ThreadFactory Interface in Java with Examples

Last Updated : 24 Jun, 2021

The **ThreadFactory** interface defined in the **java.util.concurrent** package is based on the factory design pattern. As its name suggests, it is used to create new threads on demand. Threads can be created in two ways:

1. Creating a class that extends the Thread class and then creating its objects.

Java

```
import java.io.*;

class GFG {

    public static void main(String[] args)
    {
        // Creating a thread
        Thread thread = new CustomThread();
        thread.start(); // Starting execution of the created
                        // thread
    }
}

// Creating a class that extends the Thread class
class CustomThread extends Thread {
    @Override public void run()
    {
        System.out.println("This is a thread");
    }
}
```

Output

```
This is a thread
```

2. Creating a class that implements the Runnable interface and then using its object to create threads.

Java

```java
/*package whatever //do not write package name here */

import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        // Creating a Runnable object
        Runnable task = new Task();
        // Creating a thread using the Runnable object
        Thread thread = new Thread(task);
        // Starting the execution of the created thread
        thread.start();
    }
}

class Task implements Runnable {
    @Override public void run()
    {
        System.out.println("This is a thread");
    }
}
```

Output

```
This is a thread
```

However, **ThreadFactory** is another choice to create new threads. This interface provides a factory method that creates and returns new threads when called. This factory method takes a Runnable object as an argument and creates a new thread using it.

The Hierarchy of ThreadFactory

```
java.util.concurrent
    ↳ Interface ThreadFactory
```

Implementation of ThreadFactory interface

Since ThreadFactory is an interface, the factory method defined inside it has to be implemented first in order to be used. Here is the simplest implementation of the ThreadFactory interface :

Java

```
import java.util.concurrent.ThreadFactory;
import java.io.*;

class CustomThreadFactory implements ThreadFactory {

    // newThread is a factory method
    // provided by ThreadFactory
    public Thread newThread(Runnable command)
    {
        return new Thread(command);
    }
}
```

Now, we can create objects of the **CustomThreadFactory** class and use its newThread(Runnable command) method to create new threads on demand. In the above implementation, the newThread method just creates a new thread by calling the Thread constructor which takes a Runnable command as the parameter.

There are many classes(such as ScheduledThreadPoolExecutor , **ThreadPoolExecutor** etc.) that use thread factories to create new threads when needed. Those classes have constructors that accept a ThreadFactory as

argument. If any custom ThreadFactory is not given then they use the default implementation of ThreadFactory interface.

The **Executors** class in java.util.concurrent package provides **Executors.defaultThreadFactory()** static method that returns a default implementation of ThreadFactory interface.

Example: Below example code demonstrates ThreadFactory interface.

Java

```
// Java code to demonstrate ThreadFactory interface

import java.util.concurrent.ThreadFactory;
import java.io.*;
import java.util.ArrayList;

class ThreadFactoryExample {
    public static void main(String[] args)
    {
        // Creating a CustomThreadFactory object
        CustomThreadFactory threadFactory
            = new CustomThreadFactory();

        // Creating Runnable objects using the lambda
        // expression
        Runnable command1 = ()
            -> System.out.println("Command 1 executed");
        Runnable command2 = ()
            -> System.out.println("Command 2 executed");
        Runnable command3 = ()
            -> System.out.println("Command 3 executed");
        Runnable command4 = ()
            -> System.out.println("Command 4 executed");
        Runnable command5 = ()
            -> System.out.println("Command 5 executed");

        // Putting the commands in an ArrayList
        ArrayList<Runnable> array = new ArrayList<>(5);
        array.add(command1);
        array.add(command2);
        array.add(command3);
        array.add(command4);
        array.add(command5);

        // creating threads and running them
```

```
            for (Runnable command : array) {
                threadFactory.newThread(command).start();
            }

            // print the thread count
            System.out.println(
                "Total number of threads created using CustomThreadFactory = "
                + threadFactory.getCount());
        }
    }

    // ThreadFactory class
    class CustomThreadFactory implements ThreadFactory {

        // stores the thread count
        private int count = 0;

        // returns the thread count
        public int getCount() { return count; }

        // Factory method
        @Override
          public Thread newThread(Runnable command)
        {
            count++;
            return new Thread(command);
        }
    }
```