Java Concurrency in Practice

Chapter-2: Thread Safety

Upcode Software Engineer Team

-2023-

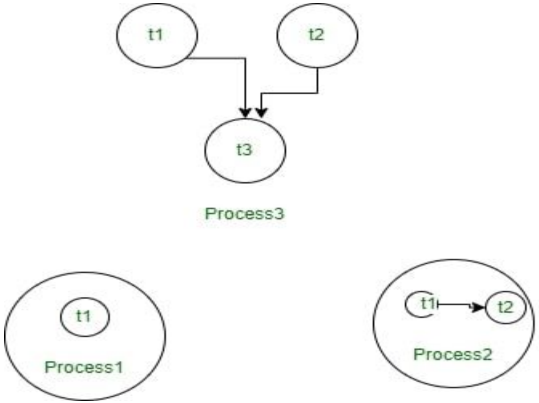
CONTENT

1. What is Thread ?
2. Thread Types
3. The Concept Of Multitasking
4. Thread methods
5. Thread Safety
6. Summary
7. Reference

What is Thread? (1/n)

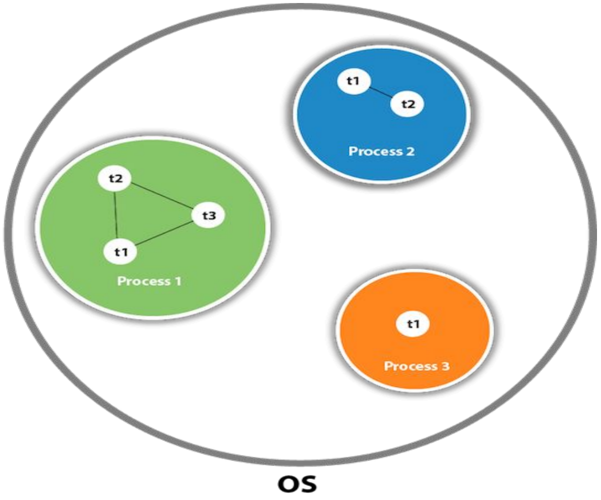
- we can define **threads as a subprocess** with lightweight with **the smallest unit of processes and also has separate paths of execution**.
- **These threads** use **shared memory** but they **act independently** hence if there is an exception in **threads that do not affect the working of other threads** despite them ***sharing the same memory.***

Threads in a Shared Memory Environment in OS



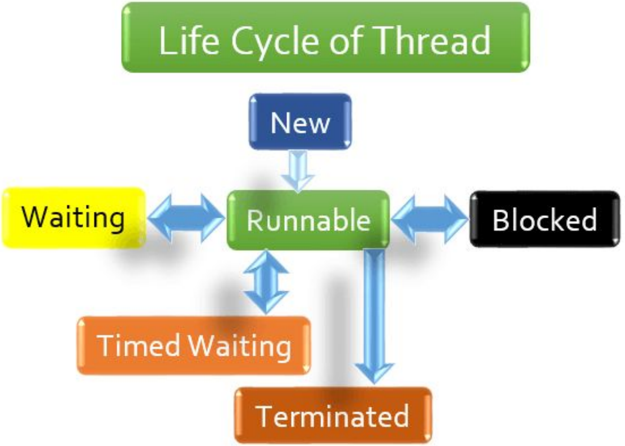
What is Thread? (2/n)

- **A thread** of execution in a program (kind of like a virtual CPU). The JVM allows an application to have **multiple threads running concurrency**.
- Each thread can execute parts of your code in parallel **with the main thread**
- Each thread has a **priority**.



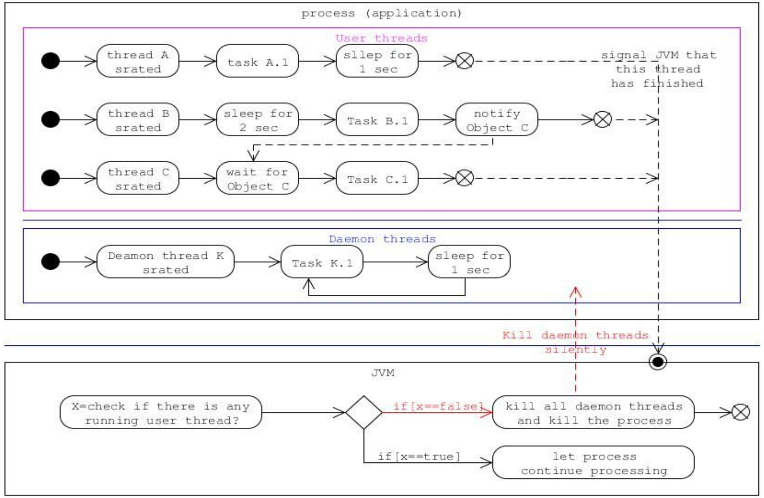
What is Thread? (3/n)

- Threads with higher priority are executed in preference compared to threads with a **lower priority**
- The Java VM continues to execute threads until either of the following occurs thread
  - The exit method of class **Runtime has been called**
  - All user threads have died



Thread Types(1/n) - Daemon and User threads

- The Thread consists of 2 types: **Daemon thread in Java is a low-priority thread** that performs background operations
- **User Threads in Java is a high priority thread that the JVM** waits till the execution is finished.

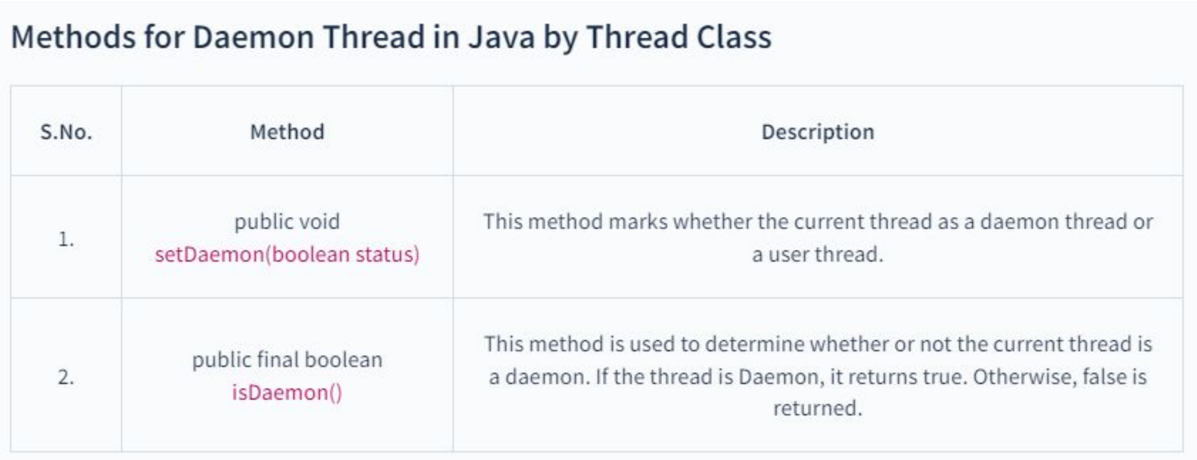


User threads

Daemon threads

Thread Types(2/n)

- Daemon thread in Java is a low-priority thread that performs background operations such as ***garbage collection, finalizer, Action Listeners, Signal dispatches, etc***.
- **Daemon thread in Java is also a service provider thread that helps the user thread.** Its life is at the mercy of user threads; when **all user threads expire, JVM immediately terminates this thread**.

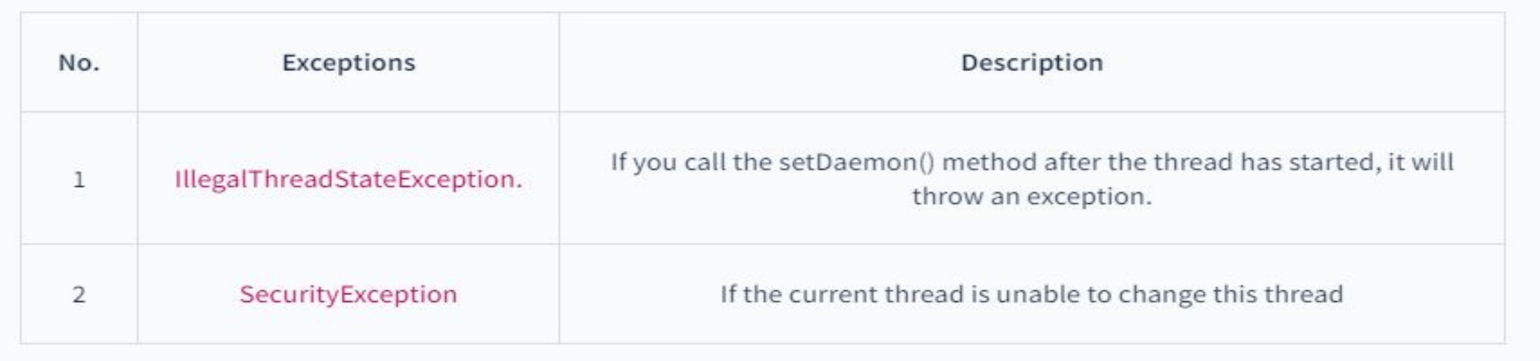


Methods for Daemon Thread in Java by Thread Class

S.No.	Method	Description
1.	public void setDaemon(boolean status)	This method marks whether the current thread as a daemon thread or a user thread.
2.	public final boolean isDaemon()	This method is used to determine whether or not the current thread is a daemon. If the thread is Daemon, it returns true. Otherwise, false is returned.

Thread Types(3/n)-Exceptions in a Daemon

- Daemon thread Exception in Java

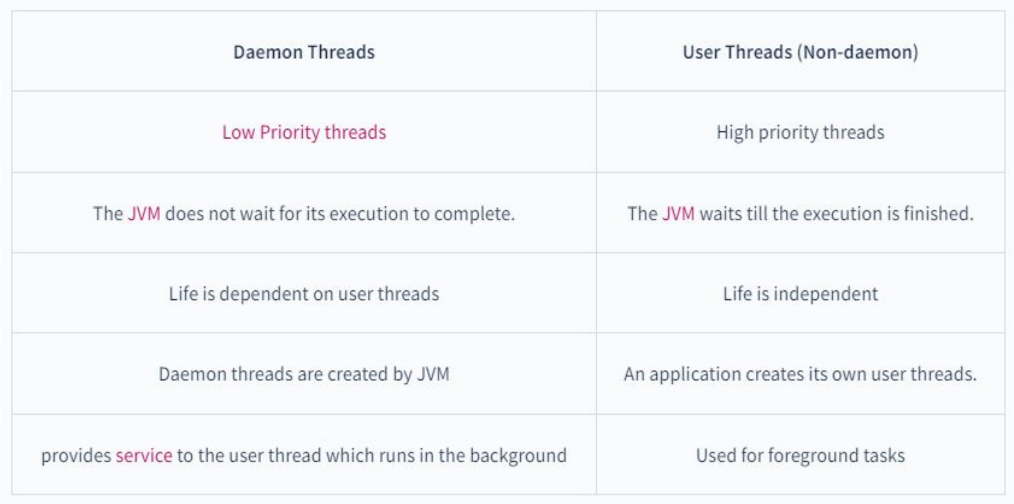


No.	Exceptions	Description
1	IllegalThreadStateException.	If you call the setDaemon() method after the thread has started, it will throw an exception.
2	SecurityException	If the current thread is unable to change this thread

```
Exception in thread "main" java.lang.IllegalThreadStateException at
java.base/java.lang.Thread.setDaemon(Thread.java:1406)at
DemoDaemonThread.main(DemoDaemonThread.java:18)
```

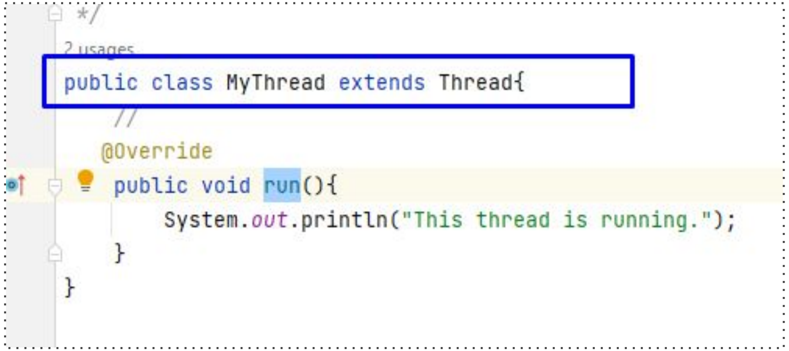
Thread Types(4/n)

- With the aid of the table below, learn more about the distinctions between Daemon and User threads
- Every user defined thread is created as non-daemon thread by default, because main thread is a non-daemon thread.



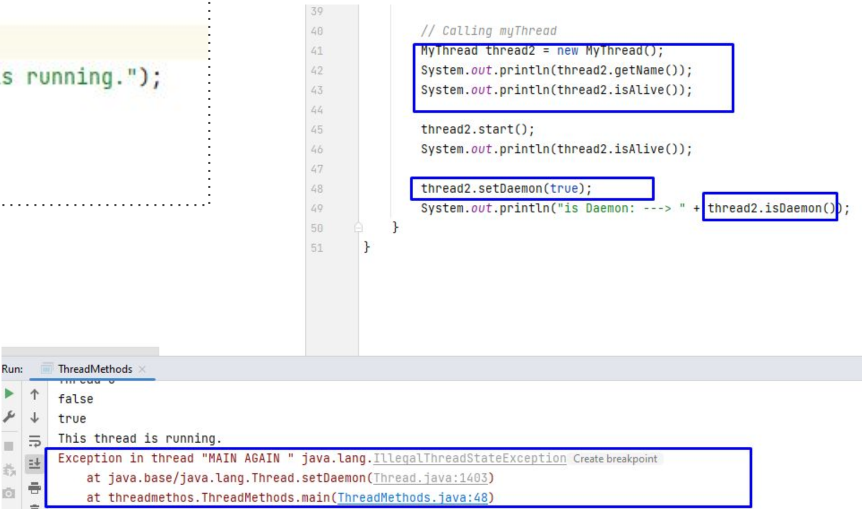
Daemon Threads	User Threads (Non-daemon)
Low Priority threads	High priority threads
The JVM does not wait for its execution to complete.	The JVM waits till the execution is finished.
Life is dependent on user threads	Life is independent
Daemon threads are created by JVM	An application creates its own user threads.
provides service to the user thread which runs in the background	Used for foreground tasks

Thread Types(4/n)- Sample code



```
*/
public class MyThread extends Thread{
    //
    @Override
    public void run(){
        System.out.println("This thread is running.");
    }
}
```

Class: MyThread



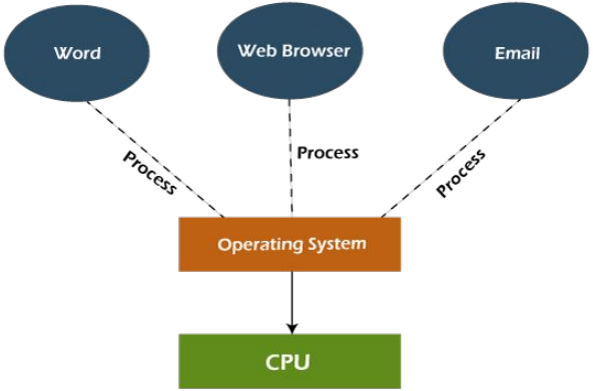
```
        // Calling myThread
        MyThread thread2 = new MyThread();
        System.out.println(thread2.getName());
        System.out.println(thread2.isAlive());

        thread2.start();
        System.out.println(thread2.isAlive());

        thread2.setDaemon(true);
        System.out.println("is Daemon: ---> " + thread2.isDaemon());
    }
}
```

```
Thread-0
false
true
This thread is running.
Exception in thread "MAIN AGAIN " java.lang.IllegalThreadStateException Create breakpoint
    at java.base/java.lang.Thread.setDaemon(Thread.java:1403)
    at threadmethos.ThreadMethods.main(ThreadMethods.java:48)
```

Class: Thread Methods

The Concept of Multitasking (1/n)

- To help users **Operating System** accommodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine.
- This Multitasking can be enabled in two ways:
  - **Process-Based Multitasking**
  - **Thread-Based Multitasking**



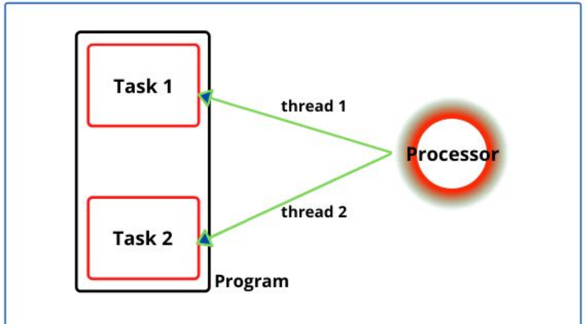
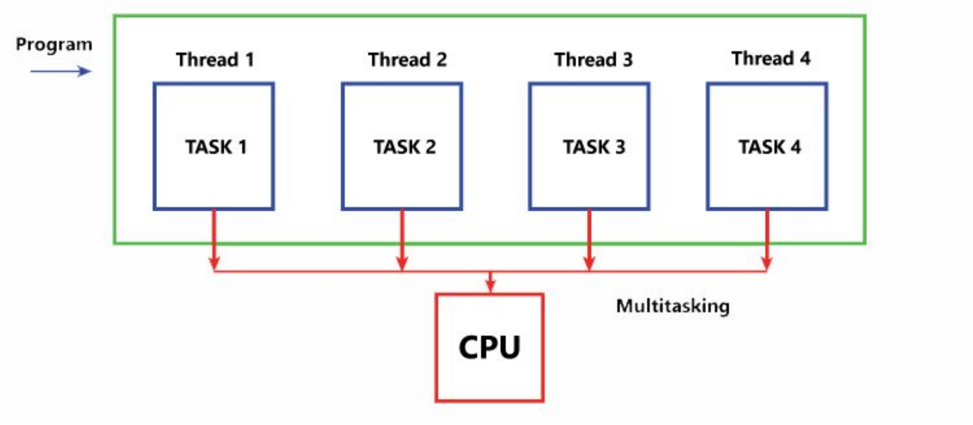


Fig: Thread-based Multitasking

The Concept of Multitasking (2/n)

- **Process-Based Multitasking** are heavyweight and each process was allocated by a separate memory area.
- And as the process is **heavyweight the cost of communication between processes is high and it takes a long time for switching between processes** as it involves actions such as ***loading, saving in registers, updating maps, lists, etc***



The Concept of Multitasking (3/n)

- As we discussed above **Threads** are provided with lightweight nature and share the same address space, and the cost of communication between threads is also low.

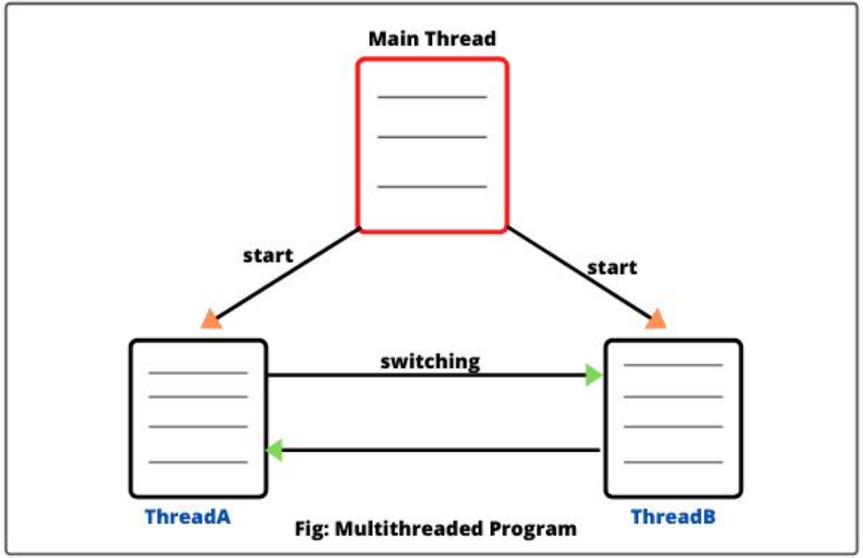
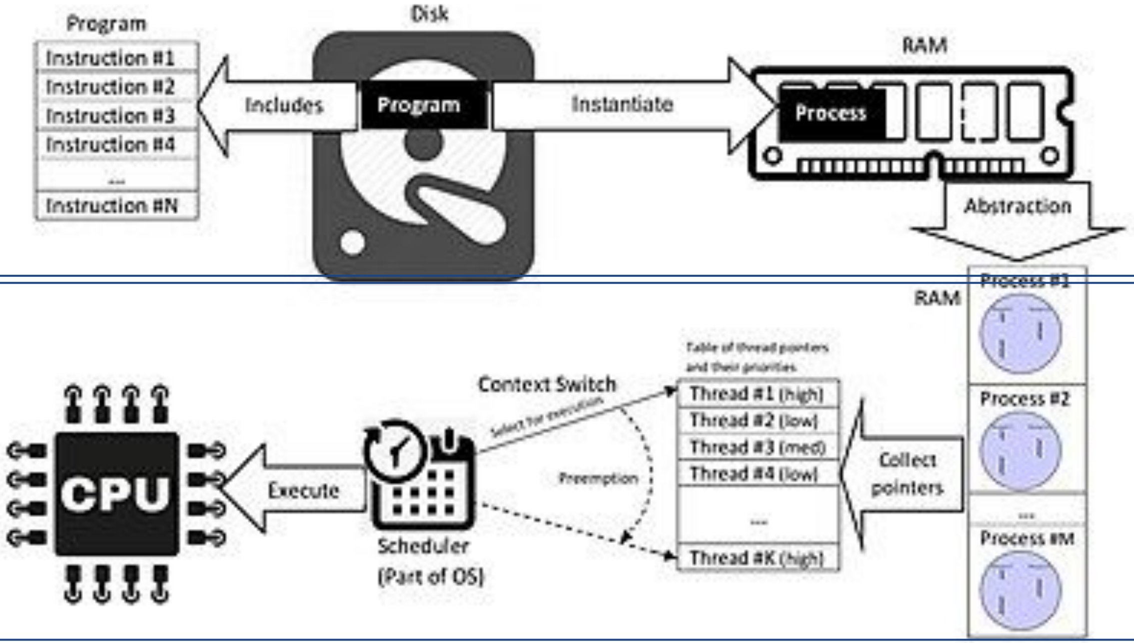


Fig: Multithreaded Program

The Concept of Multitasking (4/n)

Program vs. Process vs. Thread, Scheduling, Preemption, Context Switching



Thread method (1/n)

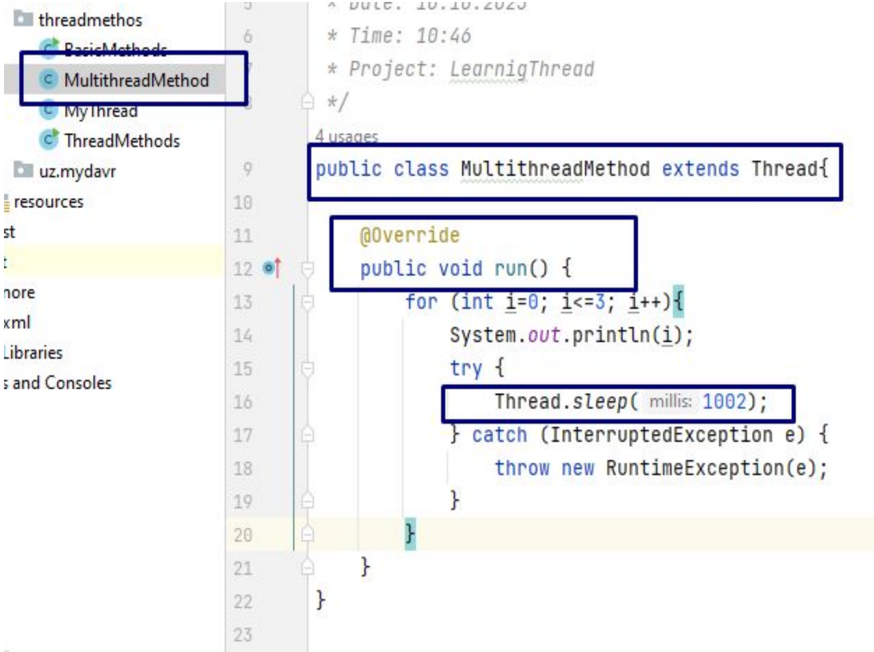
- we will review methods that deal with thread states and properties, as well as their **synchronization and interruption.**
- We also discuss methods for controlling **thread priority, daemon threads**, **sleeping and waiting**, as well as a couple of miscellaneous methods that do not fall into any of the aforementioned categories.

- start()
- run()
- getState()
- isAlive()
- getName()/setName()
- getState()
- join()

- interrupt()
- isInterrupted()
- getPriority()
- setPriority(int priority)
- wait()

Thread method (2/n)

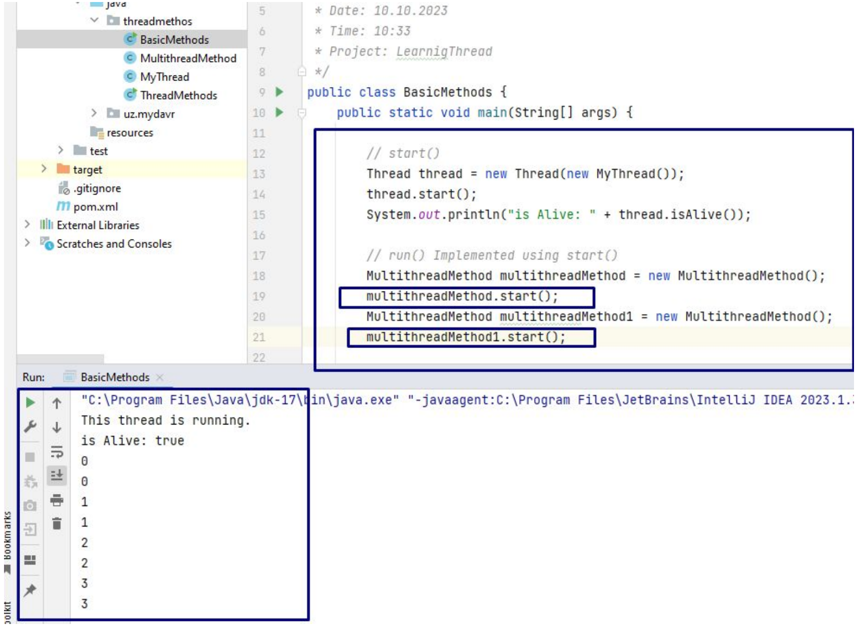
- The `run()` method contains the code that will be executed in the thread.
- It must be overridden when extending the `Thread` class or implementing the `Runnable` interface.



```java
 * Date: 10.10.2023
 * Time: 10:46
 * Project: LearnigThread
 */
4 usages
public class MultithreadMethod extends Thread{

    @Override
    public void run() {
        for (int i=0; i<=3; i++){
            System.out.println(i);
            try {
                Thread.sleep( millis: 1002);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Thread method (3/n)

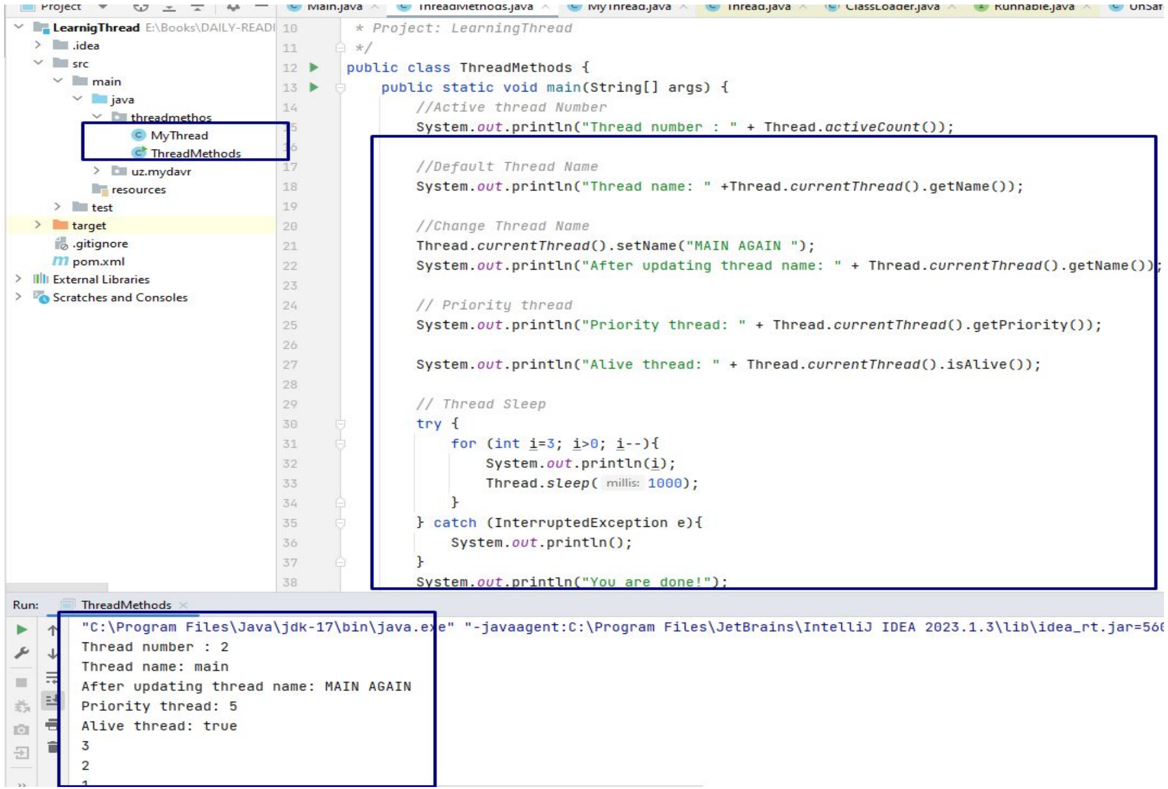
- The **start()** method initiates the execution of a thread.
- It calls the `run()` method defined in your thread class or runnable object.



```java
 * Date: 10.10.2023
 * Time: 10:33
 * Project: LearnigThread
 */
public class BasicMethods {
    public static void main(String[] args) {

        // start()
        Thread thread = new Thread(new MyThread());
        thread.start();
        System.out.println("is Alive: " + thread.isAlive());

        // run() Implemented using start()
        MultithreadMethod multithreadMethod = new MultithreadMethod();
        multithreadMethod.start();
        MultithreadMethod multithreadMethod1 = new MultithreadMethod();
        multithreadMethod1.start();

```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1.
This thread is running.
is Alive: true
0
0
1
1
2
2
3
3
```

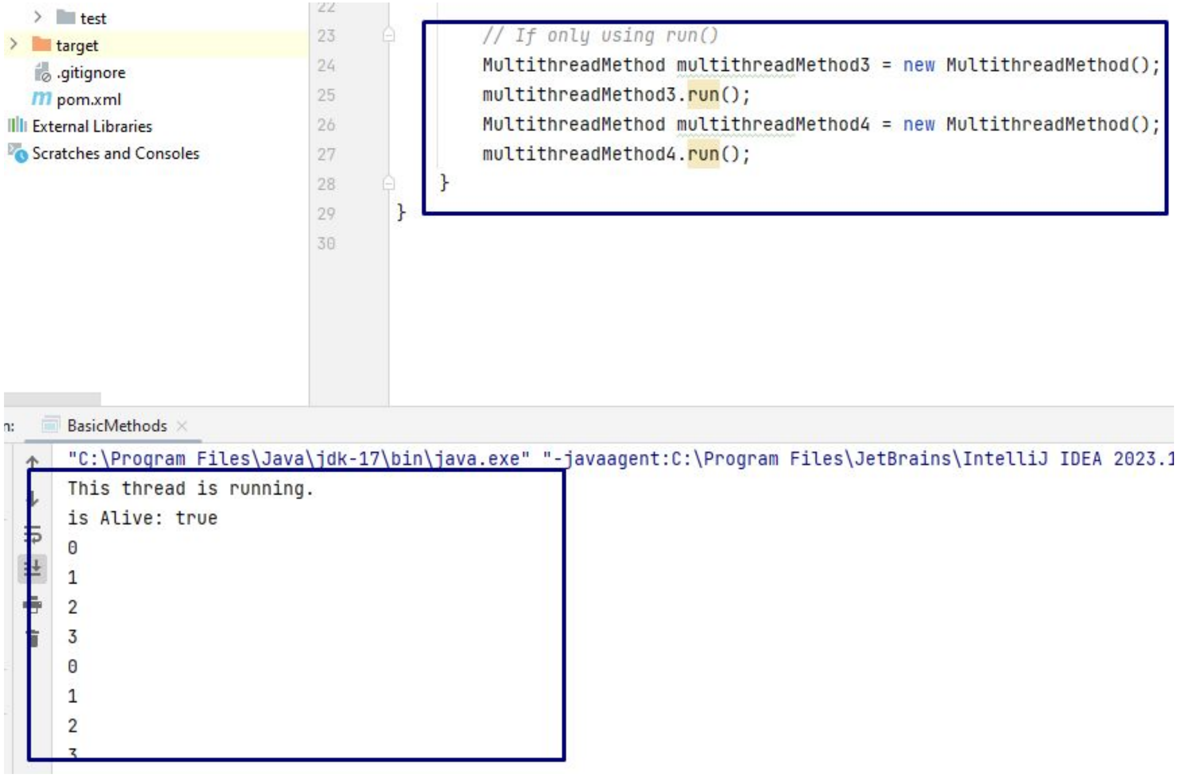
Thread method (4/n)



```java
 * Project: LearningThread
 */
public class ThreadMethods {
    public static void main(String[] args) {
        //Active thread Number
        System.out.println("Thread number : " + Thread.activeCount());

        //Default Thread Name
        System.out.println("Thread name: " +Thread.currentThread().getName());

        //Change Thread Name
        Thread.currentThread().setName("MAIN AGAIN ");
        System.out.println("After updating thread name: " + Thread.currentThread().getName());

        // Priority thread
        System.out.println("Priority thread: " + Thread.currentThread().getPriority());

        System.out.println("Alive thread: " + Thread.currentThread().isAlive());

        // Thread Sleep
        try {
            for (int i=3; i>0; i--){
                System.out.println(i);
                Thread.sleep( millis: 1000);
            }
        } catch (InterruptedException e){
            System.out.println();
        }
        System.out.println("You are done!");
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar=56
Thread number : 2
Thread name: main
After updating thread name: MAIN AGAIN
Priority thread: 5
Alive thread: true
3
2
1
```

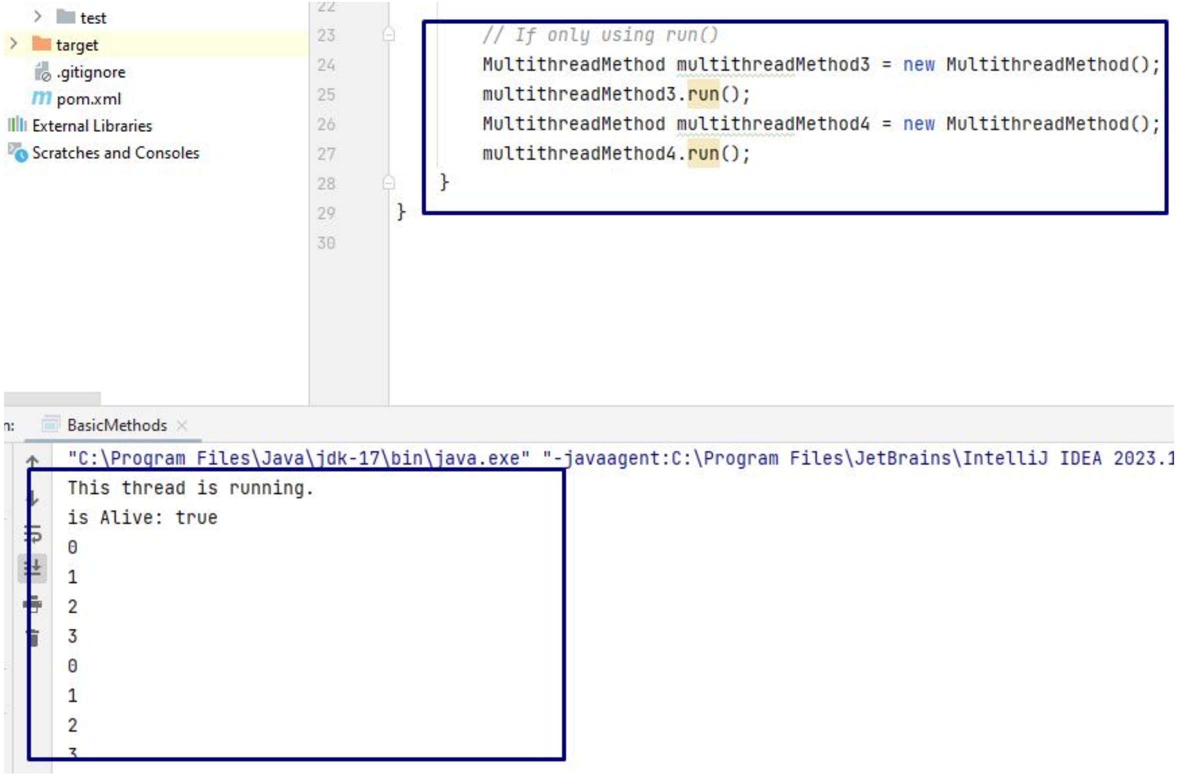
Thread method (5/n)



```
        // If only using run()
        MultithreadMethod multithreadMethod3 = new MultithreadMethod();
        multithreadMethod3.run();
        MultithreadMethod multithreadMethod4 = new MultithreadMethod();
        multithreadMethod4.run();
    }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1
This thread is running.
is Alive: true
0
1
2
3
0
1
2
3
```

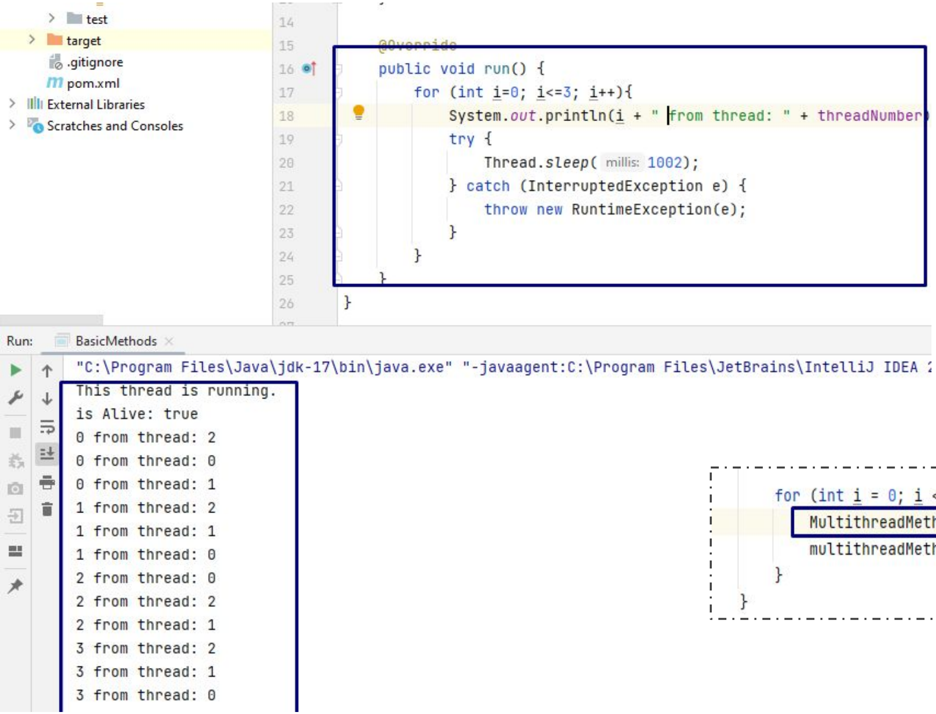
Thread method (5/n)



```java
        // If only using run()
        MultithreadMethod multithreadMethod3 = new MultithreadMethod();
        multithreadMethod3.run();
        MultithreadMethod multithreadMethod4 = new MultithreadMethod();
        multithreadMethod4.run();
    }
}
```

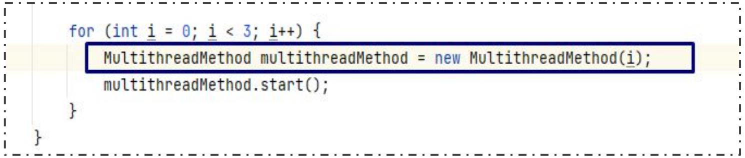
```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1
This thread is running.
is Alive: true
0
1
2
3
0
1
2
3
```

Thread method (6/n)



```
    @Override
    public void run() {
        for (int i=0; i<=3; i++){
            System.out.println(i + " from thread: " + threadNumber
            try {
                Thread.sleep( millis: 1002);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

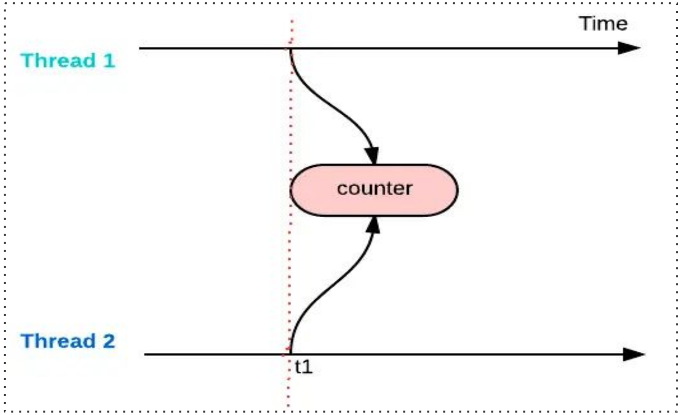
```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2
This thread is running.
is Alive: true
0 from thread: 2
0 from thread: 0
0 from thread: 1
1 from thread: 2
1 from thread: 1
1 from thread: 0
2 from thread: 0
2 from thread: 2
2 from thread: 1
3 from thread: 2
3 from thread: 1
3 from thread: 0
```



```
        for (int i = 0; i < 3; i++) {
            MultithreadMethod multithreadMethod = new MultithreadMethod(i);
            multithreadMethod.start();
        }
    }
```

Thread Safety(1/n)

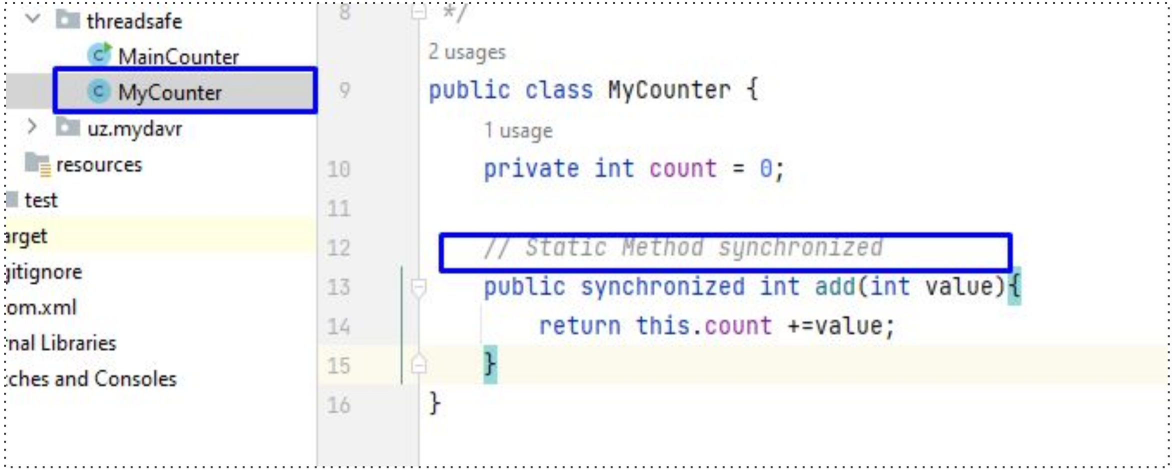
- **Thread safety** in java is the process to make our program safe to use in multithreaded environment, there are different ways through which we can make our program thread safe:
  - **Synchronization**
  - **Atomic Variable**
  - **Volatile**



resource

Thread Safety(2/n) - Synchronized

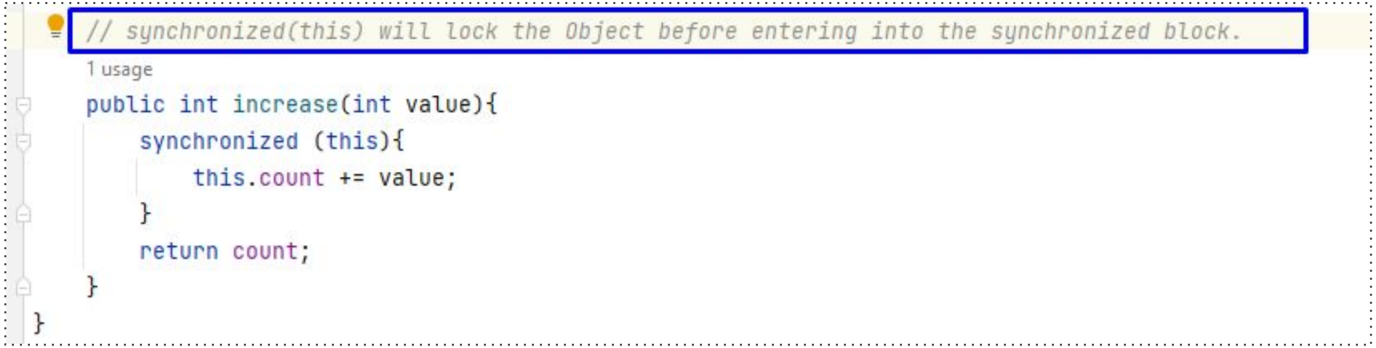
- **Synchronization** is the tool using which we can achieve **thread-safety**, JVM guarantees that synchronized code will be executed by only **one thread at a time**.
- java keyword **synchronized** is used to **create synchronized** code and internally it uses locks **on Object or Class** to make sure only one **thread** is executing the **synchronized code.**



```
*/
2 usages
public class MyCounter {
    1 usage
    private int count = 0;

    // Static Method synchronized
    public synchronized int add(int value){
        return this.count +=value;
    }
}
```

Thread Safety(3/n) - Synchronized

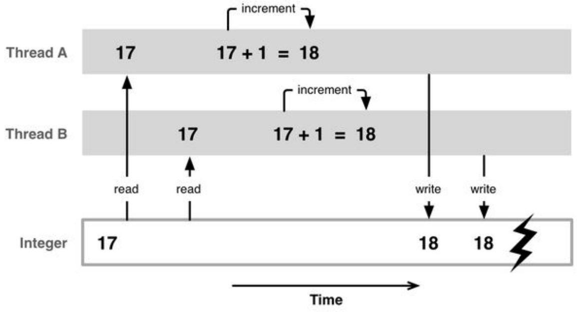
- **Java synchronization** works **on locking and unlocking** of the resource before any thread enters into synchronized code, it has to acquire **the lock on the Object** and when code execution ends, it unlocks the resource that can be locked by other threads. In the meantime, other threads are in wait state to lock **the synchronized resource**.
- When a method is synchronized, **it locks the Object, if method is static it locks the Class**, so it's always best practice to use synchronized block to lock the only sections of method that needs **synchronization**.



```
    // synchronized(this) will lock the Object before entering into the synchronized block.
    1 usage
    public int increase(int value){
        synchronized (this){
            this.count += value;
        }
        return count;
    }
}
```

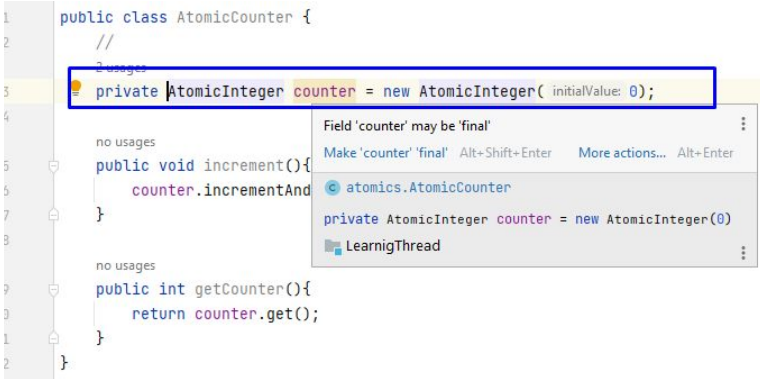
Thread Safety(4/n) - Synchronized

- Java Synchronization works only in the same JVM, so if you need to lock some resource **in multiple JVM environment**, it will not work and you might have to look after some **global locking mechanism**.
- We should not use any object that is maintained in **a constant pool**,
    - for example String should not be used for synchronization because if any other code is also **locking on same String**,
    - it will try to acquire lock on **the same reference object** from String pool and even though **both the codes are unrelated**, they will lock each other



Thread Safety(5/n) - Atomic Variable

- **Atomic variables** are used to perform atomic operations on primitive data types such as **int, long, double, etc.**
- They provide a way to modify the value of the variable atomically (i.e., in one atomic operation), thus **avoiding race conditions**.
- This means that an **atomic variable operation will complete before another operation can start**. In Java, the java.util.concurrent.atomic package provides atomic variables.



```
public class AtomicCounter {
    //
    2 usages
    private AtomicInteger counter = new AtomicInteger( initialValue: 0);

    no usages
    public void increment(){
        counter.incrementAnd
    }

    no usages
    public int getCounter(){
        return counter.get();
    }
}
```

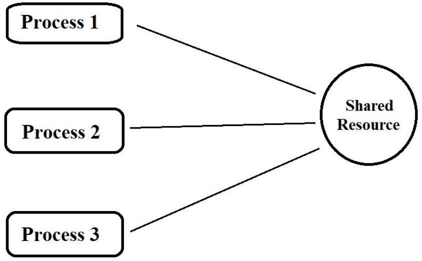
Field 'counter' may be 'final'

Make 'counter' 'final' Alt+Shift+Enter More actions... Alt+Enter

atomics.AtomicCounter

```
private AtomicInteger counter = new AtomicInteger(0)
```

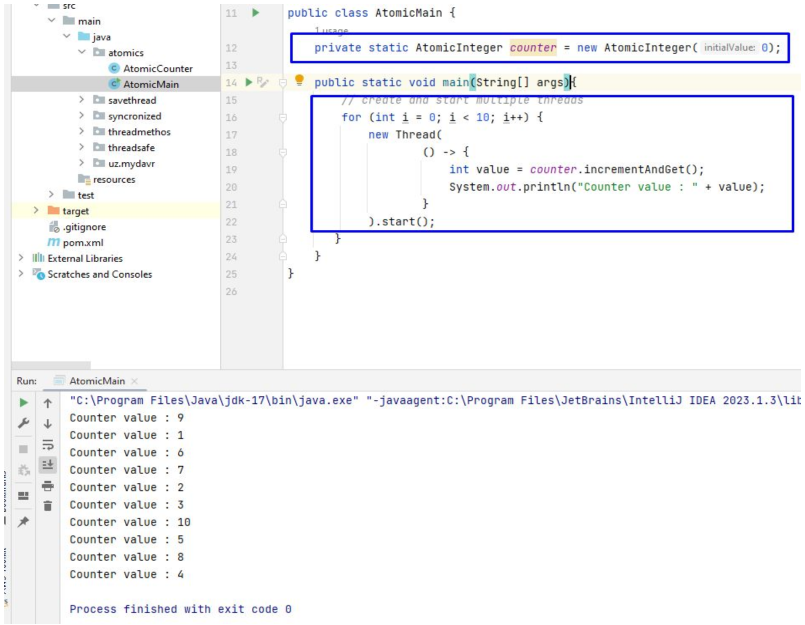
LearnigThread



race conditions

Thread Safety(6/n) - Atomic Variable

- let’s say we have an integer counter that **multiple threads** can access **concurrently**. If we use an atomic integer, we can modify the counter atomically, like this:

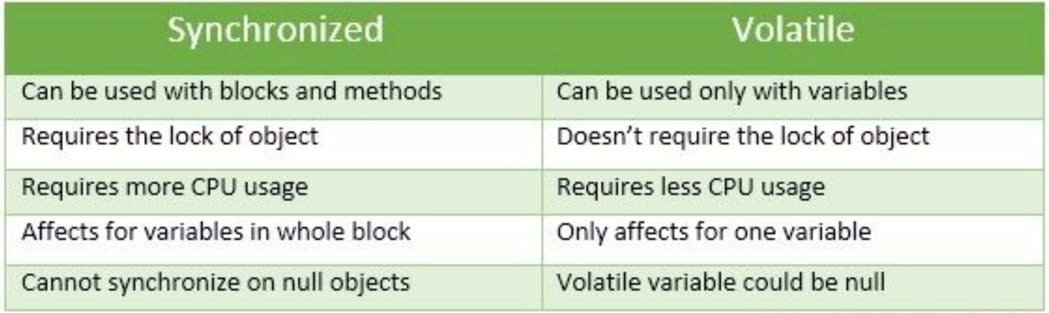


```java
public class AtomicMain {
    private static AtomicInteger counter = new AtomicInteger( initialValue: 0);

    public static void main(String[] args){
        // create and start multiple threads
        for (int i = 0; i < 10; i++) {
            new Thread(
                    () -> {
                        int value = counter.incrementAndGet();
                        System.out.println("Counter value : " + value);
                    }
            ).start();
        }
    }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1.3\lib
Counter value : 9
Counter value : 1
Counter value : 6
Counter value : 7
Counter value : 2
Counter value : 3
Counter value : 10
Counter value : 5
Counter value : 8
Counter value : 4

Process finished with exit code 0
```

Thread Safety(7/n) - Volatile Variables

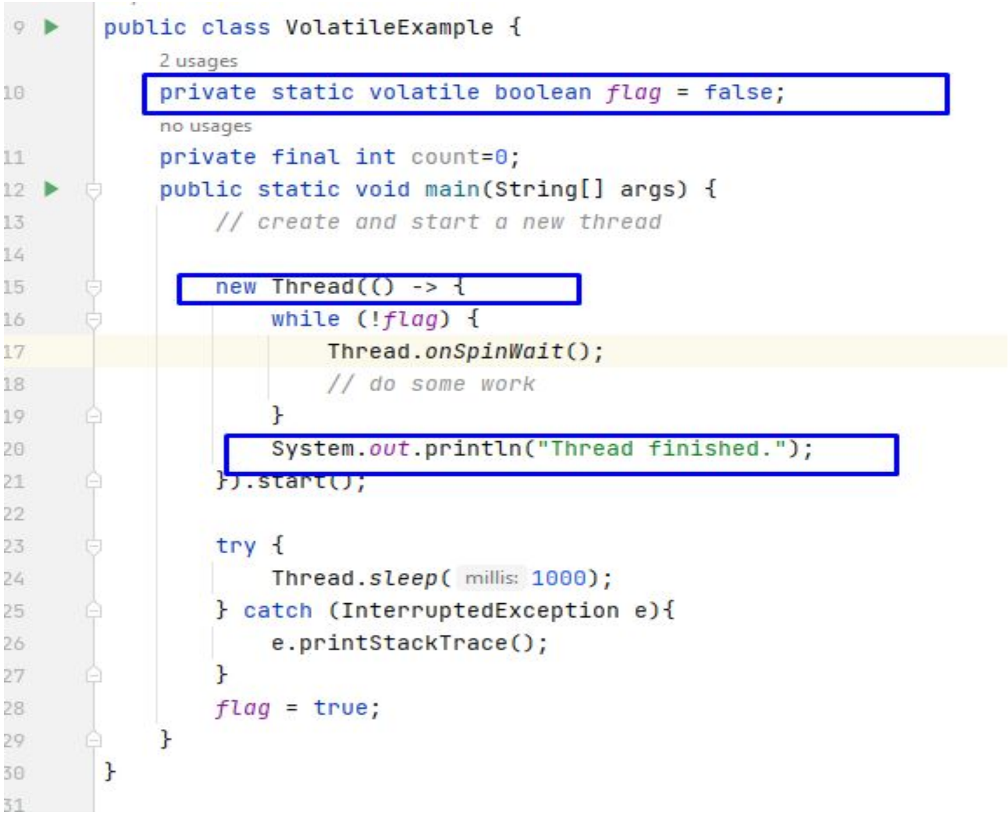
- volatile is a lightweight form of synchronization that tackles the visibility and ordering aspects. volatile is used as a **field** modifier.
- The purpose of volatile is to ensure that when one thread writes a value to a field, the value written is "immediately available" to any thread that subsequently reads it.
- `volatile` also **limits reordering** of accesses (accesses to the reference) by preventing the compiler and Runtime from reordering of code.



Synchronized	Volatile
Can be used with blocks and methods	Can be used only with variables
Requires the lock of object	Doesn’t require the lock of object
Requires more CPU usage	Requires less CPU usage
Affects for variables in whole block	Only affects for one variable
Cannot synchronize on null objects	Volatile variable could be null

www.easyjavase.blogspot.com

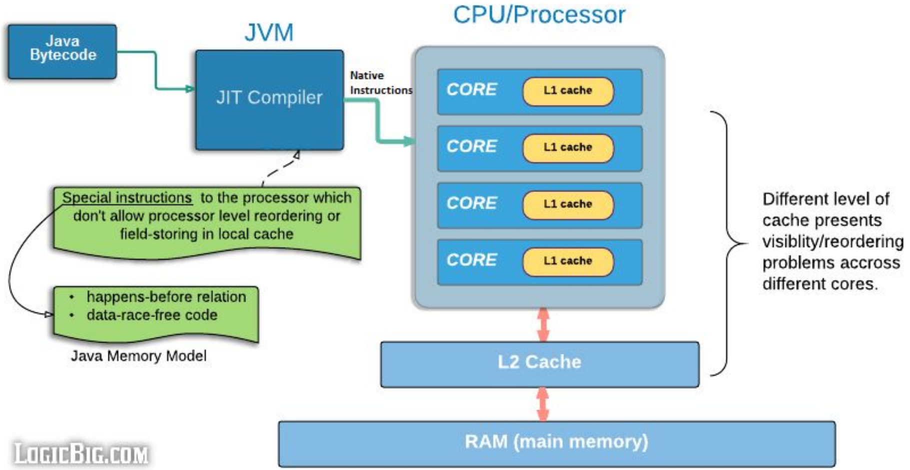
Thread Safety(8/n) - Volatile Variables



```
public class VolatileExample {
    2 usages
    private static volatile boolean flag = false;
    no usages
    private final int count=0;
    public static void main(String[] args) {
        // create and start a new thread

        new Thread(() -> {
            while (!flag) {
                Thread.onSpinWait();
                // do some work
            }
            System.out.println("Thread finished.");
        }).start();

        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e){
            e.printStackTrace();
        }
        flag = true;
    }
}
```

Thread Safety(9/n)-Happens-Before Relationship

- Two actions can be ordered by a happens-before relationship.
- If one action happens before another, then the first is **visible** to and **ordered** before the second (for example, the write of a default value to every field of an object constructed by a thread need not happen before the beginning of that thread, as long as no read ever observes that fact).


Java Bytecode
JVM
JIT Compiler
Native Instructions
CPU/Processor
CORE
L1 cache
CORE
L1 cache
CORE
L1 cache
CORE
L1 cache
Special instructions to the processor which don't allow processor level reordering or field-storing in local cache
• happens-before relation
• data-race-free code
Java Memory Model
Different level of cache presents visiblity/reordering problems accross different cores.
L2 Cache
RAM (main memory)
LogicBig.com

Thread Safety(10/n)

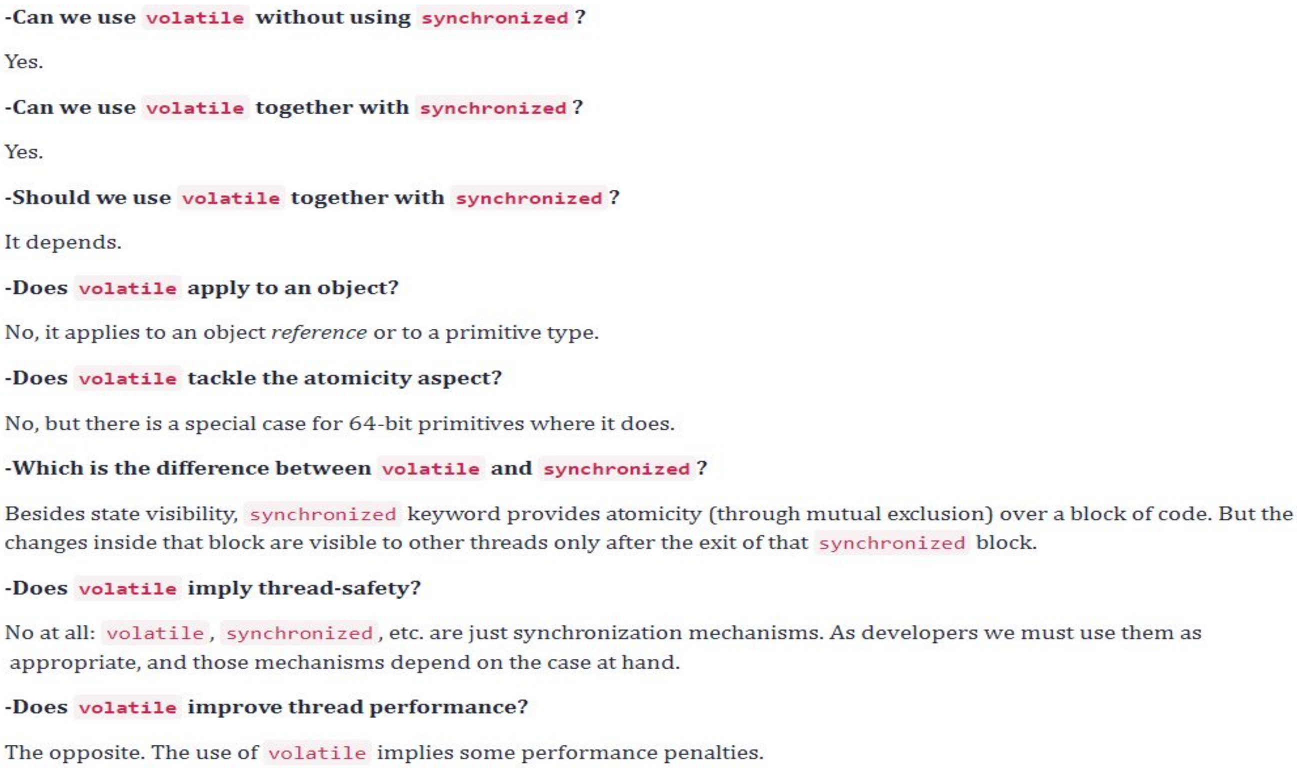
There are basically **four ways to make variable access safe in shared-memory concurrency:**

- **Confinement.** Don't share the variable between **threads**. This idea is called confinement, and we'll explore it today.
- **Immutability.** Make the shared data immutable. We've talked a lot about **immutability** already, but there are some additional constraints for concurrent programming that we'll talk about in this reading.
- **Threadsafe data type.** Encapsulate the shared data in an existing **threadsafe** data type that does the coordination for you. We'll talk about that today.
- **Synchronization.** Use synchronization to keep **the threads from accessing the variable at the same time. Synchronization** is what you need to build your own **threadsafe** data type.

Thread Safety(11/n) - Strategy 1: Confinement

- **Confinement.** Don’t share the variable between **threads**. This idea is called confinement, and we’ll explore it today
- Local variables are always thread confined
- A local variable is stored in the stack, and each thread has its own stack.

Q&A



-Can we use `volatile` without using `synchronized` ?

Yes.

-Can we use `volatile` together with `synchronized` ?

Yes.

-Should we use `volatile` together with `synchronized` ?

It depends.

-Does `volatile` apply to an object?

No, it applies to an object *reference* or to a primitive type.

-Does `volatile` tackle the atomicity aspect?

No, but there is a special case for 64-bit primitives where it does.

-Which is the difference between `volatile` and `synchronized` ?

Besides state visibility, `synchronized` keyword provides atomicity (through mutual exclusion) over a block of code. But the changes inside that block are visible to other threads only after the exit of that `synchronized` block.

-Does `volatile` imply thread-safety?

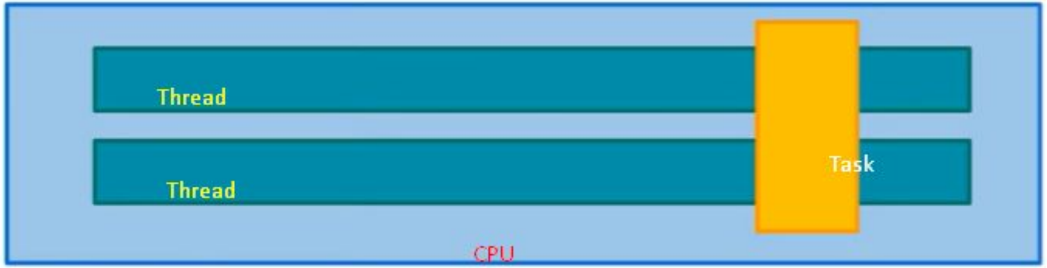
No at all: `volatile` , `synchronized` , etc. are just synchronization mechanisms. As developers we must use them as appropriate, and those mechanisms depend on the case at hand.

-Does `volatile` improve thread performance?

The opposite. The use of `volatile` implies some performance penalties.

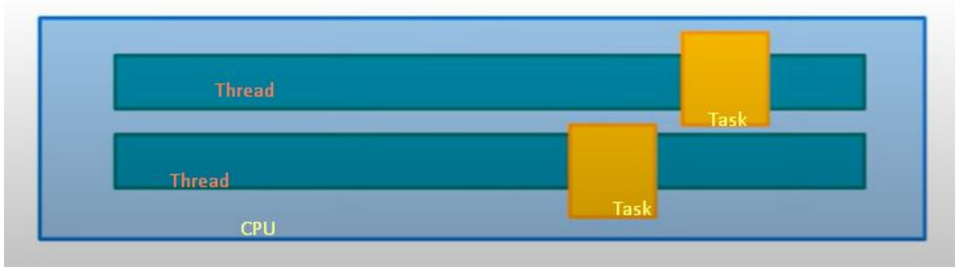
Conclusion (1/n)

- **Shared mutable state issues**
  - Race conditions
  - Invisible writes
  - Congestion
  - Nested monitor lockout
  - Starvation
  - Slipped conditions
  - Missed signals

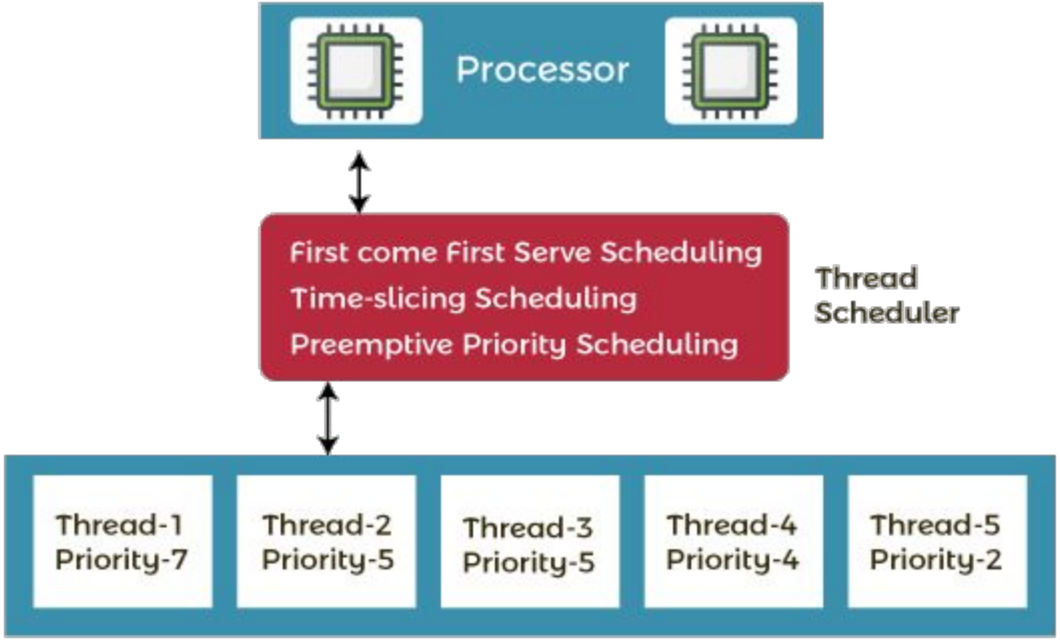


Conclusion (2/n)

- **No shared mutable state concurrency**
  - Separate state concurrency
  - Functional parallelism
  - Parallel pipelines
  - Etc.

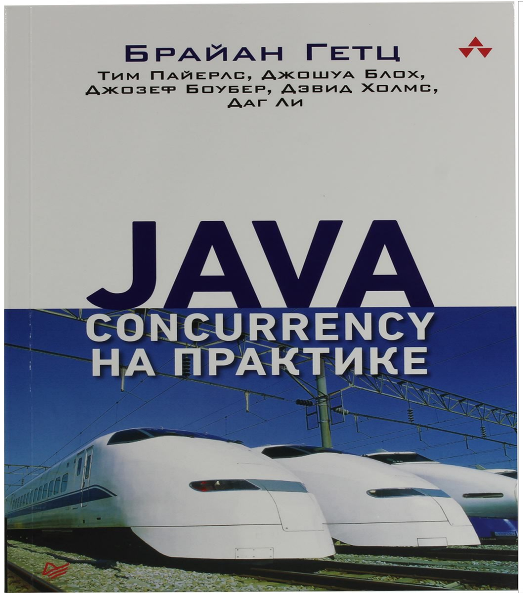


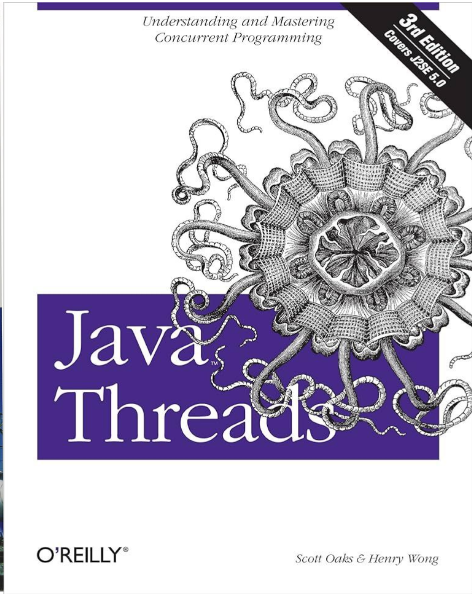
Conclusion (3/n)

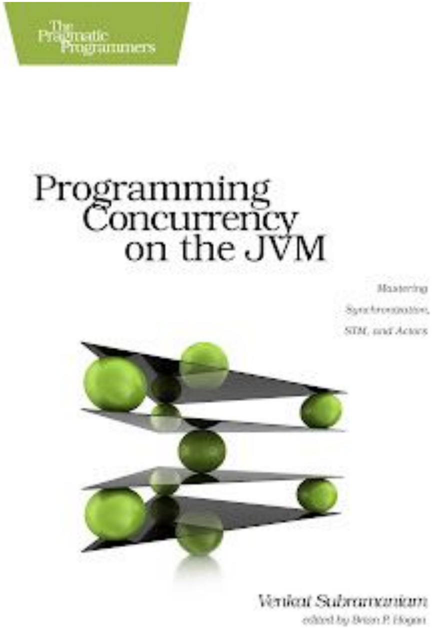


Working of Thread Scheduler

Resources


Брайан Гетц
Тим Пайерлс, Джошуа Блох, Джозеф Боубер, Дэвид Холмс, Даг Ли
JAVA
CONCURRENCY
НА ПРАКТИКЕ


Understanding and Mastering Concurrent Programming
3rd Edition
Covers J2SE 5.0
Java
Threads
O'REILLY
Scott Oaks & Henry Wong


The Pragmatic Programmers
Programming Concurrency on the JVM
Mastering Synchronization, STM, and Actors
Venkat Subramaniam
edited by Brian P. Hogan

Reference

1.Java Concurrency and Multithreading

2.Thread Type: Daemon Thread

3. Java thread methods

4. Understanding Atomic, Volatile, and Synchronized Variables in Java

5. Thread-Safety

6. Thread-Safety in Java

7. Threads in more info

8. Java concurrency understanding the volatile keyword

Thank you!

Presented by

Hamdamboy Urunov

(hamdamboy.urunov@gmail.com)