

Thread :

- `name`
- `priority`
- `daemon = false`

A daemon thread, often simply referred to as a "daemon," is a type of thread in a computer program that runs in the background, providing support to the main application or other threads.

Daemon threads are designed to perform tasks that don't require direct user interaction and can be safely terminated when the main application or other non-daemon threads have finished executing

- `stillborn = false`
- `Runnable target`
- `ThreadGroup group`
- `ClassLoader classLoader`
- `AccessControlContext accessControlContext`
- `For anonymous threads : threadInitNumber`
- `ThreadLocal.ThreadLocalMap threadLocals = null;`
- `ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;`

- **private final long stackSize;**
 - * The requested stack size for this thread, or 0 if the creator did not specify a stack size. It is up to the VM to do whatever it likes with this number; some VMs will ignore it.

- **private long nativeParkEventPointer;**

/*
* JVM-private state that persists after native thread termination.
*/

- **threadId**

- **ThreadSequenceNumber**

- **volatile int threadStatus**

- **Min priority = 1 , norm priority = 5 , max = 10**

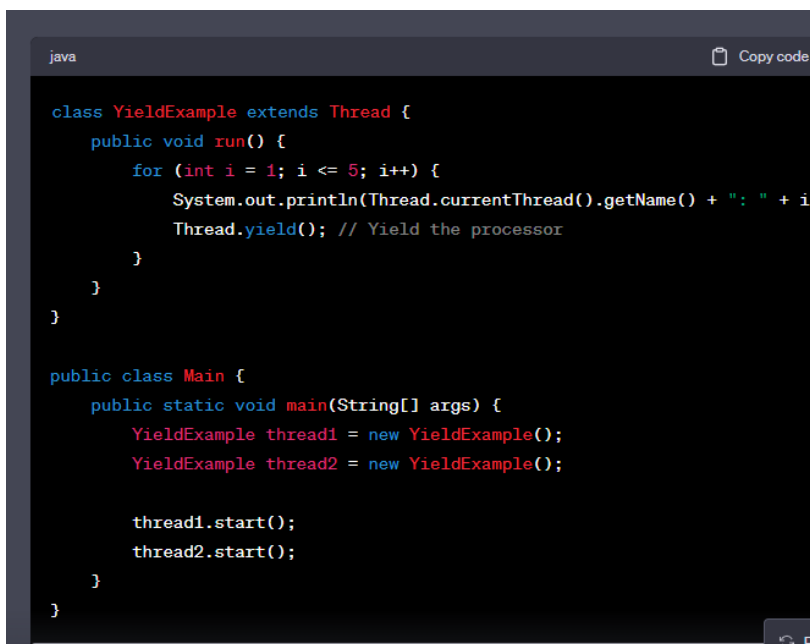
- **public static native Thread currentThread();**

Get the current active thread

Yield method :

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

- A hint to the scheduler that the current thread is willing to **yield its current use of a processor**. The scheduler is free to **ignore** this hint.
- **Yield** is a **heuristic attempt** to improve relative progression between threads that would otherwise **over-utilise** a CPU.
- Its use should be combined with detailed **profiling and benchmarking** to ensure that it actually has the desired effect.
- It is **rarely appropriate** to use this method. It may be useful for debugging or testing purposes, where **it may help to reproduce bugs due to race conditions**.
- It may also be useful when **designing concurrency control constructs** such as the ones in the `java.util.concurrent.locks` package.



```
java Copy code  
  
class YieldExample extends Thread {  
    public void run() {  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(Thread.currentThread().getName() + ": " + i);  
            Thread.yield(); // Yield the processor  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        YieldExample thread1 = new YieldExample();  
        YieldExample thread2 = new YieldExample();  
  
        thread1.start();  
        thread2.start();  
    }  
}
```

In this example, two threads (thread1 and thread2) are created. Each thread runs a loop and prints its name along with the loop variable. The Thread.yield() method is **called within the loop to give a hint that the thread is willing to yield the processor.** You might observe that both threads will likely interleave and share the processor.

Example :

1. Basic Example :

```
public class YieldMethod extends Thread{
    public void run()
    {
        for(int i = 0 ; i < 5 ; ++i)
        {
            System.out.println(
Thread.currentThread().getName() + " : Yield Method
Check" + i);
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        YieldMethod th1 = new YieldMethod();
        YieldMethod th2 = new YieldMethod();
        th1.setName("thread1 ");
        th2.setName("thread2");
        th1.start();
        th2.start();
    }
}
```

output:

```
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
```

2. Priority Based :

```
public class PriorityYieldExample extends Thread{
    private int priority;

    public PriorityYieldExample(String name, int
priority) {
        super(name);
        this.priority = priority;
    }

    public void run() {
        Thread.currentThread().setPriority(priority);
        for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + "
: " + i);

            Thread.yield(); // Yield the processor
        }
    }
}
```

```
        public static void main(String[] args) {
            PriorityYieldExample highPriority = new
PriorityYieldExample("High Priority",
Thread.MAX_PRIORITY);
            PriorityYieldExample lowPriority = new
PriorityYieldExample("Low Priority",
Thread.MIN_PRIORITY);
            highPriority.setName("Thread1");
            lowPriority.setName("Thread2");
            highPriority.start();
            lowPriority.start();
        }
    }
}
```

Output :

```
Thread2 : 1
Thread2 : 2
Thread2 : 3
Thread2 : 4
Thread1 : 1
Thread1 : 2
Thread1 : 3
Thread1 : 4
Thread2 : 5
Thread1 : 5
```

Sleep :

public static native void sleep(long millis) throws
InterruptedException;

Causes the currently executing thread to sleep
(temporarily cease execution) for the specified number
of milliseconds, subject to the precision and accuracy
of **system timers and schedulers**.

The **thread does not lose ownership of any monitors**.

Examples :

1. Basic :

```
for(int i = 0 ; i < 10 ; ++i)
{
    System.out.println(" printing  iteration "
+ i);

    try {
        System.out.println("Thread Sleeping ");
        Thread.sleep(1999) ;
        System.out.println("Thread Awaken after
2 seconds ");
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

output :

printing iteration 0

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 1

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 2

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 3

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 4

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 5

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 6

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 7

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 8

Thread Sleeping

Thread Awaken after 2 seconds

printing iteration 9

Thread Sleeping

Thread Awaken after 2 seconds

2. Handling Interrupts :

```
public class InterruptHandler implements Runnable {

    @Override
    public void run() {
        try {

            System.out.println(Thread.currentThread().getName() + "
            :  " + "Task Started");

            System.out.println(Thread.currentThread().getName() + "
            :  "+ "Sleeping the thread for 5 seconds");
                Thread.sleep(5000);
                System.out.println("Task Ended");
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        public static void main(String[] args) {
            Thread thread = new Thread(new
            InterruptHandler());
            thread.start();
            thread.setName("MyThread");
            try {

                System.out.println(Thread.currentThread().getName() + "
                :  " + "Sleeping the thread for 2 seconds " );
                    Thread.sleep(2000);

                System.out.println(Thread.currentThread().getName() + "
                :  " + "Calling Interrupt");
                    thread.interrupt();
            }
        }
    }
}
```

```

        }
        catch (InterruptedException e)
        {

System.out.println(Thread.currentThread().getName() + "
:  " + "Printing Stack Trace ");
            e.printStackTrace();
        }
    }
}

```

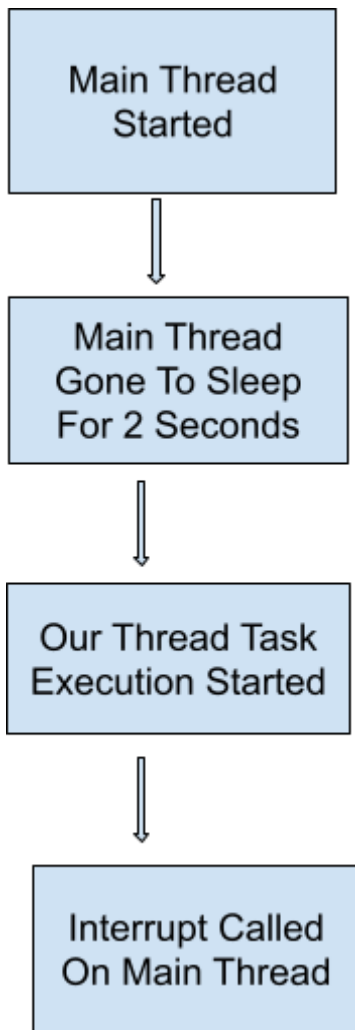
output :

```

main : Sleeping the thread for 2 seconds
MyThread : Task Started
MyThread : Sleeping the thread for 5 seconds
main : Calling Interrupt
java.lang.InterruptedException: sleep interrupted
    at java.base/java.lang.Thread.sleep(Native Method)
    at
org.example.thread.sleep.InterruptHandler.run(Interrupt
Handler.java:11)
    at java.base/java.lang.Thread.run(Thread.java:834)

Process finished with exit code 0

```



OnSpinWait() :

It is used to hint to the runtime that the current thread is willing to spin (actively wait) in a **busy-wait loop** while allowing other threads to potentially make progress.

- It's generally used in scenarios where the thread is waiting for a **specific condition to become true** but doesn't want to yield its CPU time like `Thread.yield()`, which can be more expensive in terms of performance.

-The basic idea is that `onSpinWait()` helps improve **cache locality** and reduce **memory contention**, as the thread is signaling to the system that it's actively waiting and not just idly consuming CPU resources.

Example :

```
public static void main(String[] args) {
    Thread t1 = new Thread( () -> {
        for(int i = 0 ; i < 10 ; ++i)
        {
            System.out.println("Thread 1 " + i);
            Thread.onSpinWait();
        }
    });

    Thread t2 = new Thread( () -> {
        for(int i = 0 ; i < 10 ; ++i)
        {
            System.out.println("Thread 2 " + i);
            Thread.onSpinWait();
        }
    });

    t1.start();
    t2.start();
}
```

Output :

```
Thread 2 0
Thread 1 0
Thread 1 1
Thread 1 2
Thread 1 3
Thread 1 4
Thread 1 5
Thread 1 6
Thread 1 7
Thread 1 8
Thread 1 9
Thread 2 1
Thread 2 2
Thread 2 3
Thread 2 4
Thread 2 5
Thread 2 6
Thread 2 7
Thread 2 8
Thread 2 9
```

Cache Locality :

- The principle that when a program accesses a particular memory location, it's likely to access nearby memory locations in the near future
- This is because of the way computer memory hierarchies are structured, with different levels of caches (L1, L2, L3, etc.) closer to the CPU
- Data that is accessed together tends to be stored together in these caches
- By actively **spinning and hinting to the runtime that you're waiting**, the **CPU scheduler may decide to keep the thread on the same CPU core where its cache is already loaded**. This means that the data the thread needs might still be present in the

cache, which reduces memory latency compared to if the thread was moved to a different core.

Memory Contention :

- Memory contention occurs when multiple threads or processes are competing for access to the same memory resources, such as shared variables or memory locations
- This contention can lead to inefficiencies due to delays caused by waiting for memory access
- Memory contention is a common issue in multi-threaded programs where multiple threads are accessing shared data
- When multiple threads are actively spinning, the runtime might employ techniques like delaying the progress of some threads (**backing off**) or rescheduling them in a way that reduces contention. This can prevent excessive memory contention, as threads are **actively cooperating in a more controlled manner.**

Real Life Use Cases :

1. Lock-Free Algorithm
2. Busy-Waiting for External Events
3. Low-Level Synchronization Primitives
4. Spinlocks with Short Hold Times
5. Polling for Resource Availability

of that method. A call to the `onSpinWait` method should be placed inside the spin loop.

```
class EventHandler {
    volatile boolean eventNotificationNotReceived;
    void waitForEventAndHandleIt() {
        while ( eventNotificationNotReceived ) {
            java.lang.Thread.onSpinWait();
        }
        readAndProcessEvent();
    }

    void readAndProcessEvent() {
        // Read event from some source and process it
        . . .
    }
}
```

The code above would remain correct even if the `onSpinWait` method was not called at all. However on some architectures the Java Virtual Machine may issue the processor instructions to address such code patterns in a more beneficial way

Creating Threads With Stack Size Parameter :

- On some platforms, specifying a higher value for the **stackSize** parameter may allow a thread to achieve greater recursion depth before throwing a **StackOverflowError**. Similarly, specifying a lower value may allow a greater number of threads to exist concurrently without throwing an **OutOfMemoryError** (or other internal error). The details of the relationship between the value of the **stackSize** parameter and the maximum recursion depth and concurrency level are platform-dependent. On some platforms, the value of the **stackSize** parameter may have no effect whatsoever.
- The virtual machine is free to treat the **stackSize** parameter as a suggestion. If the specified value is unreasonably low for the platform, the virtual machine may instead use some platform-specific minimum value; if the specified value is unreasonably high, the virtual machine may instead use some platform-specific maximum. Likewise, the virtual machine is free to round the specified value up or down as it sees fit (or to ignore it completely).

Start() Method :

- Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread
- The result is that two threads are running concurrently:
the **current thread**(which returns from the call to the start method) and the **other thread** (which executes its run method)
- It is never legal to start a thread more than once. In particular, **a thread may not be restarted once it has completed execution**
- This method is not invoked for the main method thread or "system" group threads created/set up by the VM.

Flow :

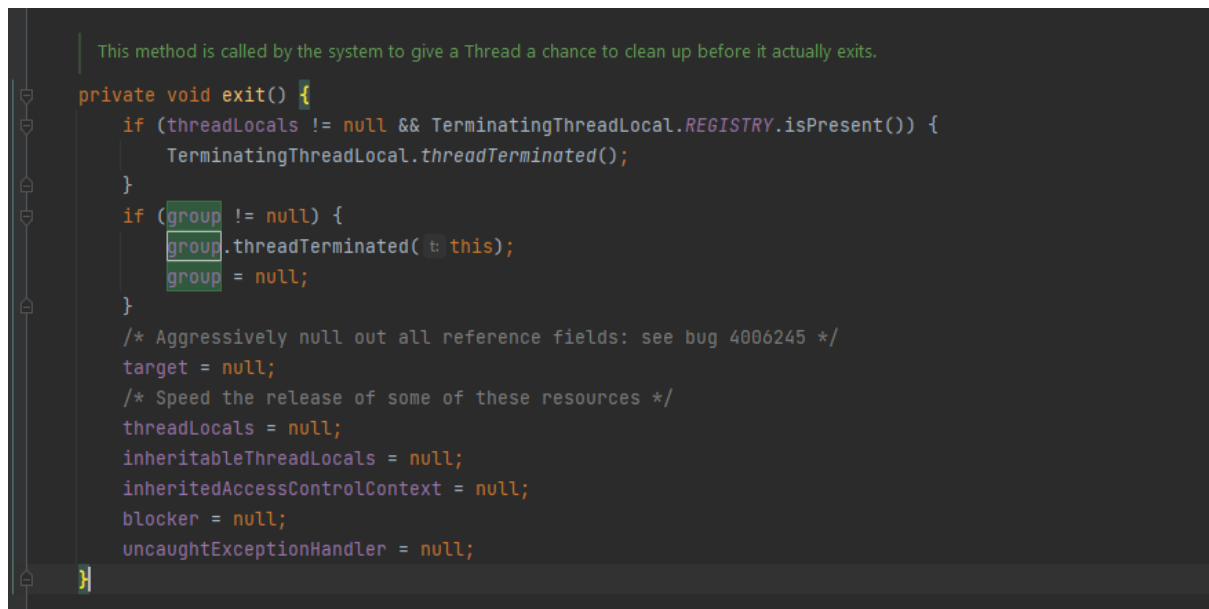
- 1.If Thread Status != 0 : Throw
IllegalThreadStateException
2. Add the Thread to the ThreadGroup

```
boolean started = false;
    try {
        start0();
        started = true;
    }
    finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        }
        catch (Throwable ignore) {
            /* do nothing. If start0 threw a
            Throwable then it will be passed up the call stack
            */
        }
    }
```

Run() Method :

```
public void run() {  
    if (target != null)  
    {  
        target.run();  
    }  
}
```

Exit () Method :



This method is called by the system to give a Thread a chance to clean up before it actually exits.

```
private void exit() {  
    if (threadLocals != null && TerminatingThreadLocal.REGISTRY.isPresent()) {  
        TerminatingThreadLocal.threadTerminated();  
    }  
    if (group != null) {  
        group.threadTerminated(this);  
        group = null;  
    }  
    /* Aggressively null out all reference fields: see bug 4006245 */  
    target = null;  
    /* Speed the release of some of these resources */  
    threadLocals = null;  
    inheritableThreadLocals = null;  
    inheritedAccessControlContext = null;  
    blocker = null;  
    uncaughtExceptionHandler = null;  
}
```

Stop() Method :

- Forces the thread to stop executing
 - The thread represented by this thread is forced to stop whatever it is doing abnormally and to throw a newly created ThreadDeath object as an exception
 - It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.
- An application should **not** normally try to catch ThreadDeath unless it must do some extraordinary cleanup operation (note ***that the throwing of ThreadDeath causes finally clauses of try statements to be executed before the thread officially dies***). If a catch clause catches a ThreadDeath object, **it is important to rethrow the object so that the thread actually dies.**

Issue with Using Stop () :

- This method is inherently unsafe. Stopping a thread with **Thread.stop** causes it to **unlock all of the monitors that it has locked** (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack).
- If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior
- Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running.
- The primary reason for deprecating the **stop()** method is that it forcefully terminates a thread **without giving it a chance to perform cleanup or**

release resources properly. This can result in a thread being stopped at an arbitrary point in its execution, leaving the **program in an unpredictable and potentially unstable state.**

- Instead of using the stop() method, it's recommended to use **cooperative mechanisms** for stopping threads, such as **using flags or signals** to indicate that a thread should terminate its execution gracefully. This allows the thread to **perform necessary cleanup operations before exiting.**

Incorrect Way :

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

```

        Thread thread = new Thread(() -> {
            int count = 0;
            while (true) {
                System.out.println("Count: " +
count);
                count++;
            }
        });

        thread.start();

        try {
            Thread.sleep(1000); // Sleep for 1
second
            thread.stop(); // Deprecated method to
stop the thread
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Correct Way :

```

public class StopThreadExample {

```

```

    static volatile boolean stopRequested = false;
// Volatile ensures visibility across threads

    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            int count = 0;
            while (!stopRequested) {
                System.out.println("Count: " +
count);
                count++;
            }
        });

        thread.start();

        try {
            Thread.sleep(1000); // Sleep for 1
second
            stopRequested = true; // Signal the
thread to stop
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

- This allows the running thread to finish its current iteration of the loop and **exit gracefully**.

- Using this cooperative mechanism for stopping threads ensures that the thread can perform cleanup operations and release resources properly before exiting
- Remember that using **volatile** ensures that changes to the stopRequested flag are **immediately visible to all threads, preventing potential visibility issues in multi-threaded environments**.

Interrupt Method () :

- Interrupts this thread

- Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown
- If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an **`InterruptedException`**.
- If this thread is blocked in an **I/O** operation upon an `InterruptibleChannel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `java.nio.channels.ClosedByInterruptException`.
- If this thread is blocked in a `java.nio.channels.Selector` then the **thread's interrupt status will be set** and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's `wakeup` method were invoked
- If none of the previous conditions hold then this thread's interrupt status will be set
- Interrupting a thread that is not alive need not have any effect.

`InterruptedException` :

- Tests whether the current thread has been interrupted

- The interrupted status of the thread is cleared by this method
- if this method were to be called **twice in succession**, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it)
- A thread interruption ignored because a thread was not alive at the time of the interrupt will be reflected by this method **returning false**

Example :

```
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        while (!Thread.interrupted()) {
            System.out.println("Working..");
        }
        System.out.println("Thread interrupted. (Interrupted
status: " + Thread.currentThread().isInterrupted() +
"");
    });

    thread.start();

    try {
        Thread.sleep(1000);
        thread.interrupt();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

output :

```
Working ..  
Working ..  
Working ..  
Working ..  
Working ..  
Thread Interrupted  
false ( status Clear Operation happens )
```

When We Print the Interrupted Boolean value after
Interrupt = False (as we clear the value)

Is Interrupted () :

- Tests whether this thread has been interrupted

- The interrupted status of the thread is unaffected by this method. (will remain true)
- A thread interruption ignored because a thread was not alive at the time of the interrupt will be reflected by this method returning false

Example :

```
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
while (!Thread.currentThread().isInterrupted()) {
    System.out.println("Working..");
}
System.out.println("Thread interrupted. (Interrupted
status: " + Thread.currentThread().isInterrupted() +
"");
    });

    thread.start();

    try {
        Thread.sleep(1000);
        thread.interrupt();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

output :

Working ..

Working ..

Working ..

Working ..

Working ..

Thread Interrupted

true (status Clear Operation doesn't happens)

Set Priority () :

- Changes the priority of this thread.

- the priority of this thread is set to the smaller of the specified newPriority and the maximum permitted priority of the thread's thread group

```
ThreadGroup.getMaxPriority()

public final void setPriority(int newPriority) {
    ThreadGroup g;
    checkAccess();
    if (newPriority > MAX_PRIORITY || newPriority < MIN_PRIORITY) {
        throw new IllegalArgumentException();
    }
    if ((g = getThreadGroup()) != null) {
        if (newPriority > g.getMaxPriority()) {
            newPriority = g.getMaxPriority();
        }
        setPriority0(priority = newPriority);
    }
}
```

Enumerate() :

- enumerate() method is used to retrieve an array of all active threads within the current thread's thread group.
- It's a convenient way to get information about the threads running within the same thread group, which is useful for debugging, monitoring, or other purposes.

Example :

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

```
Thread main = Thread.currentThread();
```

```
Thread myThread = new Thread( () -> {  
for(int i = 0 ; i < 5 ; ++i)  
{  
    System.out.println("MyThread " + i);  
    try {  
        Thread.sleep(500);  
    }  
    catch (InterruptedException e)  
    {  
        e.printStackTrace();  
    }  
}  
});
```

```
myThread.start();
```

```
Thread myThread2 = new Thread(() -> {  
for(int i = 0 ; i < 5 ; ++i)  
{    System.out.println("Thread 2 " + i);  
    try {  
        Thread.sleep(700);  
    }  
    catch (InterruptedException e)  
    {  
        e.printStackTrace();  
    }  
}  
});
```

```
myThread2.start();
```

```

Thread [] allThreads = new
Thread[Thread.activeCount()];

Thread.enumerate(allThreads);

System.out.println("Active Threads ");

for(Thread t : allThreads)
{
    if(t!=null) System.out.println(t.getName());
}

}
}

```

Output :

Active Threads

main

Monitor Ctrl-Break

Thread-0

Thread-1

Thread 2 0

MyThread 0

MyThread 1

Thread 2 1

MyThread 2

Thread 2 2

MyThread 3

MyThread 4

Thread 2 3

Thread 2 4

Join Method () :

- Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.
- join() method is used to make a calling thread wait until the thread it's invoked on completes its execution
- This method is especially useful when you need to ensure that a certain thread has finished its work before the calling thread continues
- This implementation uses a loop of this.wait calls conditioned on this.isAlive. As a thread terminates the this.notifyAll method is invoked. It is recommended that applications not use wait, notify, or notifyAll on Thread instances.

Example :

```
public class JoinMethodExample {

    public static void main(String[] args) {

        Thread thread1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread 1: " +
i);
                try {
                    Thread.sleep(1000); // Sleep for
1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```



```

        Thread thread2 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread 2: " +
i);
                try {
                    Thread.sleep(800); // Sleep for
800 milliseconds
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

thread1.start();
thread2.start();

try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

}
}

```

Output :

```
Thread 1: 1  
Thread 2: 1  
Thread 2: 2  
Thread 1: 2  
Thread 2: 3  
Thread 1: 3  
Thread 2: 4  
Thread 1: 4  
Thread 2: 5  
Thread 1: 5
```

If We Reverse the Order of Join :

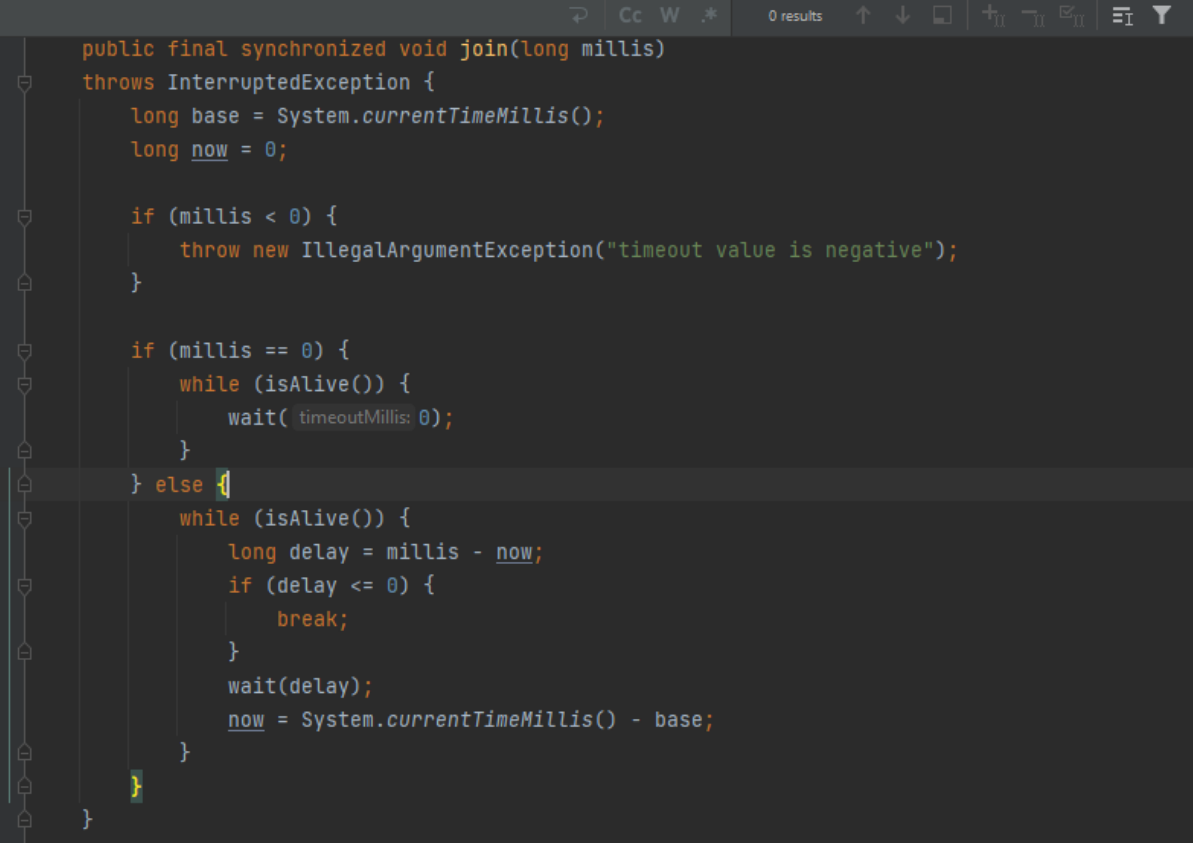
```
try {  
    thread2.join();  
    thread1.join();  
  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

Output :

```
Thread 2: 1  
Thread 1: 1
```

Thread 2: 2
Thread 1: 2
Thread 2: 3
Thread 1: 3
Thread 2: 4
Thread 1: 4
Thread 2: 5
Thread 1: 5

Implementation :

A screenshot of an IDE showing the implementation of the `join(long millis)` method in the `Thread` class. The code is written in Java and includes comments for each line. The method is public, final, and synchronized, and it throws an `InterruptedException`. It takes a `long millis` parameter. The implementation checks if `millis` is negative, zero, or positive. If negative, it throws an `IllegalArgumentException`. If zero, it calls `isAlive()` and `wait(0)`. If positive, it calculates the delay and calls `wait(delay)`. The code is as follows:

```
public final synchronized void join(long millis)
throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait( timeoutMillis: 0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

Thread States :

A thread state. A thread can be in one of the following states:

NEW

A thread that has not yet started is in this state.

RUNNABLE

Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

BLOCKED

Thread state for a thread blocked waiting for a monitor lock.

A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling `Object.wait`.

WAITING

Thread state for a waiting thread.

A thread is in the waiting state due to calling one of the following methods:

`Object.wait` with no timeout

`Thread.join` with no timeout

`LockSupport.park`

A thread in the waiting state is waiting for another thread to perform a particular action.

For example, a thread that has called `Object.wait()` on an object is waiting for another thread to call `Object.notify()` or `Object.notifyAll()` on that object.

A thread that has called `Thread.join()` is waiting for a specified thread to terminate.

TIMED_WAITING

Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

`Thread.sleep`

`Object.wait` with timeout

`Thread.join` with timeout

`LockSupport.parkNanos`

`LockSupport.parkUntil`

TERMINATED

A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

Next :

- 1. Executor**
- 2. ExecutorService**
- 3. AbstractExecutorService**
- 4. ThreadPoolExecutor**