Thread Safety and how to achieve it in Java

Difficulty Level : Medium Last Updated : 24 Jun, 2021

As we know Java has a feature, Multithreading, which is a process of running multiple threads simultaneously. When multiple threads are working on the same data, and the value of our data is changing, that scenario is not thread-safe and we will get inconsistent results. When a thread is already working on an object and preventing another thread on working on the same object, this process is called Thread-Safety.

How to achieve Thread Safety

There are four ways to achieve Thread Safety in Java. These are:

1. Using Synchronization.
2. Using Volatile Keyword.
3. Using Atomic Variable.
4. Using Final Keyword.

Using Synchronization

Synchronization is the process of allowing only one thread at a time to complete the particular task. It means when multiple threads executing simultaneously, and want to access the same resource at the same time, then the problem of inconsistency will occur. so synchronization is used to resolve inconsistency problem by allowing only one thread at a time.
Synchronization uses a *synchronized keyword*. Synchronized is the modifier that creates a block of code known as a critical section.

```
class A {
    synchronized void sum(int n)
    {

        // Creating a thread instance
        Thread t = Thread.currentThread();
```

```
            for (int i = 1; i <= 5; i++) {
                System.out.println(
                    t.getName() + " : " + (n + i));
            }
        }
    }

    // Class B extending thread class
    class B extends Thread {

        // Creating an object of class A
        A a = new A();
        public void run()
        {

            // Calling sum() method
            a.sum(10);
        }
    }
    class Test {
        public static void main(String[] args)
        {

            // Creating an object of class B
            B b = new B();

            // Initializing instance t1 of Thread
            // class with object of class B
            Thread t1 = new Thread(b);

            // Initializing instance t2 of Thread
            // class with object of class B
            Thread t2 = new Thread(b);

            // Initializing thread t1 with name
            //'Thread A'
            t1.setName("Thread A");

            // Initializing thread t2 with name
            //'Thread B'
            t2.setName("Thread B");

            // Starting thread instance t1 and t2
            t1.start();
            t2.start();
        }
    }
```

Output:

```
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15
```

Using Volatile keyword

A volatile keyword is a field modifier that ensures that the object can be used by multiple threads at the same time without having any problem. volatile is one good way of ensuring that the Java program is thread-safe. a volatile keyword can be used as an alternative way of achieving Thread Safety in Java.

```
public class VolatileExample {

    // Initializing volatile variables
    // a, b
    static volatile int a = 0, b = 0;

    // Defining a static void method
    static void method_one()
    {
        a++;
        b++;
    }
```

```
    // Defining static void method
    static void method_two()
    {
        System.out.println(
            "a=" + a + " b=" + b);
    }

    public static void main(String[] args)
    {

        // Creating an instance t1 of
        // Thread class
        Thread t1 = new Thread() {
            public void run()
            {
                for (int i = 0; i < 5; i++)
                    method_one();
            }
        };

        // Creating an instance t2 of
        // Thread class
        Thread t2 = new Thread() {
            public void run()
            {
                for (int i = 0; i < 5; i++)
                    method_two();
            }
        };

        // Starting instance t1 and t2
        t1.start();
        t2.start();
    }
}
```

Output:

```
a=5 b=5
a=5 b=5
a=5 b=5
a=5 b=5
a=5 b=5
```

Using Atomic Variable

Using an atomic variable is another way to achieve thread-safety in java. When variables are shared by multiple threads, the atomic variable ensures that threads don't crash into each other.

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

class Counter {

    // Creating a variable of
    // class type AtomicInteger
    AtomicInteger count
        = new AtomicInteger();

    // Defining increment() method
    // to change value of
    // AtomicInteger variable
    public void increment()
    {
        count.incrementAndGet();
    }
}

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

        // Creating an instance of
        // Counter class
        Counter c = new Counter();

        // Creating an instance t1 of
        // Thread class
        Thread t1 = new Thread(
            new Runnable() {
```

```
                public void run()
                {
                    for (int i = 1; i <= 2000; i++) {
                        c.increment();
                    }
                }
            });

        // Creating an instance t2
        // of Thread class
        Thread t2 = new Thread(
            new Runnable() {
                public void run()
                {
                    for (int i = 1; i <= 2000; i++) {
                        c.increment();
                    }
                }
            });

        // Calling start() method with t1 and t2
        t1.start();
        t2.start();

        // Calling join method with t1 and t2
        t1.join();
        t2.join();

        System.out.println(c.count);
    }
}
```

Output:

```
4000
```

Using Final keyword

Final Variables are also thread-safe in java because once assigned some reference of an object It cannot point to reference of another object.

```
public class FinalTest {

    // Initializing a string
    // variable of final type
    final String str
        = new String("hello");

    // Defining a method to
    // change the value of the final
    // variable which is not possible,
    // hence the error will be shown
    void method()
    {
        str = "world";
    }
}
```

Output:

```
Compilation Error in java code :-
prog.java:14: error: cannot assign a value to final variable str
        str = "world";
        ^
1 error
```