreadPoolExecutorForLoopExample.main()
started thread: pool-1-thread-3
started thread: pool-1-thread-1
started thread: pool-1-thread-2
started thread: pool-1-thread-4
started thread: pool-1-thread-5
completed thread: pool-1-thread-1
started thread: pool-1-thread-1
completed thread: pool-1-thread-4
started thread: pool-1-thread-4
completed thread: pool-1-thread-5
started thread: pool-1-thread-5
completed thread: pool-1-thread-3
started thread: pool-1-thread-3
completed thread: pool-1-thread-2
started thread: pool-1-thread-2
completed thread: pool-1-thread-1
completed thread: pool-1-thread-4
completed thread: pool-1-thread-5
completed thread: pool-1-thread-3
completed thread: pool-1-thread-2
```