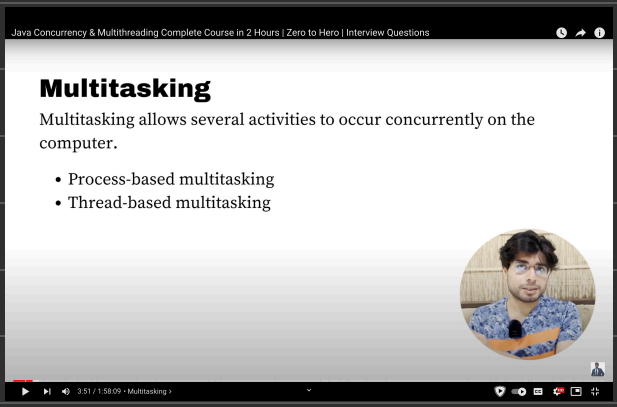
Multithreading in Java



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 ex
  between processes.
- The cost of communication between threads is relative

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 :

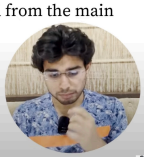
- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).
- They also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

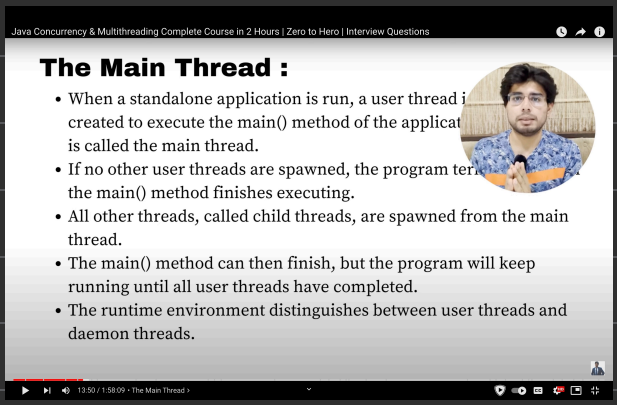
Threads :

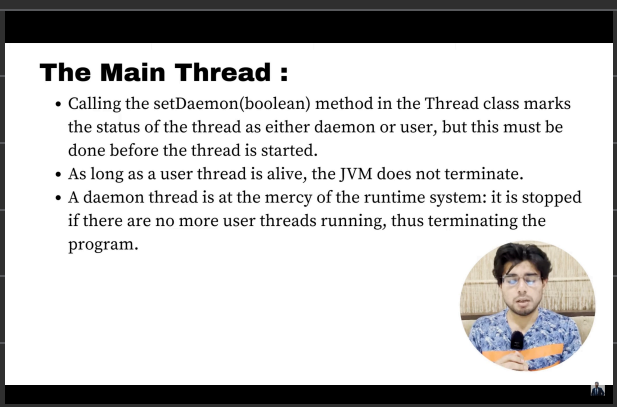
- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).
- They also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

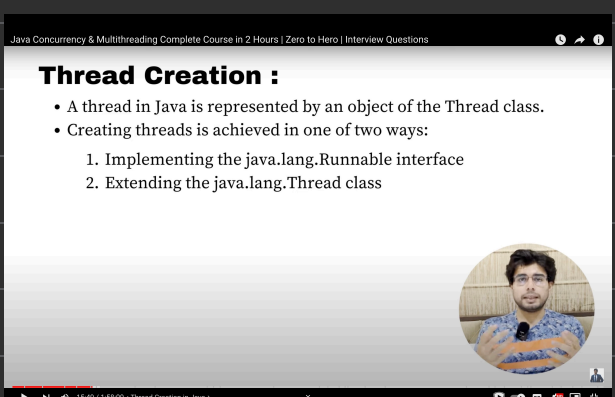
The Main Thread :

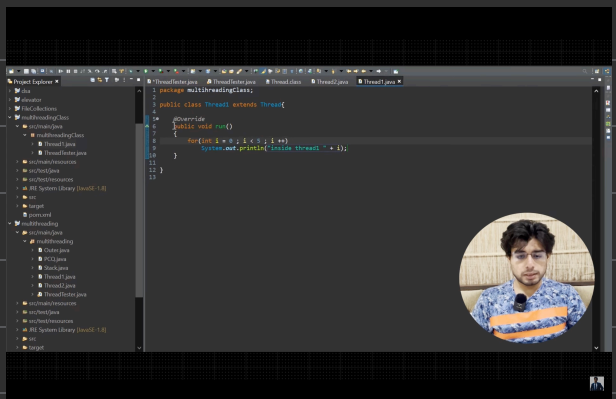
- When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the main thread.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.
- All other threads, called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

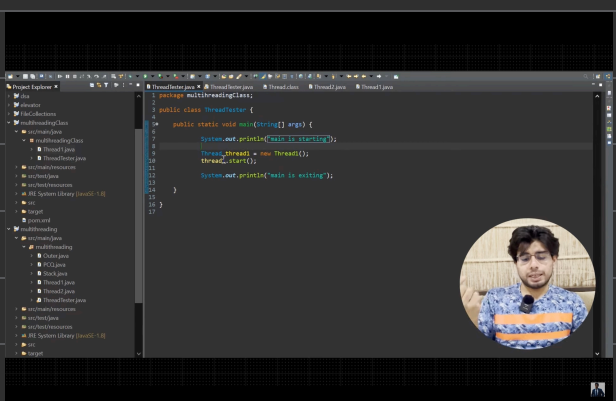


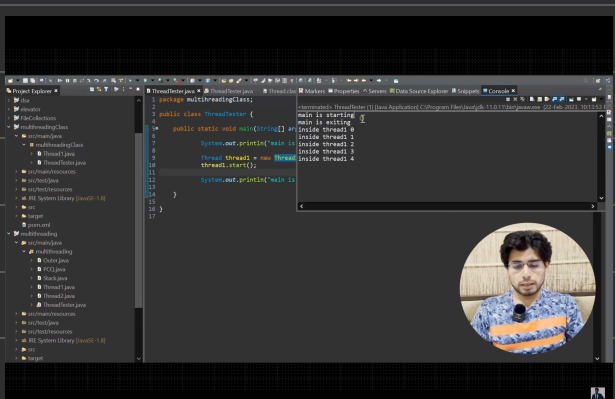

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions
The Main Thread :
• When a standalone application is run, a user thread i
created to execute the main() method of the applicat
is called the main thread.
• If no other user threads are spawned, the program ter
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.
13:50 / 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.


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 ›

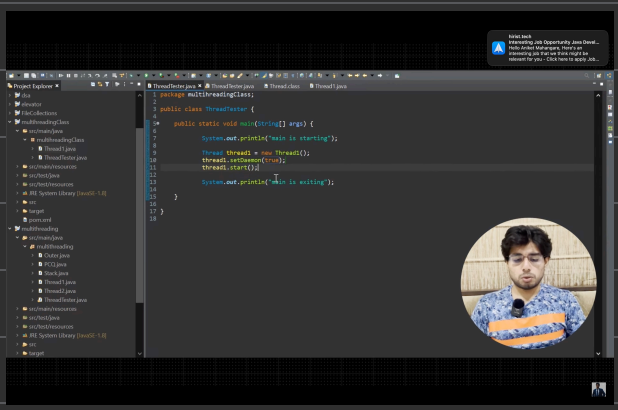


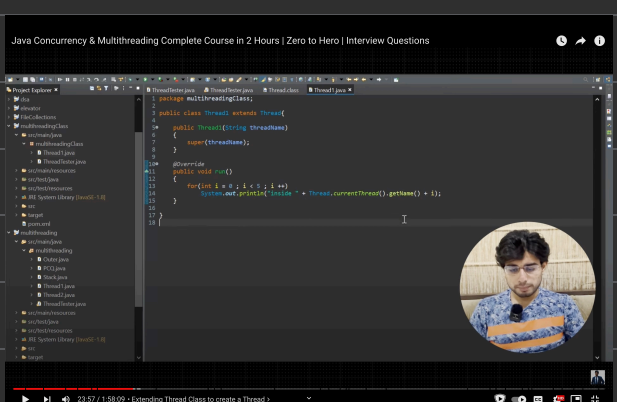




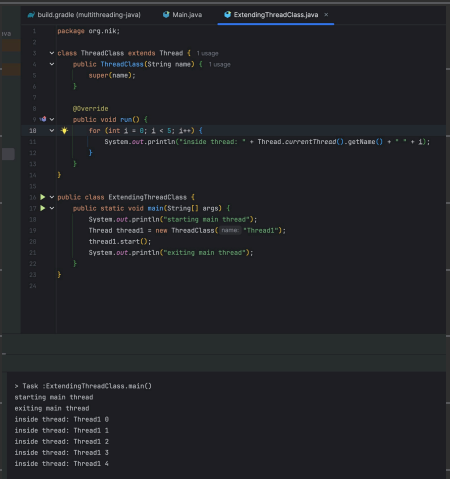
no order guaranteed.

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.

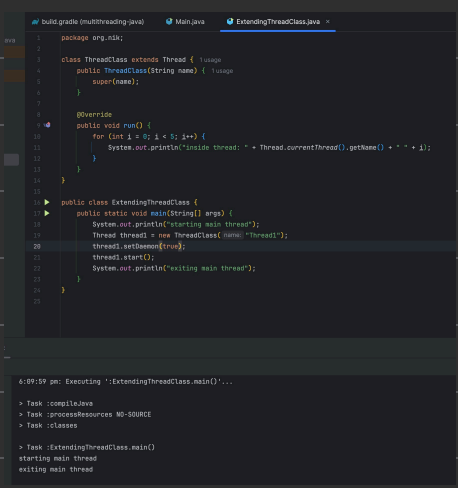




Extending Thread class.



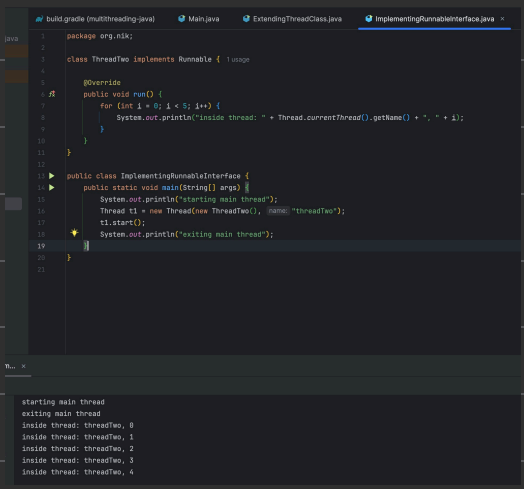
User Thread

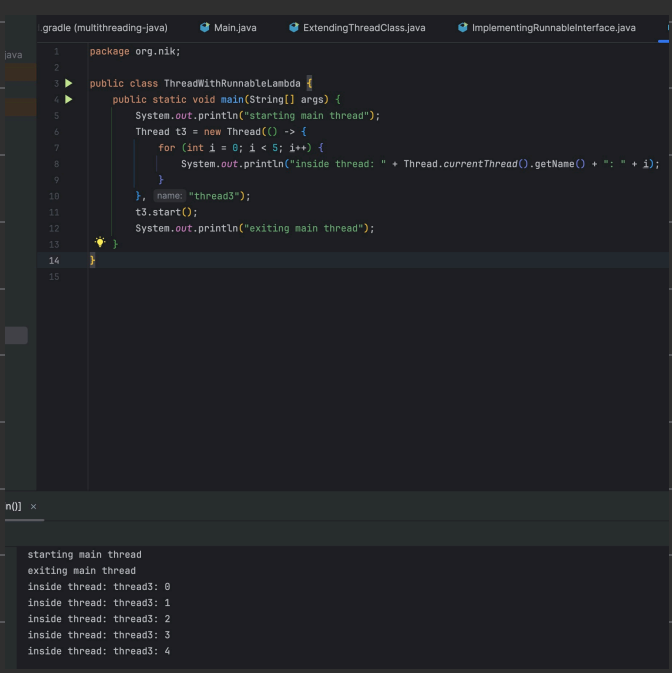


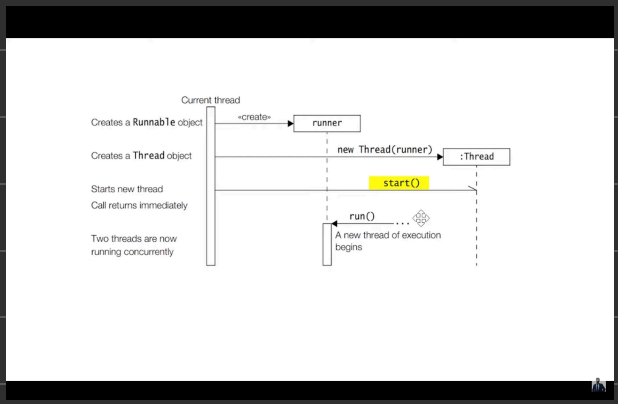
Daemon Thread

Implementing Runnable Interface:

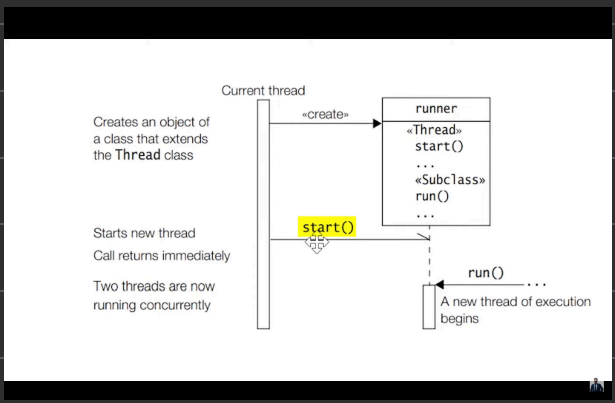
Better than extending as Java supports to extend only one class.



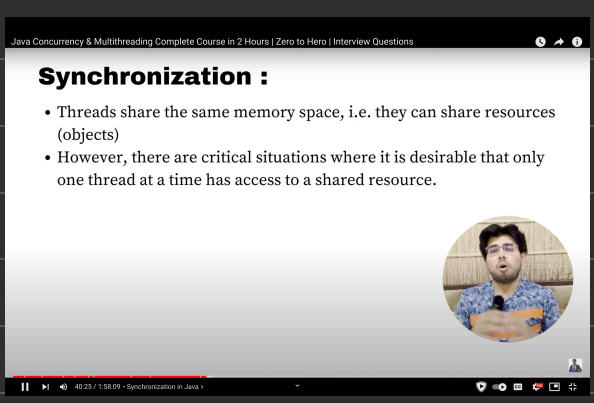




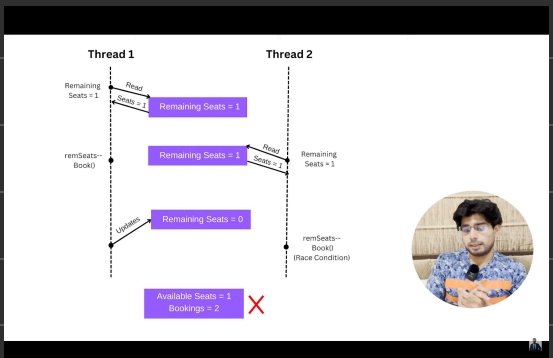
Implementing Runnable Interface.



Extending Thread Class



Race Conditions




```java
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

```java
    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");
    }
}
```

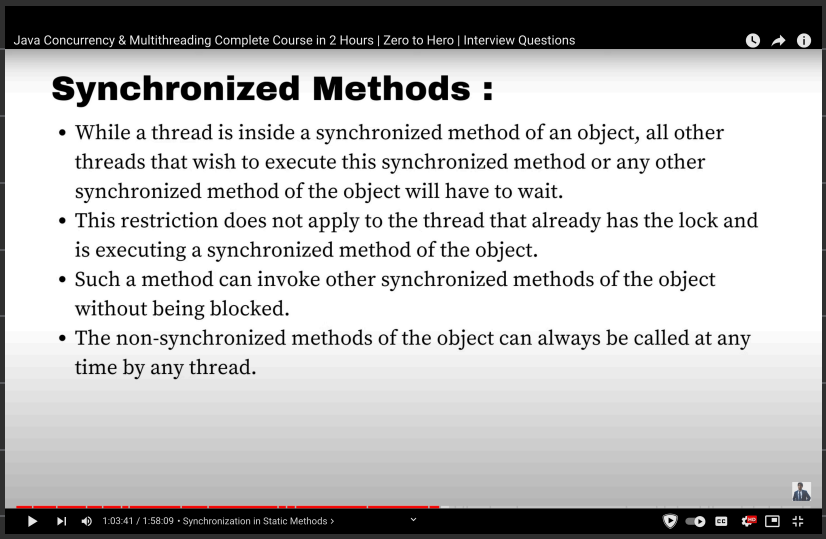
Testing Thread safe stack class.

```java
package singleton;

public class TVSet {

    private static volatile TVSet tvSetInstance = null;

    private TVSet() {
        System.out.println("TV Set instantiated");
    }

    public static TVSet getTVSetInstance() {
        if(tvSetInstance == null) { // optimisation
            synchronized(TVSet.class) { //t2
                if(tvSetInstance == null) // double checking
                    tvSetInstance = new TVSet();
            }
        }
        // heavy work done here
        return tvSetInstance;
    }
}

// time 0 - t1 , t2
// time 5 - t5 , t6 , t7.....

```

Singleton pattern with volatile keyword & synchronized block.

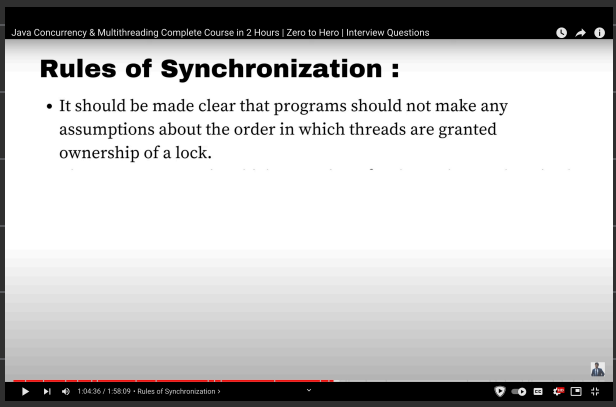
Notice that, we are using TVSet.class here, as a lock.

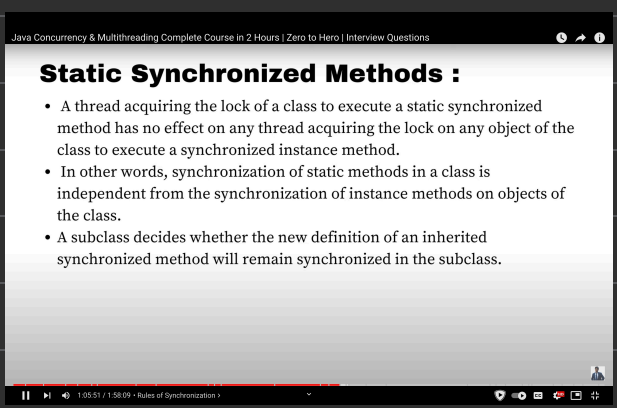
for static method, we can make a method **synchronized** which does the same

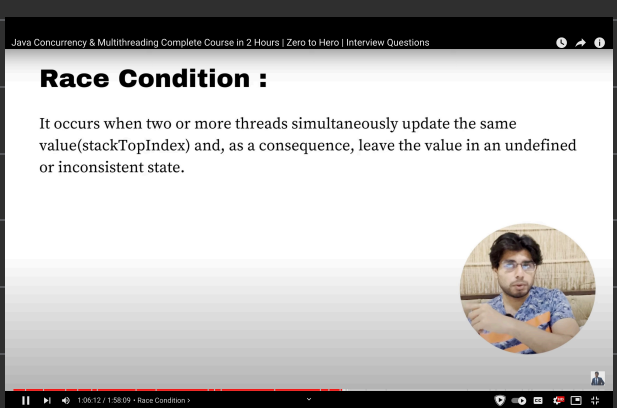


Rules of Synchronization :

- 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.

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

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)

1:06:52 / 1:58:09 • Race Condition ›

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

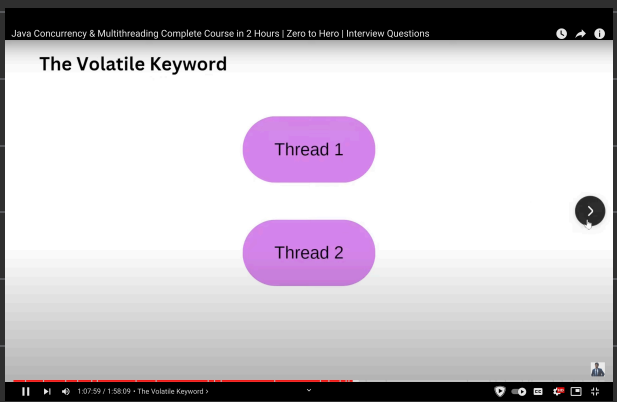
Thread safety :

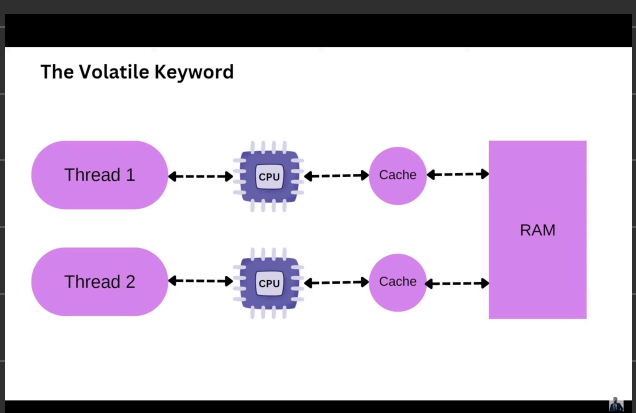
It's the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads. Eg StringBuffer.

1:07:12 / 1:58:09 • Thread Safety ›

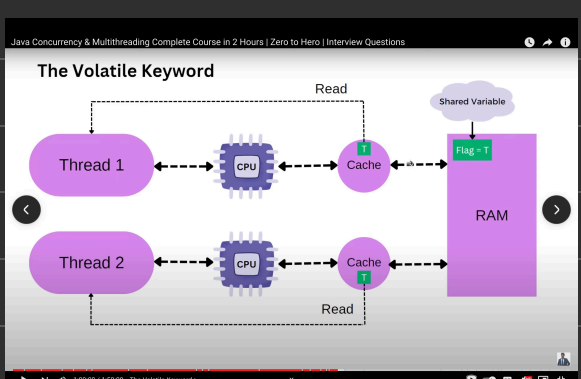
- StringBuffer is Thread Safe
- StringBuilder is not ThreadSafe.

Volatile keyword

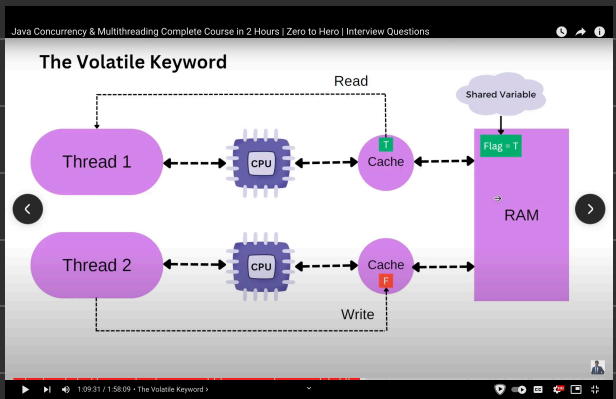




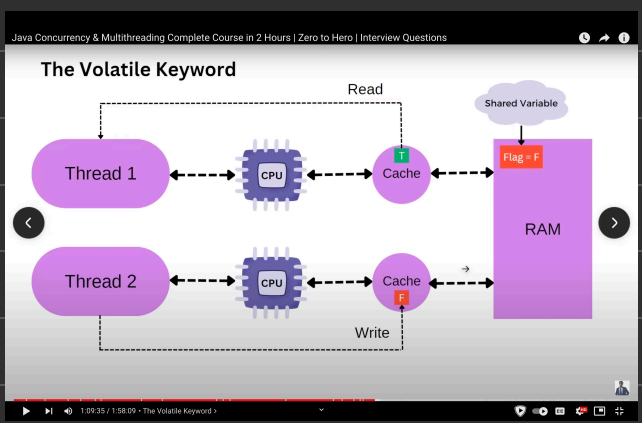
Threads store data in cache at times, as it's more efficient



Threads read value of shared variable from cache

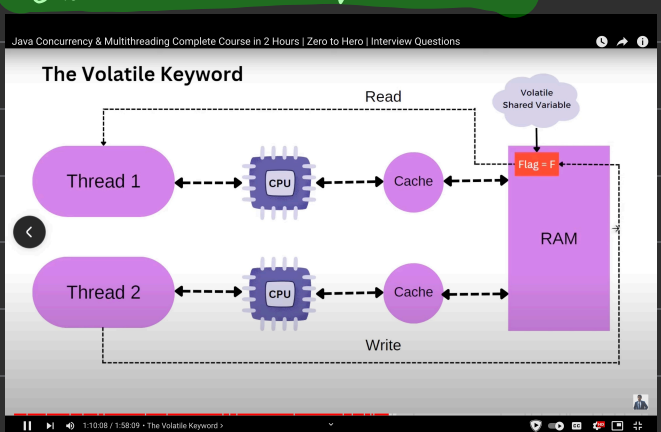


Thread 2 updates the value to false, which initially gets updated in it's cache. → T1 reads value I

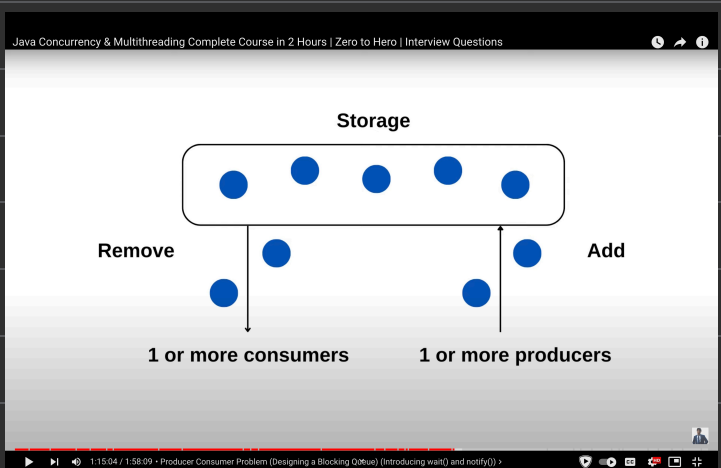


After some time, the write is reflected in main memory. → T1 reads value I

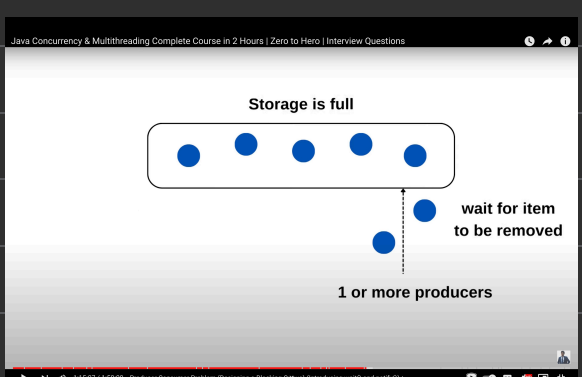
Should be used in Singleton Pattern



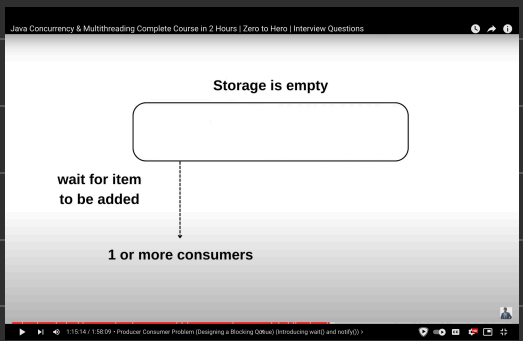
Volatile keyword ensures that the shared variable is stored only in RAM & not cached, which makes reads & writes consistent.



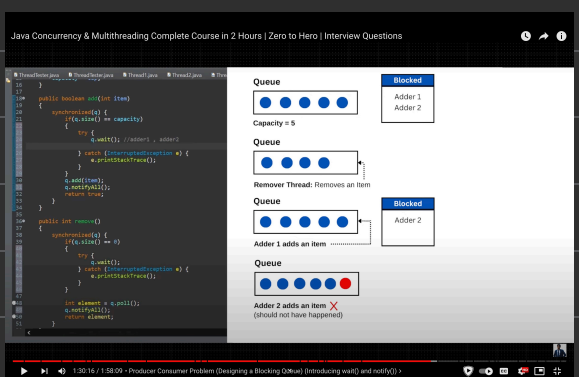
Publishers are trying to publish messages to queue, whilst consumers are trying to consume messages from queue.



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**

```java
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;
    }
}
```

Solution:

Use **while**

```java
package org.nik;

import java.util.LinkedList;

public class BlockingQueue {
    private final LinkedList<Integer> queue;  11 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;
        }
    }
```

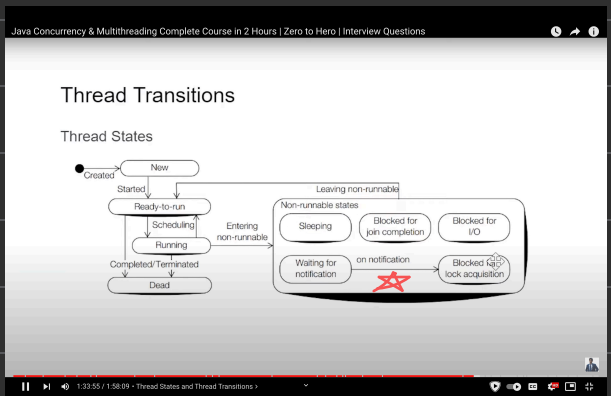
Thread Safe blocking queue.

```java
    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.



Sleeping thread doesn't release lock

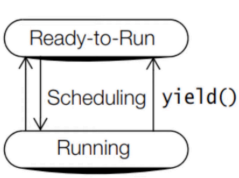
Thread States

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 {
     Thread.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;
}
```

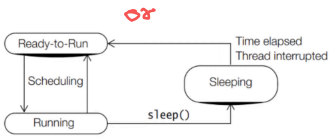
```
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
RUNNABLE
RUNNABLE
RUNNABLE
TERMINATED
```

Running and Yielding :

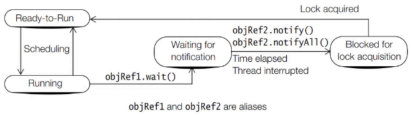


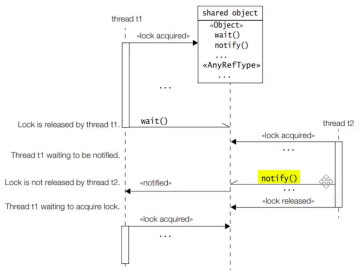
This is just an advice to the JVM, and it doesn't guarantee that the thread gets into the ready to run state immediately

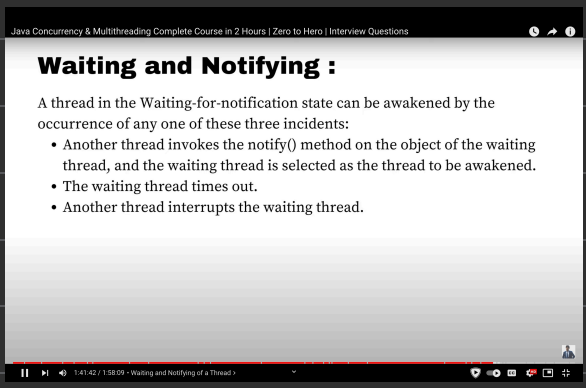
Sleeping and Waking Up :

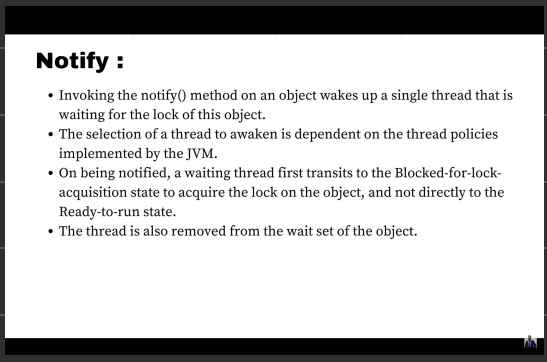


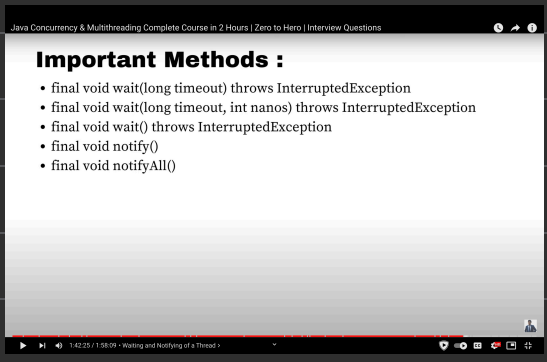
Waiting and Notifying :

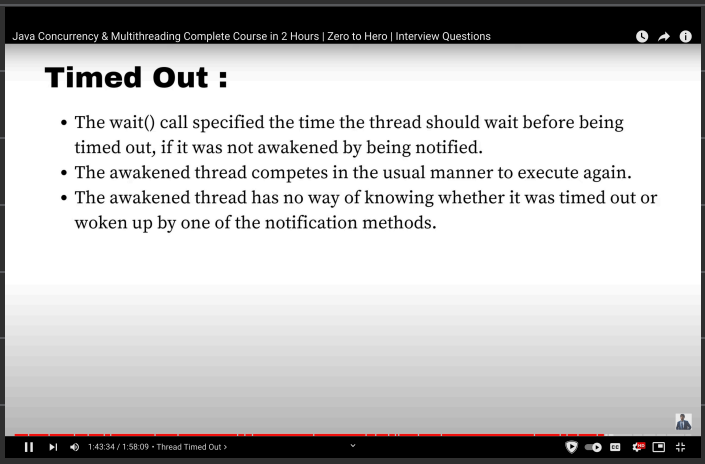




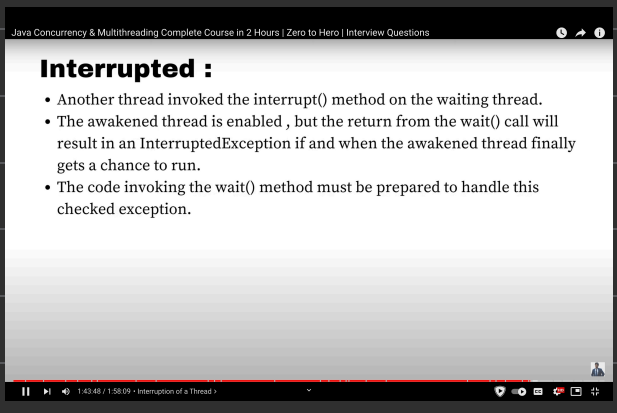

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions
Waiting and Notifying :
A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:
• Another thread invokes the notify() method on the object of 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.
1:41:42 / 1:58:09 • Waiting and Notifying of a 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.

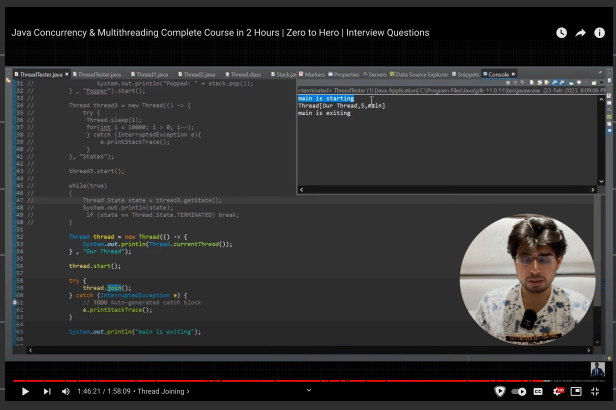

Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero | Interview Questions
Important Methods :
• final void wait(long timeout) throws InterruptedException
• final void wait(long timeout, int nanos) throws InterruptedException
• final void wait() throws InterruptedException
• final void notify()
• final void notifyAll()
1:42:25 / 1:58:09 • Waiting and Notifying of a Thread >



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



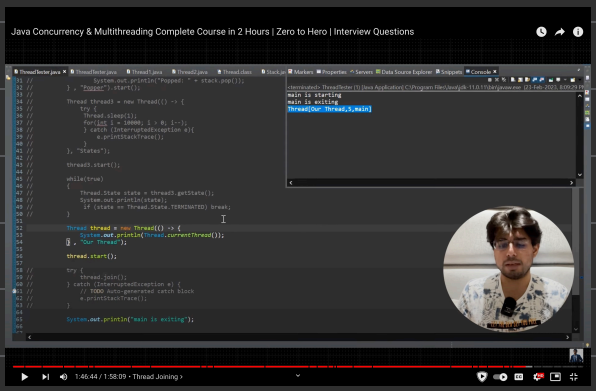
In Interrupted, a thread knows it got interrupted.



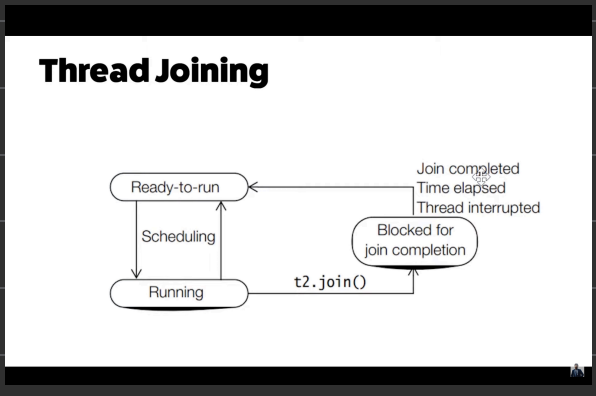
join() waits for the child thread to get completed.

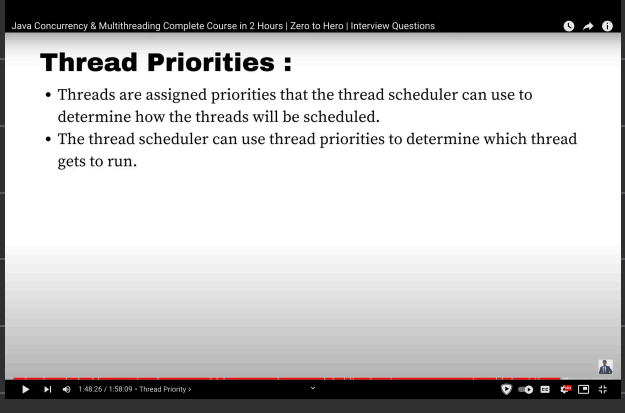
↓

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



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





Thread Priorities :

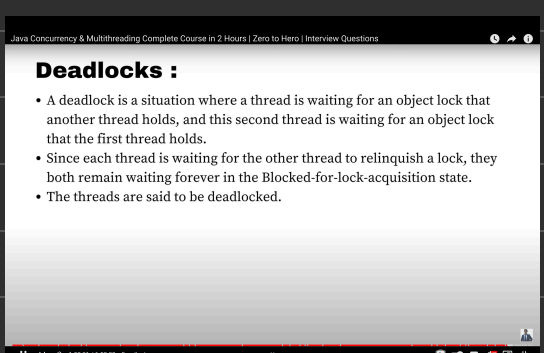
- Priorities are integer values from 1 (lowest priority given by the constant Thread. MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).
- A thread inherits the priority of its parent thread.
- The priority of a thread can 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.

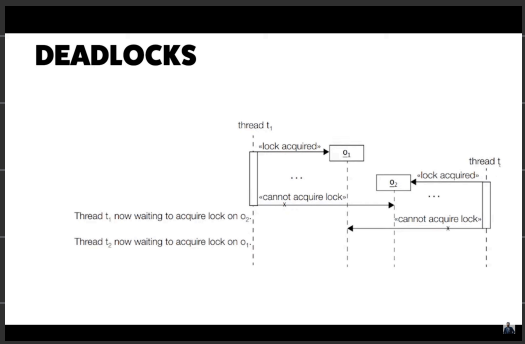
```
thread.getPriority();
thread.setPriority(6);
```

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.






```java
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( millis: 5000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                synchronized (lockTwo) {
                    System.out.println("lock acquired");
                }
            }
        }, name: "t1");

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

```java
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);
    }
}
```

```java
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( timeout: 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
```

ReadWriteLock

```java
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();
        }
    }
}
```

Conditions

```java
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();
        }
    }
```

```java
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);
        }
    }, name: "pusher");

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

```java
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

```java
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
```