Object - Oriented Programming 2

Week 3. Multithreading In Java(Introduction, Creating a thread, Joining threads)

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

1. Introduction to Multithreading
2. Creating a thread,
3. Joining threads

Introduction to Java Multithreading

Multithreading is a concept of running multiple threads simultaneously. like many modern programming languages, Java support multi-threaded applications.
In Java, threads of execution are represented by the **java.lang.Thread class**, while code for tasks that are designed to run in a separate thread is represented by the **java.lang.Runnable** interface. It is very important that developers be aware of both, **(**David R.,Michael R. 2002).

Thread is a lightweight unit of a process that executes in multithreading environment.

Introduction to java Multithreading+

When a program is divided into a number of small processes, it is said to be a multi-threaded program. Each small process can be addressed as a single thread (a lightweight process).

Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. **For example,** one thread is **writing content** on a file at the same time another thread is **performing spelling** check.(Studytonigh.com, n.d.).

Thread

In Java, the word **thread** means two different things.

1. An **instance** of **Thread** class, or
2. A thread of execution.

An **instance of Thread** class is just an object, like any other object in java. But **a thread of execution** means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.

Advantage of Multithreading

The advantages of multithreading are;-

1. Multithreading **reduces** the CPU **idle time** that increase overall performance of the system.
2. Since thread is lightweight process then it takes **less memory** and perform **context switching** as well that helps to share the memory and reduce time of switching between threads.

Multitasking

Multitasking is a process of performing multiple tasks simultaneously.

For example, a computer system that perform multiple tasks like:

1. writing data to a file,
2. playing music,
3. downloading file from remote server at the same time,
4. running presentation file while recording a lecture video at the same time.

How is multitasking achieved?

Multitasking can be achieved either by using **multiprocessing** or **multithreading**.

Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.

Why Multithreading ?

Thread has many **advantages** over the **process** to perform multitasking.

Process is **heavy weight**, takes *more memory* and *occupy CPU for longer time* that may lead to performance issue with the system.

To overcome these issue, process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently.

So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.

How Can One Create Thread ?

To create a thread, Java provides a class called **Thread** and an interface called **Runnable** both are located into **java.lang package.**

We can create thread either **by extending Thread class** or **implementing Runnable interface**. Both includes a run method that must be overridden to provide thread implementation.

It is recommended to use Runnable interface if you just want to create a thread but can use Thread class for implementation of other thread functionalities as well, (Studytonight.com,n.d.)

The main thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in **main** method. This thread is known as **main** thread.

Although the **main thread** is automatically created, you can control it by obtaining a reference to it through calling **currentThread()** method.

The main thread+ important things to know

Two important things to know about **main** thread are,

1. It is the thread from which other threads will be produced.
2. It must always be the last thread to finish execution.

The main thread+ calling currentThread()

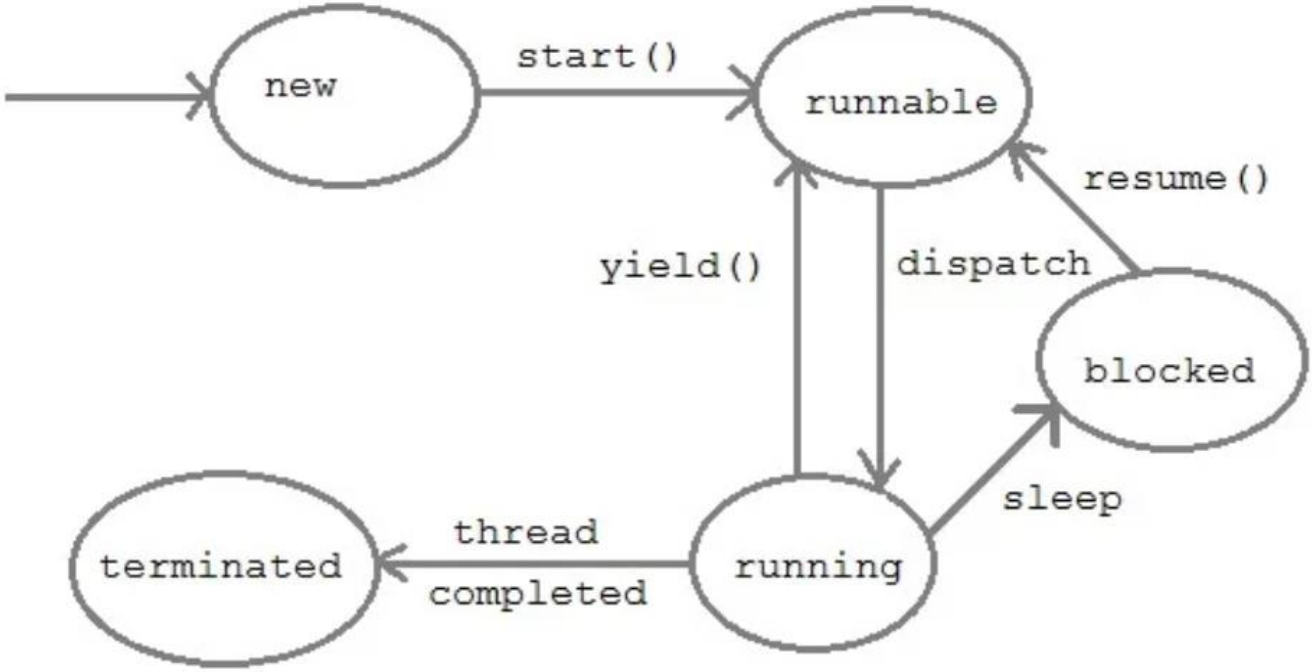
```
class MainThread{
    public static void main(String[] args){
    Thread ob = Thread.currentThread();
    ob.setName("MainThread");
    System.out.println("Name of thread is "+ob);
    }
}
```

(Studytonight.com,n.d.)

```
run:
Name of thread is Thread[MainThread,5,main]
```

Life cycle of a Thread

Like process, thread has a life cycle that includes various phases like: new, running, terminated etc. as see in the figure below.



(studytonight.com/java/multithreading-in-java.php)

Life cycle of a Thread Explained

Like process, thread has a life cycle that includes various phases like: -

1. **New :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable :** After invocation of start() method on new thread, the thread becomes runnable.
3. **Running :** A thread is in running state if the thread scheduler has selected it.
4. **Waiting :** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task

(studytonight.com/java/multithreading-in-java.php)

Daemon Thread

Daemon threads are **low priority threads** that provide supports to user threads. These threads can be user defined and system defined as well.

For example, Garbage collection thread is one of the system generated daemon thread that runs in background.

These threads run in the background to perform **tasks such as garbage collection**. Daemon thread allow JVM from existing until all the threads finish their execution.

Whenever JVM finds daemon threads it terminates the thread and then shutdown itself, it does not care whether Daemon thread is running or not, (Studytonight.com,n.d.)

Thread Creation in Java

Thread Creation in Java

To implement multithreading, Java defines two ways by which a thread can be created.

1. By implementing the **Runnable** interface.
2. By extending the **Thread** class.

Creating a thread by Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the run() method.

Syntax:

```
class MyThreadPro1 implements Runnable { }

public void run()
```

Run Method Syntax

Three things we need to know about the run() are: -

1. It introduces a concurrent thread into your program. This thread will end when run() method terminates.
2. You must specify the code that your thread will execute inside run() method.
3. run() method can call other methods, can use other classes and declare variables just like any other normal method.

Example1

```
class MyThreadPro1 implements Runnable {
     public void run() {
     System.out.println("concurrent thread started
running..");
     }
}
```

```
class MyThreadPro2 {
     public static void main(String args[]){
     MyThreadPro1 mt = new MyThreadPro1();
     Thread ob = new Thread(mt);
     ob.start(); //executes run() of MyThreadPro1 Class
     }
}
```

Output

```
package Multithreading;
public class MyThreadPro1 implements Runnable {
    public void run() {
        System.out.println("Concurrent thread started running.
        }
}
```

```
2   package Multithreading;
3   public class MyThreadPro2 {
4       public static void main(String args[]){
5           MyThreadPro1 mt = new MyThreadPro1();
6           Thread ob = new Thread(mt);
7           ob.start();
8           }
9   }
```

```
run:
Concurrent thread started running..
```

Note that start() method of Thread class is called inside MyThreadPro2 class is able to execute the run() method of MyThreadPro1 class

Key Notes

start() method is used to call the run() method. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

If you are implementing Runnable interface in your class, then you need to explicitly create a Thread class object and need to pass the Runnable interface implemented class object as a parameter in its constructor.

Creating a thread by Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override run() method which is the entry point of new thread.

```
class MyThreadExtend extends Thread {
        public void run(){
                System.out.println("concurrent thread started running..");
        }
}
```

```
Class MyThreadExtend1{
        public static void main(String args[]){
        MyThreadExtend ob = new MyThreadExtend();
        ob.start();
        }
}
```

```
run:
Concurrent thread started running..
```

Can we Start a thread twice?

No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

```
public static void main(String args[]) {
     MyThread mt = new MyThread();
     mt.start();
     mt.start(); //Exception thrown
}
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using **start()** method, exception is thrown.

Thread Pool

Thread pool is agroup of thread created. It is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing.

From a pool, one thread is selected and assigned a job and after completion of job, it is sent back into the group.

Thread Pool- Three Methods of Thread pool

There are three methods of a Thread pool as seen below:

1. newFixedThreadPool(int)

2. newCachedThreadPool()

3. newSingleThreadExecutor()

Thread Pool-Steps for creating a program of the thread pool

The Following are the steps for creating a program of the thread pool

1. Create a runnable object to execute.
2. Using executors, create an executor pool
3. Now Pass the object to the executor pool
4. At last shutdown the executor pool.

Example:Thread Pool

```
import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;
class WorkerThread implements Runnable{
	private String message;
	public WorkerThread(String a){
	this.message=a;
	}
	public void run() {
	System.out.println(Thread.currentThread().getName()+" (Start) message =
"+message);
	processmessage();
	System.out.println(Thread.currentThread().getName()+" (End)");
	}
	private void processmessage(){
		try {
		Thread.sleep(5000);
		}
		catch (InterruptedException e) {
		System.out.println(e);
		}
	} }
```

Example:Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolPro {
	public static void main(String[] args){
	ExecutorService ex = Executors.newFixedThreadPool(5);
	for (int i = 0; i < 10; i++) {
		Runnable obj = new WorkerThread("" + i);
		ex.execute(obj);
		}
		ex.shutdown();
		while (!ex.isTerminated()) { }
		System.out.println("********All threads are Finished********");
	}
}
```

Example: Thread Pool-Output

```
run:
pool-1-thread-2 (Start) message = 1
pool-1-thread-1 (Start) message = 0
pool-1-thread-3 (Start) message = 2
pool-1-thread-4 (Start) message = 3
pool-1-thread-5 (Start) message = 4
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
```

```
pool-1-thread-3 (Start) message = 7
pool-1-thread-4 (End)
pool-1-thread-5 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (Start) message = 9
pool-1-thread-3 (End)
pool-1-thread-4 (End)
pool-1-thread-2 (End)
pool-1-thread-5 (End)
pool-1-thread-1 (End)
********All threads are Finished********
```

Thread Priorities

In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler.

The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between **1** to **10** and there are three variables which are static to define priority in a Thread Class.

Note: Thread priorities *cannot guarantee* that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

Joining threads in Java

Joining threads in Java

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

```
final boolean isAlive()
```

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

Java join() method

```
final void join() throws InterruptedException
```

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which we want to wait for the specified thread to terminate.

```
final void join(long milliseconds) throws InterruptedException
```

Java isAlive() method

Lets take an example and see how the isAlive() method works. It returns true if thread status is live, false otherwise.

```
public class MyThreadisAlive extends Thread {
        public void run() {
        System.out.println("r1 ");
        try {
                Thread.sleep(500);
        }
        catch(InterruptedException ie) {/* do something*/ }
        System.out.println("r2 ");
        }
        public static void main(String[] args) {
                MyThreadisAlive t1=new MyThreadisAlive();
                MyThreadisAlive t2=new MyThreadisAlive();
                t1.start();
                t2.start();
                System.out.println(t1.isAlive());
                System.out.println(t2.isAlive());
} }
```

Java isAlive() method-Output

```
1   package Multithreading;
2   public class MyThreadisAlive extends Thread{
3       public void run() {
4           System.out.println("r1 ");
5           try {
6               Thread.sleep(500);
7           }
8           catch(InterruptedException ie) { /* do something*/ }
9           System.out.println("r2 ");
10          }
11          public static void main(String[] args){
12              MyThreadisAlive t1=new MyThreadisAlive();
13              MyThreadisAlive t2=new MyThreadisAlive();
14              t1.start();
15              t2.start();
16              System.out.println(t1.isAlive());
17              System.out.println(t2.isAlive());
18          }
19  }
```

```
run:
true
true
r1
r1
r2
r2
```

Example of thread without join() method

If we run a thread without using join() method then the execution of thread cannot be predict. Thread scheduler schedules the execution of thread.

```
public class MyThreadNoJoin extends Thread {
	public void run() {
		System.out.println("Thread1 ");
		try {
		Thread.sleep(500);
		}
		catch(InterruptedException ie){ }
		System.out.println("Thread2 ");
	}
	public static void main(String[] args) {
		MyThreadNoJoin ob1=new MyThreadNoJoin();
		MyThreadNoJoin ob2=new MyThreadNoJoin();
		ob1.start();
		ob2.start();
	}
}
```

Example of thread without join() method- Output

```
1   package Multithreading;
2   public class MyThreadNoJoin extends Thread {
3           public void run() {
4                   System.out.println("Thread1 ");
5                   try {
6                   Thread.sleep(500);
7                   }
8                   catch(InterruptedException ie){ }
9                   System.out.println("Thread2 ");
10          }
11          public static void main(String[] args) {
12                  MyThreadNoJoin ob1=new MyThreadNoJoin();
13                  MyThreadNoJoin ob2=new MyThreadNoJoin();
14                  ob1.start();
15                  ob2.start();
16          }
17  }
```

```
run:
Thread1
Thread1
Thread2
Thread2
```

How it works

In the above program, two thread ob1 and ob2 are created. ob1 starts first and after printing “Thread1" on console thread ob1 goes to sleep for 500 ms. At the same time Thread ob2 will start its process and print “Thread1" on console and then go into sleep for 500 ms. Thread ob1 will wake up from sleep and print “Thread2" on console similarly thread ob2 will wake up from sleep and print “thread2" on console. *Thus the following output :-*

Thread1

Thread1

Thread2

Thread2

Example of thread with join() method

In this example, we are using join() method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

```
public class MyThreadWithJoin extends Thread {
      public void run() {
      System.out.println("Thread1 ");
      try { Thread.sleep(600);
      }
      catch(InterruptedException ie){ }
            System.out.println("Thread2 ");
      }
      public static void main(String[] args){
      MyThreadWithJoin ob1=new MyThreadWithJoin();
      MyThreadWithJoin ob2=new MyThreadWithJoin();
      ob1.start();
      try{
      ob1.join(); /*Waiting for t1 to finish*/  }
      catch(InterruptedException ie){}
      ob2.start();
} }
```

Example of thread with join() method

```
1   package Multithreading;
2   public class MyThreadWithJoin extends Thread {
3       public void run() {
4           System.out.println("Thread1 ");
5           try { Thread.sleep(600);
6           }
7           catch(InterruptedException ie){ }
8                   System.out.println("Thread2 ");
9       }
10      public static void main(String[] args){
11          MyThreadWithJoin ob1=new MyThreadWithJoin();
12          MyThreadWithJoin ob2=new MyThreadWithJoin();
13          ob1.start();
14          try{
15          ob1.join(); /*Waiting for t1 to finish*/ }
16          catch(InterruptedException ie){}
17          ob2.start();
18      }
19  }
```

In the above program join() method on thread ob1 ensures that it finishes its process by printing Thread1 and Thread2 before thread ob2 starts its own process then print Thread1 and Thread 2 thus the output below.

```
run:
Thread1
Thread2
Thread1
Thread2
```

Specifying time with join()

If in the above program, we specify time while using **join()** with **ob 1**, then **ob1** will execute for that time, and then **ob2** will join it.

Note that, initially ob1 will execute for 1.5 seconds, after which ob2 will join it.

```
public class MyThreadTime extends Thread{
        MyThreadTime(String str){
        super(str);
        }
        public void run() {
        System.out.println(Thread.currentThread().getName());
        }
        public static void main(String[] args) {
        MyThreadTime ob1=new MyThreadTime("Thread 1 ");
        MyThreadTime ob2=new MyThreadTime("Thread 2 ");
        ob1.start();
        try{ob1.join(1500); //Waiting for t1 to finish
        }
        catch(InterruptedException ie){
        System.out.println(ie);
        }
         ob2.start();
        try{ob2.join(1500); //Waiting for t2 to finish
        }catch(InterruptedException ie){
        System.out.println(ie);
} } }
```

Specifying time with join()-Output

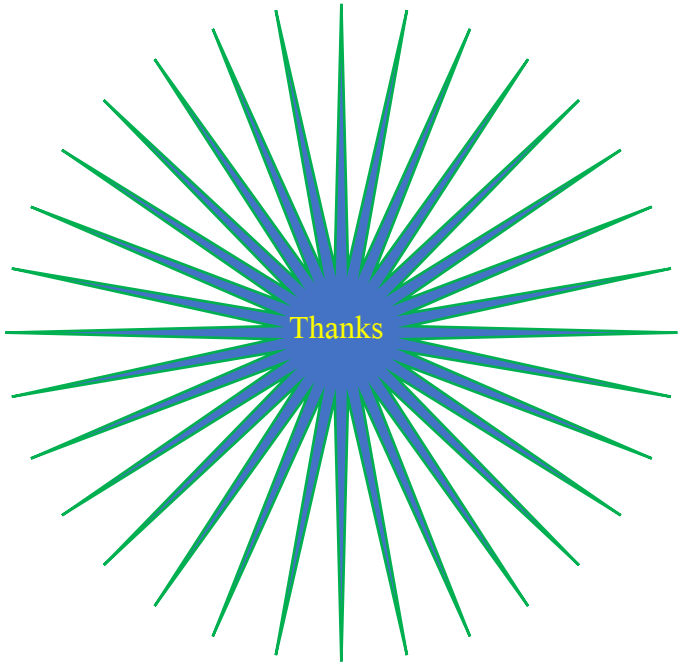
```
1   package Multithreading;
2   public class MyThreadTime extends Thread{
3       MyThreadTime(String str){
4           super(str); }
5       public void run() {
6           System.out.println(Thread.currentThread().getName());
7       }
8       public static void main(String[] args) {
9           MyThreadTime ob1=new MyThreadTime("Thread 1 ");
10          MyThreadTime ob2=new MyThreadTime("Thread 2 ");
11          ob1.start();
12          try{ob1.join(1500); }
13          catch(InterruptedException ie){
14          System.out.println(ie); }
15           ob2.start();
16          try{ob2.join(1500); /*Waiting for t2 to finish*/
17          catch(InterruptedException ie){
18          System.out.println(ie);
19  } } }
```

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

Summary

1. Introduction to Multithreading(dinitions, thread, multitasking, advantages of Multithreading)
2. Creating a thread (bying extending thread class, by implementing Runnable interface)
3. Joining threads(shared much on isAlive() and join() etc.)

Thank you for Listening



References

Java™ Network Programming and Distributed Computing, (David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374

Introduction to multithreading in Java. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/multithreading-in-java.php

Java thread class. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/thread-class-and-functions.php

Creating a thread in Java. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/creating-a-thread.php