MIT World Peace University

Object Oriented Programming with Java and C++
Second Year B. Tech, Semester 1

Multithreading using Thread Class and Runnable Interface in Java

---

Practical Report
Assignment 7

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

November 23, 2022

Contents

1 Aim and Objectives

Aim

Implementing Solutions on Multithreading using Thread Class and Runnable Interface

Objectives

1. To understand Multithreading in Java
2. To learn two different ways to create threads in Java

2 Problem Statements

2.1 Problem 1 in Java

Write a program to create a multithreaded calculator that does addition, subtraction, multiplication, and division using separate threads. Additionally also handle '/ by zero' exception by the division method.

2.2 Problem 2 in Java

Print even and odd numbers in increasing order using two threads in Java

3 Theory

3.1 Multithreading in Java

In Java, Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU. A thread in Java is a lightweight process requiring fewer resources to create and share the process resources.Multithreading and Multiprocessing are used for multitasking in Java, but we prefer multithreading over multiprocessing.

3.2 Life Cycle of a Thread in Java

There are five states a thread has to go through in its life cycle. This life cycle is controlled by JVM (Java Virtual Machine). These states are:

1. New : In this state, a new thread begins its life cycle. This is also called a born thread. The thread is in the new state if you create an instance of Thread class but before the invocation of the start() method.
2. Runnable : A thread becomes runnable after a newly born thread is started. In this state, a thread would be executing its task.
3. Running : When the thread scheduler selects the thread then, that thread would be in a running state.
4. Non-Runnable (Blocked) : The thread is still alive in this state, but currently, it is not eligible to run.
5. Terminated : A thread that is in a terminated state does not consume ant cycle of the CPU.

3.3 Ways of Creating a Thread

There are multiple ways of creating Threads in Java

1. By Extending the Thread Class

```
class Multi extends Thread{
  public void run(){
    System.out.println("thread is running...");
  }
  public static void main(String args[]){
    Multi t1=new Multi();
    t1.start();
  }
}

```

2. By Implementing the Runnable Interface

```
class Multi3 implements Runnable{
  public void run(){
    System.out.println("thread is running...");
  }

  public static void main(String args[]){
    Multi3 m1=new Multi3();
    Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
    t1.start();
  }
}

```

3. Using the Thread Class

```
public class MyThread1
{
  // Main method
  public static void main(String argvs[])
  {
    // creating an object of the Thread class using the constructor Thread(
    String name)
    Thread t= new Thread("My first thread");

    // the start() method moves the thread to the active state
    t.start();
    // getting the thread name by invoking the getName() method
    String str = t.getName();
    System.out.println(str);
  }
}
```

3.4 Performing Multiple tasks by multiple threads

You could just create multiple classes that perform their own tasks, each a child of the Thread class. So you could then invoke instances of those classes, and run them as multiple threads.

```
class MultithreadEx3 extends Thread
{
```

```
    public void run()
    {
        System.out.println("Start task one");
    }
}
class MultithreadEx4 extends Thread
{
    public void run()
    {
        System.out.println("Start task two");
    }
}
class Run
{
    public static void main(String args[])
    {
        MultithreadEx3 th1 = new MultithreadEx3();
        MultithreadEx4 th2 = new MultithreadEx4();
        th1.start();
        th2.start();
    }
}
```

3.5 Thread Scheduler

A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.

However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

- *Priority*: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.
- *Time of Arrival*: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, arrival time of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

3.6 Joining a Thread in Java

java.lang.Thread class provides the join() method which allows one thread to wait until another thread completes its execution.

If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program. If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period.

However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify. There are three overloaded join functions.

```java
public class JoinExample1 extends Thread
{
    public void run()
    {
        for(int i=1; i<=4; i++)
        {
            try
            {
                Thread.sleep(500);
            }catch(Exception e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        JoinExample1 thread1 = new JoinExample1();
        JoinExample1 thread2 = new JoinExample1();
        JoinExample1 thread3 = new JoinExample1();
        thread1.start();
       try
        {
        thread1.join();
        }catch(Exception e){System.out.println(e);}
        thread2.start();
        thread3.start();
    }
}
```

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++, and javac, with JDK 18.0.2 for Java

5 Input

For Problem 1

1. 2 numbers
2. Choice about what to do with those numbers

For Problem 2

1. The Maximum limit up to which the user wants to see the odd and even numbers printed

6 Output

For Problem 1

1. Menu about what to do with numbers
2. Output of the calculation done with those numbers

For Problem 2

1. Even numebers and Odd numbers in Ascending order upto the specified limit.

7 Code

7.1 Java Implementation of Problem 1

```java
// Krishnaraj Thadesar
// Batch A1, PA20
// OOPCJ Assignment 7.1
// Write a program to create a multithreaded calculator that does addition, subtraction,
// multiplication, and division using separate threads.
// Additionally also handle '/ by zero' exception by the division method.


import java.lang.Thread;
import java.util.Scanner;

class Calculator extends Thread implements Runnable {
    public int a, b, what_to_do = 0;

    Calculator(int a, int b, int choice, String name) {
        this.a = a;
        this.b = b;
        this.what_to_do = choice;
        this.setName(name);
    }

    @Override
    public synchronized void start() {
        System.out.println("Starting the Thread");
        System.out.println("The Name of this Thread is: " + getName());
        super.start();
    }

    @Override
    public void run() {
        switch (what_to_do) {
            case 1:
                System.out.println(a + b);
                break;
            case 2:
                System.out.println(a - b);
                break;
            case 3:
                System.out.println(a * b);
                break;
            case 4:
                try {
                    System.out.println(a / b);
                } catch (ArithmeticException e) {
                    System.out.println("You cant Divide by Zero!");
                }
                break;
            default:
```

```
                break;
            }
        }
}

public class assignment_7_problem_1 {
    public static Calculator add, sub, mul, div;
    public static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int choice = 0;
        int a, b;
        System.out.println("Welcome To Thread Calculator of Assignment 7");
        while (choice != 5) {
            System.out.println("What would you like to do? ");
            System.out.println(
                    "1. Addition of 2 Numbers\n2. Subtraction of 2 Numbers\n3. Multiplication of 2 Numbers\n4. Division of 2 Numbers\n\n");
            choice = input.nextInt();
            if (choice == 5) {
                break;
            }
            System.out.println("Enter the 2 Numbers");
            a = input.nextInt();
            b = input.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("You have chosen Addition!");
                    add = new Calculator(a, b, choice, "Adder");
                    try {
                        add.start();
                        add.join();
                    } catch (Exception e) {
                        System.out.println("Got some problem with making the thread!");
                        System.out.println(e);
                    }
                    break;
                case 2:
                    System.out.println("You have chosen Subtraction!");
                    sub = new Calculator(a, b, choice, "Subtractor");
                    try {
                        sub.start();
                        sub.join();
                    } catch (Exception e) {
                        System.out.println("Got some problem with making the thread!");
                        System.out.println(e);
                    }
                    break;
                case 3:
                    System.out.println("You have chosen Multiplication!");
                    mul = new Calculator(a, b, choice, "Multiplier");
                    try {
                        mul.start();
                        mul.join();
                    } catch (Exception e) {
                        System.out.println("Got some problem with making the thread!");
```

```
                        System.out.println(e);
                    }
                    break;
                case 4:
                    System.out.println("You have chosen Division!");
                    div = new Calculator(a, b, choice, "Divider");
                    try {
                        div.start();
                        div.join();
                    } catch (Exception e) {
                        System.out.println("Got some problem with making the
    thread!");
                        System.out.println(e);
                    }
                    break;
                case 5:
                    System.out.println("You have chosed to Exit!");
                default:
                    break;
            }

        }
        System.exit(0);
    }
}
```

Listing 1: Probelm 1.java

7.1.1 Java Output

```
Welcome To Thread Calculator of Assignment 7
What would you like to do?
1. Addition of 2 Numbers
2. Subtraction of 2 Numbers
3. Multiplication of 2 Numbers
4. Division of 2 Numbers


1
Enter the 2 Numbers
2
2
You have chosen Addition!
Starting the Thread
The Name of this Thread is: Adder
4
What would you like to do?
1. Addition of 2 Numbers
2. Subtraction of 2 Numbers
3. Multiplication of 2 Numbers
4. Division of 2 Numbers


4
Enter the 2 Numbers
5
0
You have chosen Division!
Starting the Thread
```

```
The Name of this Thread is: Divider
You cant Divide by Zero!
What would you like to do?
1. Addition of 2 Numbers
2. Subtraction of 2 Numbers
3. Multiplication of 2 Numbers
4. Division of 2 Numbers


5
```

Listing 2: Output for Problem 1 - calculations

7.2 Java Implementation of Problem 2

```java
// Krishnaraj Thadesar
// Batch A1, PA20
// OOPCJ Assignment 7.2
// Print even and odd numbers in increasing order using two threads in Java

import java.security.ProtectionDomain;
import java.util.Scanner;

import javax.swing.InputMap;

class printEven extends Thread implements Runnable {
    int limit;

    printEven(int limit) {
        this.limit = limit;
    }

    @Override
    public synchronized void start() {
        super.start();
        System.out.println("Printing Even Numbers");
    }

    @Override
    public void run() {
        for (int i = 0; i < limit; i++) {
            if (i % 2 == 0) {
                System.out.println(i);
            }
        }
    }
}

class printOdd extends Thread implements Runnable {
    int limit;

    printOdd(int limit) {
        this.limit = limit;
    }

    @Override
    public synchronized void start() {
        super.start();
        System.out.println("Printing Odd Numbers");
```

```
    }

    @Override
    public void run() {
        for (int i = 0; i < limit; i++) {
            if (i % 2 != 0) {
                System.out.println(i);
            }
        }
    }
}

public class assignment_7_problem_2 {
    static printEven pe;
    static printOdd po;
    static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int limit = 0;
        System.out.println("Enter To what limit Even or Odd numbers you want to
    See");
        limit = input.nextInt();
        pe = new printEven(limit);
        po = new printOdd(limit);
        try {
            pe.start();
            pe.join();
            po.start();
            po.join();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Listing 3: Multithreading Even Odd

7.2.1 Java Output

```
Enter To what limit Even or Odd numbers you want to See
10
Printing Even Numbers
0
2
4
6
8
Printing Odd Numbers
1
3
5
7
9
```

Listing 4: Output for ProblemHillStation 2

8 Conclusion

Thus, learnt the use of thread class in java and performed multithreading operations.

9 FAQs

1. *What are the challenges for multithreading implementation using Java Programming?*

   (a) **Difficulty of writing code**: Multithreaded and multicontexted applications are not easy to write. Only experienced programmers should undertake coding for these types of applications.

   (b) **Difficulty of debugging**: It is much harder to replicate an error in a multithreaded or multicontexted application than it is to do so in a single-threaded, single-contexted application. As a result, it is more difficult, in the former case, to identify and verify root causes when errors occur.

   (c) **Difficulty of managing concurrency**: The task of managing concurrency among threads is difficult and has the potential to introduce new problems into an application.

   (d) **Difficulty of testing**: Testing a multithreaded application is more difficult than testing a single-threaded application because defects are often timing-related and more difficult to reproduce.

   (e) **Difficulty of porting existing code**: Existing code often requires significant re-architecting to take advantage of multithreading and multicontexting. Programmers need to:

   - Remove static variables
   - Replace any function calls that are not thread-safe
   - Replace any other code that is not thread-safe

2. *What is the difference between the start and run method in Java Thread?*
   start()

   - Creates a new thread and the run() method is executed on the newly created thread.
   - Can't be invoked more than one time otherwise throws java.lang.IllegalStateException
   - Defined in java.lang.Thread class.

   run()

   - No new thread is created and the run() method is executed on the calling thread itself.
   - Multiple invocation is possible.
   - It is just a normal function in Runnable Interface.
   - Defined in java.lang.Runnable interface and must be overriden in the implementing class.

3. *Which one is better to implement thread in Java? extending Thread class or implementing Runnable?*

   (a) The significant differences between extending Thread class and implementing Runnable interface

   (b) When we extend Thread class, we cant extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.

(c) When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.

4. *What is the difference between wait and sleep in Java? Explain with example.*
wait():

- Wait() method belongs to Object class.
- Wait() method releases lock during Synchronization.
- Wait() should be called only from Synchronized context.
- Wait() is not a static method.
- Wait() Has Three Overloaded Methods:
  (a) wait()
  (b) wait(long timeout)
  (c) wait(long timeout, int nanos)

sleep()

- Sleep() method belongs to Thread class.
- Sleep() method does not release the lock on object during Synchronization.
- There is no need to call sleep() from Synchronized context.
- Sleep() is a static method.
- Sleep() Has Two Overloaded Methods:
  (a) sleep(long millis)millis: milliseconds
  (b) sleep(long millis,int nanos) nanos: Nanoseconds

5. *What is the difference between the submit() and execute() method of Executor and ExecutorService in Java? Explain with example.*
submit():

(a) This function executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

(b) Unlike the execute method, this method returns a future. The future object is used to handle the task after the execution has started.

(c) Therefore, when we need the result of the execution, then we can use the submit() method of the future object. In order to get the result, we can use the get() methods on the Future. The get() method returns an object.

```
1         import java.util.concurrent.*;
2 public class Test {
3     public static void main(String[] args) throws Exception
4     {
5         ExecutorService executorService = Executors.newFixedThreadPool(1)
      ;
6         Future obj = executorService.submit(new Callable() {
7                   // Overriding the call method
8                   public Object call()
9                   {
10                      System.out.println(
```

```
                            "This is submit() "
                            + "method example");

                        return "Returning Callable "
                            + "Task Result";
                    }
                });
        System.out.println(obj.get());
        executorService.shutdown();
    }
}

```

execute():

(a) This function executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

(b) This method is a void method meaning it doesn't return any function. Once the task is assigned in the execute() method, we won't get any response and we can forget about the task.

```
import java.util.concurrent.*;
public class Test {

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

    ExecutorService executorService = Executors.newSingleThreadExecutor()
    ;
    executorService.execute(new Runnable() {
      // Override the run method
      public void run()
      {
        System.out.println(
          "This is execute() "
          + "method example");
      }
    });
    executorService.shutdown();
  }
}


```