**How will you create your own thread pool in Java**

**import** java.util.concurrent.BlockingQueue;  
**import** java.util.concurrent.LinkedBlockingQueue;  
  
**public class CustomThreadPool** {  
 **private BlockingQueue<Runnable> blockingQueue;**  
 **private boolean shutdown** = **false**;  
  
 **public** CustomThreadPool(**int** nThreads) {  
 **blockingQueue** = **new LinkedBlockingQueue<>(nThreads);** **for** (**int** i = 0; i < nThreads; i++) {  
 **TaskExecutor executor = new TaskExecutor**(**this**, **blockingQueue**);  
 **executor.setName("Thread-Pool-" + i);  
 executor.start();** }  
 }  
  
 **public synchronized void** execute(Runnable runnable) **throws** Exception {  
 **blockingQueue.put(runnable);**  
 }  
  
 **public void** shutdown() {  
 **this**.**shutdown** = **true**;  
 }  
  
 **public boolean** isShutdown() {  
 **return shutdown**;  
 }  
}

**import** java.util.concurrent.BlockingQueue;  
  
**public class TaskExecutor** **extends** Thread {  
 **private BlockingQueue<Runnable> blockingQueue;  
 private CustomThreadPool threadPool;**  
 **public** TaskExecutor(CustomThreadPool threadPool, BlockingQueue<Runnable> blockingQueue) {  
 **this**.**blockingQueue** = blockingQueue;  
 **this**.**threadPool** = threadPool;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 **while** (**true**) {  
 **Runnable runnable = blockingQueue.take();  
 runnable.run();** System.***out***.println(Thread.*currentThread*().getName() + **" executed ..."**);  
  
 **if (blockingQueue.size() == 0 && threadPool.isShutdown()) {  
 this.interrupt();  
 }** }  
 } **catch** (InterruptedException ie) {  
 System.***out***.println(Thread.*currentThread*().getName() + **" has been stopped ..."**);  
 }  
 }  
}

Test Program is given below about how to test.

**import** java.util.concurrent.TimeUnit;  
  
**public class TestThread** **extends** Thread {  
 @Override  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName() + **" running ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(3);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed"**);  
 }  
}

**public class TestThreadPool** {  
 **public static void** main(String[] args) {  
 CustomThreadPool threadPool = **new** CustomThreadPool(3);  
 **for** (**int** i = 0; i < 3; i++) {  
 Thread thread = **new** TestThread();  
 **try** {  
 threadPool.execute(thread);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 threadPool.shutdown();  
 }  
}

OUTPUT

Thread-Pool-0 running ...

Thread-Pool-2 running ...

Thread-Pool-1 running ...

Thread-Pool-1 completed

Thread-Pool-2 completed

Thread-Pool-2 executed ...

Thread-Pool-0 completed

Thread-Pool-0 executed ...

Thread-Pool-1 executed ...

Thread-Pool-0 has been stopped ...

Thread-Pool-2 has been stopped ...

Thread-Pool-1 has been stopped ...

**Better way of ThreadPool Creation**

import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.BlockingQueue;  
import java.util.concurrent.LinkedBlockingQueue;  
  
public class CustomThreadPool {  
  
 *// holds tasks***private BlockingQueue<Runnable> runnableQueue;**  
 *// holds the pool of worker threads* **private List<WorkerThread> threads;**  
 *// check if shutdown is initiated* **private boolean isShutDown;**  
  
 public CustomThreadPool(final int noOfThreads) {  
 this.runnableQueue = new LinkedBlockingQueue<>();  
 this.threads = new ArrayList<>(noOfThreads);  
 this.isShutDown = false;  
 *// create worker threads* **for (int i = 1; i <= noOfThreads; i++) {  
 WorkerThread thread = new WorkerThread(runnableQueue, this);  
 thread.setName("Worker Thread - " + i);  
 thread.start();  
 threads.add(thread);  
 }** }  
  
 public void execute(Runnable r) throws InterruptedException {  
 **if (!isShutDown) {  
 runnableQueue.put(r);  
 } else {  
 throw new InterruptedException("Thread Pool shutdown is initiated, unable to execute task");  
 }** }  
  
 **public void shutdown() {  
 isShutDown = true;  
 }**

**private class WorkerThread extends Thread** {  
 *// holds tasks* **private BlockingQueue<Runnable> runnableQueue;**  
 *// check if shutdown is initiated***private CustomThreadPool threadPool**;  
  
 **public WorkerThread(BlockingQueue<Runnable> taskQueue, CustomThreadPool threadPool) {  
 this.runnableQueue = taskQueue;  
 this.threadPool = threadPool;  
 }**  
 @Override  
 public void run() {  
 try {  
 *// continue until all tasks finished processing* **while (!threadPool.isShutDown || !runnableQueue.isEmpty()) {  
 Runnable runnable;  
 *// Poll a runnable from the queue and execute it* while ((runnable = runnableQueue.poll()) != null) {  
 runnable.run();  
 }  
 Thread.*sleep*(1);  
 }** } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
}

**Test Class**

public class CustomThreadPoolTest {  
  
 public static void main(String[] args) throws InterruptedException {  
 Runnable r = () -> {  
 try {  
 Thread.*sleep*(5000);  
 System.*out*.println(Thread.*currentThread*().getName() + " is executing task.");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 };  
  
 CustomThreadPool threadPool = new CustomThreadPool(2);  
  
 threadPool.execute(r);  
 threadPool.execute(r);  
 threadPool.shutdown();  
 }  
  
}

**How to create background thread**

A background thread is created using **Thread.setDemon(true)**. There are two types of threads one is user thread and another is demon/background/low priority thread. Main application cannot exit if the user threads are running, but in case of demon threads, if the main application exits, demon threads get killed/stopped automatically. Demon threads are useful for house keeping works like log rotating, log file movement, deleting old temp files etc.

Code is given below.

**import** java.util.concurrent.TimeUnit;  
**public class RunnableTask** **implements** Runnable {  
  
 @Override  
 **public void** run() {  
 **try** {  
 **while** (**true**) {  
 System.***out***.println(Thread.*currentThread*().getName() + **" running ..."**);  
 TimeUnit.***SECONDS***.sleep(3);  
 }  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.TimeUnit;  
**public class TestDemonThread1** {  
 **public static void** main(String[] args) **throws** Exception {  
 Thread t1 = **new** Thread(**new** RunnableTask(), **"Demon-1"**);  
 t1.setDaemon(**true**); 🡸Demon thread.  
 t1.start();  
  
 TimeUnit.***SECONDS***.sleep(10);  
 System.***out***.println(**"All tasks done ..."**);  
 }  
}

In this case, main thread/application exits after 10 seconds and all the demon threads get killed/stopped.

If it had been user thread, if the main application exits, still all the user threads will continue to run.

**Use of ThreadLocal in Java Multithreading**

Let us see first, if we do not use ThreadLocal, what will happen. There is a List of animal, one thread will iterate the list, another thread will remove the data from the list and another thread will add the data to the list.

Code is given below.

**import** java.util.Iterator;  
**import** java.util.List;  
  
**public class Thread1** **extends** Thread {  
 **private** List<String> **list**;  
  
 **public** Thread1(List<String> list) {  
 **this**.**list** = list;  
 }  
@Override  
 **public void** run() {**for** (Iterator<String> itr = **list**.iterator(); itr.hasNext(); ) {  
 String value = itr.next();  
 System.***out***.println(**"Now Value : "** + value);  
 }  
 }  
}

**import** java.util.Iterator;  
**import** java.util.List;  
  
**public class Thread2** **extends** Thread {  
 **private** List<String> **list**;  
  
 **public** Thread2(List<String> list) {  
 **this**.**list** = list;  
 }  
@Override  
 **public void** run() {**for** (**int** i = 0; i < 10; i++) {  
 **list**.add(**"Add-Value-"** + i);  
 }  
 }  
}

**import** java.util.List;  
  
**public class Thread3** **extends** Thread {  
 **private** List<String> **list**;  
  
 **public** Thread3(List<String> list) {  
 **this**.**list** = list;  
 }  
@Override  
 **public void** run() {**for** (**int** i = 0; i < 2; i++) {  
 **list**.remove(i);  
 }  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
**public class Test1** {  
 **public static void** main(String[] args) **throws** Exception {  
 List<String> list =  
 **new** ArrayList<>(Arrays.*asList*(**"Elephant"**, **"Tiger"**, **"Lion"**, **"Rat"**, **"Deer"**, **"Zebra"**));  
 Thread t1 = **new** Thread1(list);  
 Thread t2 = **new** Thread2(list);  
 Thread t3 = **new** Thread3(list);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 t1.join();  
 t2.join();  
 t3.join();  
 System.***out***.println(**"All threads completed operations ..."**);  
 }  
}

OUTPUT

Now Value : Elephant

Exception in thread "Thread-0" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)

at java.util.ArrayList$Itr.next(ArrayList.java:859)

at com.ddlab.rnd.threadlocal.Thread1.run(Thread1.java:23)

All threads completed operations ..., Here we found ConcurrentModificationException.

Let us see below the modified code.

**import** java.util.ArrayList;  
**import** java.util.List;  
**public class ThreadLocaList** {  
 **private** List<String> **list**;  
  
 **public** ThreadLocaList(List<String> list) {  
 **this**.**list** = list;  
 }  
  
 **private** ThreadLocal<List<String>> **tLocal** = **new** ThreadLocal() {  
 **protected** List<String> initialValue() {  
 **return new ArrayList(list);**  
 }  
 };  
  
 **public** List<String> getList() {  
 **return tLocal.get();** }  
}

**import** java.util.Iterator;  
**import** java.util.List;  
**public class Thread1** **extends** Thread {  
 **private** ThreadLocaList **thList**;  
 **public** Thread1(ThreadLocaList thList) {  
 **this**.**thList** = thList;  
 }  
  
 @Override  
 **public void** run() {  
 List<String> tList = **thList**.getList();  
 **for** (Iterator<String> itr = tList.iterator(); itr.hasNext(); ) {  
 String value = itr.next();  
 System.***out***.println(**"Now Value : "** + value);  
 }  
 }  
}

**import** java.util.List;  
**public class Thread2** **extends** Thread {  
 **private** ThreadLocaList **thList**;  
 **public** Thread2(ThreadLocaList thList) {  
 **this**.**thList** = thList;  
 }  
  
 @Override  
 **public void** run() {  
 List<String> tList = **thList**.getList();  
 **for** (**int** i = 0; i < 10; i++)

tList.add(**"Add-Value-"** + i);  
 }  
}

**import** java.util.List;  
**public class** Thread3 **extends** Thread {  
 **private** ThreadLocaList **thList**;  
 **public** Thread3(ThreadLocaList thList) {  
 **this**.**thList** = thList;  
 }  
  
 @Override  
 **public void** run() {  
 List<String> tList = **thList**.getList();  
 **for** (**int** i = 0; i < 2; i++) {  
 tList.remove(i);  
 }  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
  
**public class Test2** {  
 **public static void** main(String[] args) **throws** Exception {  
 List<String> list =  
 **new** ArrayList<>(Arrays.*asList*(**"Elephant"**, **"Tiger"**, **"Lion"**, **"Rat"**, **"Deer"**, **"Zebra"**));  
 ThreadLocaList thList = **new** ThreadLocaList(list);  
 Thread t1 = **new** Thread1(thList);  
 Thread t2 = **new** Thread2(thList);  
 Thread t3 = **new** Thread3(thList);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 t1.join();  
 t2.join();  
 t3.join();  
 System.***out***.println(**"All threads completed operations ..."**);  
 }  
}

OUTPUT

Now Value : Elephant

Now Value : Tiger

Now Value : Lion

Now Value : Rat

Now Value : Deer

Now Value : Zebra

All threads completed operations ...

In this case, it does not throw any exception as each thread gets a local copy of List from ThreadLocal.

**If you want, you pass as new ArrayList(list) and it works.**

**Do not use *ThreadLocal* with *ExecutorService***

If we want to use an *ExecutorService*and submit a *Runnable*to it, using *ThreadLocal*will yield non-deterministic results – because we do not have a guarantee that every *Runnable*action for a given *userId* will be handled by the same thread every time it is executed.

Because of that, our *ThreadLocal* will be shared among different *userIds.* That’s why we should not use a *TheadLocal* together with *ExecutorService.* It should only be used when we have full control over which thread will pick which runnable action to execute.

**Use of ThreadLocal in Java Multithreading**

Let us see first, if we do not use ThreadLocal, what will happen.

**public class** Check1 {  
 **public void** add(List<String> list) {  
 **for** (**int** i = 0; i < 10; i++) {  
 list.add(**"Add-Value-"** + i);  
 }  
 }  
  
 **public void** remove(List<String> list) {  
 **for** (**int** i = 0; i < 2; i++) {  
 list.remove(i);  
 }  
 }  
  
 **public void** show(List<String> list) {  
 **for** (Iterator<String> itr = list.iterator(); itr.hasNext(); ) {  
 String value = itr.next();  
 System.***out***.println(**"Now Value : "** + value);  
 }  
 }  
  
 **public void** check() **throws** InterruptedException {  
 List<String> alphabets = Arrays.*asList*(**"A"**, **"B"**, **"C"**, **"D"**, **"E"**, **"F"**);  
 List<String> list = **new** ArrayList<>(alphabets);

Runnable add = () -> add(list);  
 Runnable remove = () -> remove(list);  
 Runnable show = () -> show(list);  
  
 Thread t1 = **new** Thread(add);  
 Thread t2 = **new** Thread(remove);  
 Thread t3 = **new** Thread(show);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 t1.join();  
 t2.join();  
 t3.join();  
 System.***out***.println(**"All threads completed operations ..."**);  
  
 }

**OUTPUT**

Now Value : B

All threads completed operations ...

Exception in thread "Thread-2" java.util.ConcurrentModificationException

at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)

at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)

at com.ddlab.rnd.type1.Check1.show(Check1.java:24)

at com.ddlab.rnd.type1.Check1.lambda$check$2(Check1.java:34)

at java.base/java.lang.Thread.run(Thread.java:834)

Let use a modified program using ThreadLocal.

Create a class extending ThreadLocal and use it other classes.

**public class** MyThreadLocal **extends** ThreadLocal<List<String>> {  
  
 **private** List<String> **list**;  
  
 **public** MyThreadLocal(List<String> list) {  
 **this**.**list** = list;  
 }  
  
 @Override  
 **protected** List<String> initialValue() {  
 **return new** ArrayList<String>(**list**);  
 }  
}

**public class** Check1 {  
 **public void** add(MyThreadLocal thList) {  
 List<String> list = thList.get();  
 **for** (**int** i = 0; i < 10; i++) {  
 list.add(**"Add-Value-"** + i);  
 }  
 }  
  
 **public void** remove(MyThreadLocal thList) {  
 List<String> list = thList.get();  
 **for** (**int** i = 0; i < 2; i++) {  
 list.remove(i);  
 }  
 }  
  
 **public void** show(MyThreadLocal thList) {  
 List<String> list = thList.get();  
 **for**(String s: list)  
 System.***out***.println(**"Now Value : "** +s);

}  
  
 **public void** check() **throws** InterruptedException {  
 List<String> alphabets = Arrays.*asList*(**"A"**, **"B"**, **"C"**, **"D"**, **"E"**, **"F"**);  
 List<String> list = **new** ArrayList<>(alphabets);  
 MyThreadLocal threadLocal = **new** MyThreadLocal(list);  
 Runnable add = () -> add(threadLocal);  
 Runnable remove = () -> remove(threadLocal);  
 Runnable show = () -> show(threadLocal);  
  
 Thread t1 = **new** Thread(add);  
 Thread t2 = **new** Thread(remove);  
 Thread t3 = **new** Thread(show);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 t1.join();  
 t2.join();  
 t3.join();  
 System.***out***.println(**"All threads completed operations ..."**);  
 }  
  
 **public static void** main(String[] args) **throws** Exception {  
 **new** Check1().check();  
 }  
}

In this case, it does not throw any exception as each thread gets a local copy of List from ThreadLocal.

**Do not use *ThreadLocal* with *ExecutorService***

If we want to use an *ExecutorService*and submit a *Runnable*to it, using *ThreadLocal*will yield non-deterministic results – because we do not have a guarantee that every *Runnable*action for a given *userId* will be handled by the same thread every time it is executed.

Because of that, our *ThreadLocal* will be shared among different *userIds.* That’s why we should not use a *TheadLocal* together with *ExecutorService.* It should only be used when we have full control over which thread will pick which runnable action to execute.