Java Concurrency - Quick Guide

Previous Page Next Page

Java Concurrency - Overview

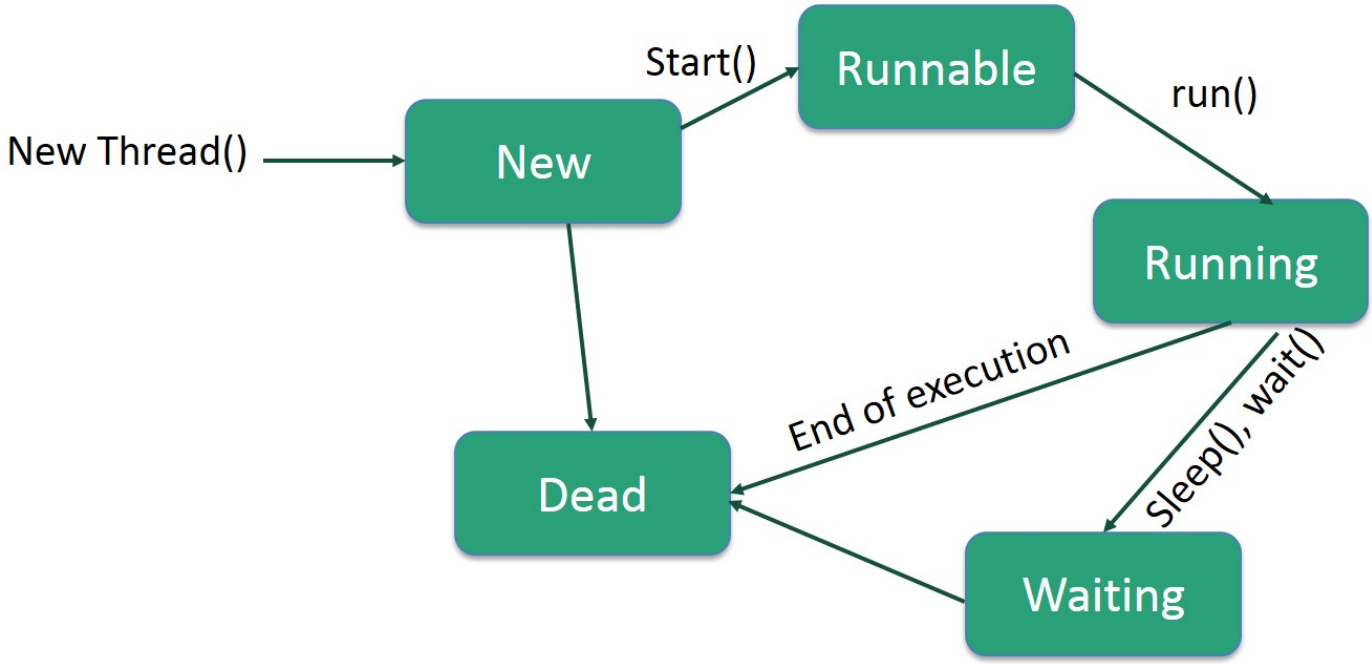
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle –

New – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.

Runnable – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee

the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps –

Step 1

As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method –

```
public void run( )
```

Step 2

As a second step, you will instantiate a **Thread** object using the following constructor –

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method –

```
void start();
```

Example

Here is an example that creates a new thread and starts running it –

Live Demo

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo(String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );

      try {

         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);

            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
```

```
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}

public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo("Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo("Thread-2");
      R2.start();
   }
}
```

This will produce the following result –

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling

multiple threads created using available methods in Thread class.

Step 1

You will need to override **run( )** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of start() method −

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread −

Live Demo

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo(String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );

      try {

         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);

            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
public class TestThread {

   public static void main(String args[]) {
      ThreadDemo T1 = new ThreadDemo("Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo("Thread-2");
      T2.start();
   }
}
```

This will produce the following result –

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Java Concurrency - Environment Setup

In this chapter, we will discuss on the different aspects of setting up a congenial environment for Java.

Local Environment Setup

If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Following are the steps to set up the environment.

Java SE is freely available from the link Download Java . You can download a version based on your operating system.

Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you will need to set environment variables to point to correct installation directories –

Setting Up the Path for Windows

Assuming you have installed Java in *c:\Program Files\java\jdk* directory –

Right-click on 'My Computer' and select 'Properties'.

Click the 'Environment variables' button under the 'Advanced' tab.

Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation, if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH = /path/to/java:$PATH'

Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following –

Notepad – On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.

Netbeans – A Java IDE that is open-source and free which can be downloaded from https://netbeans.org/index.html .

Eclipse – A Java IDE developed by the eclipse open-source community and can be downloaded from https://www.eclipse.org/ .

Java Concurrency - Major Operations

Core Java provides complete control over multithreaded program. You can develop a multithreaded program which can be suspended, resumed, or stopped completely based on your requirements. There are various static methods which you can use on thread objects to control their behavior. Following table lists down those methods –

Sr.No.	Method & Description
1	**public void suspend()** This method puts a thread in the suspended state and can be resumed using resume() method.

2	**public void stop()** This method stops a thread completely.
3	**public void resume()** This method resumes a thread, which was suspended using suspend() method.
4	**public void wait()** Causes the current thread to wait until another thread invokes the notify().
5	**public void notify()** Wakes up a single thread that is waiting on this object's monitor.

Be aware that the latest versions of Java has deprecated the usage of suspend( ), resume( ), and stop( ) methods and so you need to use available alternatives.

Example

Live Demo

```
class RunnableDemo implements Runnable {
   public Thread t;
   private String threadName;
   boolean suspended = false;

   RunnableDemo(String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );

      try {

         for(int i = 10; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);

            // Let the thread sleep for a while.
            Thread.sleep(300);

            synchronized(this) {

               while(suspended) {
                  wait();
               }
            }
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
```

```
   }

   public void start () {
      System.out.println("Starting " +  threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

   void suspend() {
      suspended = true;
   }

   synchronized void resume() {
      suspended = false;
      notify();
   }
}

public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo("Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo("Thread-2");
      R2.start();

      try {
         Thread.sleep(1000);
         R1.suspend();
         System.out.println("Suspending First Thread");
         Thread.sleep(1000);
         R1.resume();
         System.out.println("Resuming First Thread");

         R2.suspend();
         System.out.println("Suspending thread Two");
         Thread.sleep(1000);
         R2.resume();
         System.out.println("Resuming thread Two");
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      } try {
         System.out.println("Waiting for threads to finish.");
         R1.t.join();
         R2.t.join();
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      }
      System.out.println("Main thread exiting.");
   }
}
```

The above program produces the following output –

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 10
Running Thread-2
Thread: Thread-2, 10
Thread: Thread-1, 9
Thread: Thread-2, 9
Thread: Thread-1, 8
Thread: Thread-2, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Suspending thread Two
Thread: Thread-1, 6
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```

Interthread Communication

If you are aware of interprocess communication then it will be easy for you to understand interthread communication. Interthread communication is important when you develop an application where two or more threads exchange some information.

There are three simple methods and a little trick which makes thread communication possible. All the three methods are listed below –

Sr.No.	Method & Description

1	**public void wait()** Causes the current thread to wait until another thread invokes the notify().
2	**public void notify()** Wakes up a single thread that is waiting on this object's monitor.
3	**public void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

These methods have been implemented as **final** methods in Object, so they are available in all the classes. All three methods can be called only from within a **synchronized** context.

Example

This examples shows how two threads can communicate using **wait()** and **notify()** method. You can create a complex system using the same concept.

Live Demo

```
class Chat {
   boolean flag = false;

   public synchronized void Question(String msg) {

      if (flag) {

         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = true;
      notify();
   }

   public synchronized void Answer(String msg) {

      if (!flag) {

         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = false;
      notify();
```

```
   }
}

class T1 implements Runnable {
   Chat m;
   String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };

   public T1(Chat m1) {
      this.m = m1;
      new Thread(this, "Question").start();
   }

   public void run() {

      for (int i = 0; i < s1.length; i++) {
         m.Question(s1[i]);
      }
   }
}

class T2 implements Runnable {
   Chat m;
   String[] s2 = { "Hi", "I am good, what about you?", "Great!" };

   public T2(Chat m2) {
      this.m = m2;
      new Thread(this, "Answer").start();
   }

   public void run() {

      for (int i = 0; i < s2.length; i++) {
         m.Answer(s2[i]);
      }
   }
}

public class TestThread {

   public static void main(String[] args) {
      Chat m = new Chat();
      new T1(m);
      new T2(m);
   }
}
```

When the above program is complied and executed, it produces the following result –

Output

```
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```

Above example has been taken and then modified from [https://stackoverflow.com/questions/2170520/inter-thread-communication-in-java]

Java Concurrency - Synchronization

Multithreading Example with Synchronization

Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

Example

Live Demo

```
class PrintDemo {

   public void printCount() {

      try {

         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
      } catch (Exception e) {
         System.out.println("Thread  interrupted.");
      }
   }
}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo(String name,  PrintDemo pd) {
      threadName = name;
      PD = pd;
   }

   public void run() {

      synchronized(PD) {
         PD.printCount();
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );

      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}

public class TestThread {
```

```
   public static void main(String args[]) {
      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo("Thread - 1 ", PD);
      ThreadDemo T2 = new ThreadDemo("Thread - 2 ", PD);

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch (Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

This produces the same result every time you run this program –

Output

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Java Concurrency - Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

Example

Live Demo

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();
```

```
   public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }

   private static class ThreadDemo1 extends Thread {

      public void run() {

         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");

            try {
               Thread.sleep(10);
            } catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");

            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }

   private static class ThreadDemo2 extends Thread {

      public void run() {

         synchronized (Lock2) {
            System.out.println("Thread 2: Holding lock 2...");

            try {
               Thread.sleep(10);
            } catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");

            synchronized (Lock1) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program –

Output

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
```

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by

pressing CTRL+C.

Deadlock Solution Example

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

Example

Live Demo

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }

   private static class ThreadDemo1 extends Thread {

      public void run() {

         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");

            try {
               Thread.sleep(10);
            } catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");

            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }

   private static class ThreadDemo2 extends Thread {

      public void run() {

         synchronized (Lock1) {
            System.out.println("Thread 2: Holding lock 1...");

            try {
               Thread.sleep(10);
            } catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 2...");

            synchronized (Lock2) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

So just changing the order of the locks prevent the program in going into a deadlock situation and completes with the following result –

Output

```
Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...
```

The above example is to just make the concept clear, however, it is a complex concept and you should deep dive into it before you develop your applications to deal with deadlock situations.

Java Concurrency - ThreadLocal Class

The ThreadLocal class is used to create thread local variables which can only be read and written by the same thread. For example, if two threads are accessing code having reference to same threadLocal variable then each thread will not see any modification to threadLocal variable done by other thread.

ThreadLocal Methods

Following is the list of important methods available in the ThreadLocal class.

Sr.No.	Method & Description
1	**public T get()** Returns the value in the current thread's copy of this thread-local variable.
2	**protected T initialValue()** Returns the current thread's "initial value" for this thread-local variable.
3	**public void remove()** Removes the current thread's value for this thread-local variable.
4	**public void set(T value)** Sets the current thread's copy of this thread-local variable to the specified value.

Example

The following TestThread program demonstrates some of these methods of the ThreadLocal class. Here we've used two counter variable, one is normal variable and another one is ThreadLocal.

Live Demo

```
class RunnableDemo implements Runnable {
   int counter;
   ThreadLocal<Integer> threadLocalCounter = new ThreadLocal<Integer>();

   public void run() {
      counter++;

      if(threadLocalCounter.get() != null) {
         threadLocalCounter.set(threadLocalCounter.get().intValue() + 1);
      } else {
         threadLocalCounter.set(0);
      }
      System.out.println("Counter: " + counter);
      System.out.println("threadLocalCounter: " + threadLocalCounter.get());
   }
}

public class TestThread {

   public static void main(String args[]) {
      RunnableDemo commonInstance = new RunnableDemo();

      Thread t1 = new Thread(commonInstance);
      Thread t2 = new Thread(commonInstance);
      Thread t3 = new Thread(commonInstance);
      Thread t4 = new Thread(commonInstance);

      t1.start();
      t2.start();
      t3.start();
      t4.start();

      // wait for threads to end
      try {
         t1.join();
         t2.join();
         t3.join();
         t4.join();
      } catch (Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

This will produce the following result.

Output

```
Counter: 1
threadLocalCounter: 0
Counter: 2
```

```
threadLocalCounter: 0
Counter: 3
threadLocalCounter: 0
Counter: 4
threadLocalCounter: 0
```

You can see the value of counter is increased by each thread, but threadLocalCounter remains 0 for each thread.

ThreadLocalRandom Class

A java.util.concurrent.ThreadLocalRandom is a utility class introduced from jdk 1.7 onwards and is useful when multiple threads or ForkJoinTasks are required to generate random numbers. It improves performance and have less contention than Math.random() method.

ThreadLocalRandom Methods

Following is the list of important methods available in the ThreadLocalRandom class.

Sr.No.	Method & Description
1	**public static ThreadLocalRandom current()** Returns the current thread's ThreadLocalRandom.
2	**protected int next(int bits)** Generates the next pseudorandom number.
3	**public double nextDouble(double n)** Returns a pseudorandom, uniformly distributed double value between 0 (inclusive) and the specified value (exclusive).
4	**public double nextDouble(double least, double bound)** Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
5	**public int nextInt(int least, int bound)** Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
6	**public long nextLong(long n)**

	Returns a pseudorandom, uniformly distributed value between 0 (inclusive) and the specified value (exclusive).
7	**public long nextLong(long least, long bound)** Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
8	**public void setSeed(long seed)** Throws UnsupportedOperationException.

Example

The following TestThread program demonstrates some of these methods of the Lock interface. Here we've used lock() to acquire the lock and unlock() to release the lock.

Live Demo

```
import java.util.Random;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.ThreadLocalRandom;

public class TestThread {

   public static void main(final String[] arguments) {
      System.out.println("Random Integer: " + new Random().nextInt());
      System.out.println("Seeded Random Integer: " + new Random(15).nextInt());
      System.out.println(
         "Thread Local Random Integer: " + ThreadLocalRandom.current().nextInt());

      final ThreadLocalRandom random = ThreadLocalRandom.current();
      random.setSeed(15); //exception will come as seeding is not allowed in ThreadLocalRandom.
      System.out.println("Seeded Thread Local Random Integer: " + random.nextInt());
   }
}
```

This will produce the following result.

Output

```
Random Integer: 1566889198
Seeded Random Integer: -1159716814
Thread Local Random Integer: 358693993
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.concurrent.ThreadLocalRandom.setSeed(Unknown Source)
        at TestThread.main(TestThread.java:21)
```

Here we've used ThreadLocalRandom and Random classes to get random numbers.

Java Concurrency - Lock Interface

A java.util.concurrent.locks.Lock interface is used to as a thread synchronization mechanism similar to synchronized blocks. New Locking mechanism is more flexible and provides more options than a synchronized block. Main differences between a Lock and a synchronized block are following –

Guarantee of sequence – Synchronized block does not provide any guarantee of sequence in which waiting thread will be given access. Lock interface handles it.

No timeout – Synchronized block has no option of timeout if lock is not granted. Lock interface provides such option.

Single method – Synchronized block must be fully contained within a single method whereas a lock interface's methods lock() and unlock() can be called in different methods.

Lock Methods

Following is the list of important methods available in the Lock class.

Sr.No.	Method & Description
1	**public void lock()** Acquires the lock.
2	**public void lockInterruptibly()** Acquires the lock unless the current thread is interrupted.
3	**public Condition newCondition()** Returns a new Condition instance that is bound to this Lock instance.
4	**public boolean tryLock()** Acquires the lock only if it is free at the time of invocation.
5	**public boolean tryLock()** Acquires the lock only if it is free at the time of invocation.
6	**public boolean tryLock(long time, TimeUnit unit)** Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

7	**public void unlock()** Releases the lock.

Example

The following TestThread program demonstrates some of these methods of the Lock interface. Here we've used lock() to acquire the lock and unlock() to release the lock.

Live Demo

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class PrintDemo {
   private final Lock queueLock = new ReentrantLock();

   public void print() {
      queueLock.lock();

      try {
         Long duration = (long) (Math.random() * 10000);
         System.out.println(Thread.currentThread().getName()
            + "  Time Taken " + (duration / 1000) + " seconds.");
         Thread.sleep(duration);
      } catch (InterruptedException e) {
         e.printStackTrace();
      } finally {
         System.out.printf(
            "%s printed the document successfully.\n", Thread.currentThread().getName());
         queueLock.unlock();
      }
   }
}

class ThreadDemo extends Thread {
   PrintDemo  printDemo;

   ThreadDemo(String name,  PrintDemo printDemo) {
      super(name);
      this.printDemo = printDemo;
   }

   @Override
   public void run() {
      System.out.printf(
         "%s starts printing a document\n", Thread.currentThread().getName());
      printDemo.print();
   }
}

public class TestThread {

   public static void main(String args[]) {
      PrintDemo PD = new PrintDemo();

      ThreadDemo t1 = new ThreadDemo("Thread - 1 ", PD);
      ThreadDemo t2 = new ThreadDemo("Thread - 2 ", PD);
      ThreadDemo t3 = new ThreadDemo("Thread - 3 ", PD);
      ThreadDemo t4 = new ThreadDemo("Thread - 4 ", PD);
```

```
      t1.start();
      t2.start();
      t3.start();
      t4.start();
   }
}
```

This will produce the following result.

Output

```
Thread - 1  starts printing a document
Thread - 4  starts printing a document
Thread - 3  starts printing a document
Thread - 2  starts printing a document
Thread - 1   Time Taken 4 seconds.
Thread - 1  printed the document successfully.
Thread - 4   Time Taken 3 seconds.
Thread - 4  printed the document successfully.
Thread - 3   Time Taken 5 seconds.
Thread - 3  printed the document successfully.
Thread - 2   Time Taken 4 seconds.
Thread - 2  printed the document successfully.
```

We've use ReentrantLock class as an implementation of Lock interface here. ReentrantLock class allows a thread to lock a method even if it already have the lock on other method.

Java Concurrency - ReadWriteLock Interface

A java.util.concurrent.locks.ReadWriteLock interface allows multiple threads to read at a time but only one thread can write at a time.

Read Lock – If no thread has locked the ReadWriteLock for writing then multiple thread can access the read lock.

Write Lock – If no thread is reading or writing, then one thread can access the write lock.

Lock Methods

Following is the list of important methods available in the Lock class.

Sr.No.	Method & Description
1	**public Lock readLock()** Returns the lock used for reading.

2	**public Lock writeLock()** Returns the lock used for writing.

Example

The following TestThread program demonstrates these methods of the ReadWriteLock interface. Here we've used readlock() to acquire the read-lock and writeLock() to acquire the write-lock.

Live Demo

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class TestThread {
   private static final ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);
   private static String message = "a";

   public static void main(String[] args) throws InterruptedException {
      Thread t1 = new Thread(new WriterA());
      t1.setName("Writer A");

      Thread t2 = new Thread(new WriterB());
      t2.setName("Writer B");

      Thread t3 = new Thread(new Reader());
      t3.setName("Reader");
      t1.start();
      t2.start();
      t3.start();
      t1.join();
      t2.join();
      t3.join();
   }

   static class Reader implements Runnable {

      public void run() {

         if(lock.isWriteLocked()) {
            System.out.println("Write Lock Present.");
         }
         lock.readLock().lock();

         try {
            Long duration = (long) (Math.random() * 10000);
            System.out.println(Thread.currentThread().getName()
               + "  Time Taken " + (duration / 1000) + " seconds.");
            Thread.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         } finally {
            System.out.println(Thread.currentThread().getName() +": "+ message );
            lock.readLock().unlock();
         }
      }
   }

   static class WriterA implements Runnable {

      public void run() {
```

```
         lock.writeLock().lock();

         try {
            Long duration = (long) (Math.random() * 10000);
            System.out.println(Thread.currentThread().getName()
               + "  Time Taken " + (duration / 1000) + " seconds.");
            Thread.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         } finally {
            message = message.concat("a");
            lock.writeLock().unlock();
         }
      }
   }

   static class WriterB implements Runnable {

      public void run() {
         lock.writeLock().lock();

         try {
            Long duration = (long) (Math.random() * 10000);
            System.out.println(Thread.currentThread().getName()
               + "  Time Taken " + (duration / 1000) + " seconds.");
            Thread.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         } finally {
            message = message.concat("b");
            lock.writeLock().unlock();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Writer A  Time Taken 6 seconds.
Write Lock Present.
Writer B  Time Taken 2 seconds.
Reader  Time Taken 0 seconds.
Reader: aab
```

Java Concurrency - Condition Interface

A java.util.concurrent.locks.Condition interface provides a thread ability to suspend its execution, until the given condition is true. A Condition object is necessarily bound to a Lock and to be obtained using the newCondition() method.

Condition Methods

Following is the list of important methods available in the Condition class.

Sr.No.	Method & Description
1	**public void await()** Causes the current thread to wait until it is signalled or interrupted.
2	**public boolean await(long time, TimeUnit unit)** Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
3	**public long awaitNanos(long nanosTimeout)** Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
4	**public long awaitUninterruptibly()** Causes the current thread to wait until it is signalled.
5	**public long awaitUntil()** Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
6	**public void signal()** Wakes up one waiting thread.
7	**public void signalAll()** Wakes up all waiting threads.

Example

The following TestThread program demonstrates these methods of the Condition interface. Here we've used signal() to notify and await() to suspend the thread.

Live Demo

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class TestThread {

   public static void main(String[] args) throws InterruptedException {
      ItemQueue itemQueue = new ItemQueue(10);

      //Create a producer and a consumer.
```

```
        Thread producer = new Producer(itemQueue);
        Thread consumer = new Consumer(itemQueue);

        //Start both threads.
        producer.start();
        consumer.start();

        //Wait for both threads to terminate.
        producer.join();
        consumer.join();
    }

    static class ItemQueue {
        private Object[] items = null;
        private int current = 0;
        private int placeIndex = 0;
        private int removeIndex = 0;

        private final Lock lock;
        private final Condition isEmpty;
        private final Condition isFull;

        public ItemQueue(int capacity) {
            this.items = new Object[capacity];
            lock = new ReentrantLock();
            isEmpty = lock.newCondition();
            isFull = lock.newCondition();
        }

        public void add(Object item) throws InterruptedException {
            lock.lock();

            while(current >= items.length)
                isFull.await();

            items[placeIndex] = item;
            placeIndex = (placeIndex + 1) % items.length;
            ++current;

            //Notify the consumer that there is data available.
            isEmpty.signal();
            lock.unlock();
        }

        public Object remove() throws InterruptedException {
            Object item = null;

            lock.lock();

            while(current <= 0) {
                isEmpty.await();
            }
            item = items[removeIndex];
            removeIndex = (removeIndex + 1) % items.length;
            --current;

            //Notify the producer that there is space available.
            isFull.signal();
            lock.unlock();

            return item;
```

```
      }

      public boolean isEmpty() {
         return (items.length == 0);
      }
   }

   static class Producer extends Thread {
      private final ItemQueue queue;

      public Producer(ItemQueue queue) {
         this.queue = queue;
      }

      @Override
      public void run() {
         String[] numbers =
            {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12"};

         try {

            for(String number: numbers) {
               System.out.println("[Producer]: " + number);
            }
            queue.add(null);
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      }
   }

   static class Consumer extends Thread {
      private final ItemQueue queue;

      public Consumer(ItemQueue queue) {
         this.queue = queue;
      }

      @Override
      public void run() {

         try {

            do {
               Object number = queue.remove();
               System.out.println("[Consumer]: " + number);

               if(number == null) {
                  return;
               }
            } while(!queue.isEmpty());
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
[Producer]: 1
[Producer]: 2
[Producer]: 3
[Producer]: 4
[Producer]: 5
[Producer]: 6
[Producer]: 7
[Producer]: 8
[Producer]: 9
[Producer]: 10
[Producer]: 11
[Producer]: 12
[Consumer]: null
```

Java Concurrency - AtomicInteger Class

A java.util.concurrent.atomic.AtomicInteger class provides operations on underlying int value that can be read and written atomically, and also contains advanced atomic operations. AtomicInteger supports atomic operations on underlying int variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicInteger Methods

Following is the list of important methods available in the AtomicInteger class.

Sr.No.	Method & Description
1	**public int addAndGet(int delta)** Atomically adds the given value to the current value.
2	**public boolean compareAndSet(int expect, int update)** Atomically sets the value to the given updated value if the current value is same as the expected value.
3	**public int decrementAndGet()** Atomically decrements by one the current value.
4	**public double doubleValue()**

	Returns the value of the specified number as a double.
5	**public float floatValue()** Returns the value of the specified number as a float.
6	**public int get()** Gets the current value.
7	**public int getAndAdd(int delta)** Atomiclly adds the given value to the current value.
8	**public int getAndDecrement()** Atomically decrements by one the current value.
9	**public int getAndIncrement()** Atomically increments by one the current value.
10	**public int getAndSet(int newValue)** Atomically sets to the given value and returns the old value.
11	**public int incrementAndGet()** Atomically increments by one the current value.
12	**public int intValue()** Returns the value of the specified number as an int.
13	**public void lazySet(int newValue)** Eventually sets to the given value.
14	**public long longValue()** Returns the value of the specified number as a long.
15	**public void set(int newValue)** Sets to the given value.

16	**public String toString()** Returns the String representation of the current value.
17	**public boolean weakCompareAndSet(int expect, int update)** Atomically sets the value to the given updated value if the current value is same as the expected value.

Example

The following TestThread program shows a unsafe implementation of counter in thread based environment.

Live Demo

```
public class TestThread {

   static class Counter {
      private int c = 0;

      public void increment() {
         c++;
      }

      public int value() {
         return c;
      }
   }

   public static void main(final String[] arguments) throws InterruptedException {
      final Counter counter = new Counter();

      //1000 threads
      for(int i = 0; i < 1000 ; i++) {

         new Thread(new Runnable() {

            public void run() {
               counter.increment();
            }
         }).start();
      }
      Thread.sleep(6000);
      System.out.println("Final number (should be 1000): " + counter.value());
   }
}
```

This may produce the following result depending upon computer's speed and thread interleaving.

Output

```
Final number (should be 1000): 1000
```

Example

The following TestThread program shows a safe implementation of counter using AtomicInteger in thread based environment.

Live Demo

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

public class TestThread {

   static class Counter {
      private AtomicInteger c = new AtomicInteger(0);

      public void increment() {
         c.getAndIncrement();
      }

      public int value() {
         return c.get();
      }
   }

   public static void main(final String[] arguments) throws InterruptedException {
      final Counter counter = new Counter();

      //1000 threads
      for(int i = 0; i < 1000 ; i++) {

         new Thread(new Runnable() {
            public void run() {
               counter.increment();
            }
         }).start();
      }
      Thread.sleep(6000);
      System.out.println("Final number (should be 1000): " + counter.value());
   }
}
```

This will produce the following result.

Output

```
Final number (should be 1000): 1000
```

Java Concurrency - AtomicLong Class

A java.util.concurrent.atomic.AtomicLong class provides operations on underlying long value that can be read and written atomically, and also contains advanced atomic operations. AtomicLong supports atomic operations on underlying long variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicLong Methods

Following is the list of important methods available in the AtomicLong class.

Sr.No.	Method & Description
1	**public long addAndGet(long delta)** Atomically adds the given value to the current value.
2	**public boolean compareAndSet(long expect, long update)** Atomically sets the value to the given updated value if the current value is same as the expected value.
3	**public long decrementAndGet()** Atomically decrements by one the current value.
4	**public double doubleValue()** Returns the value of the specified number as a double.
5	**public float floatValue()** Returns the value of the specified number as a float.
6	**public long get()** Gets the current value.
7	**public long getAndAdd(long delta)** Atomiclly adds the given value to the current value.
8	**public long getAndDecrement()** Atomically decrements by one the current value.
9	**public long getAndIncrement()** Atomically increments by one the current value.
10	**public long getAndSet(long newValue)** Atomically sets to the given value and returns the old value.
11	**public long incrementAndGet()** Atomically increments by one the current value.

12	**public int intValue()** Returns the value of the specified number as an int.
13	**public void lazySet(long newValue)** Eventually sets to the given value.
14	**public long longValue()** Returns the value of the specified number as a long.
15	**public void set(long newValue)** Sets to the given value.
16	**public String toString()** Returns the String representation of the current value.
17	**public boolean weakCompareAndSet(long expect, long update)** Atomically sets the value to the given updated value if the current value is same as the expected value.

Example

The following TestThread program shows a safe implementation of counter using AtomicLong in thread based environment.

Live Demo

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

public class TestThread {

   static class Counter {
      private AtomicLong c = new AtomicLong(0);

      public void increment() {
         c.getAndIncrement();
      }

      public long value() {
         return c.get();
      }
   }

   public static void main(final String[] arguments) throws InterruptedException {
      final Counter counter = new Counter();

      //1000 threads
      for(int i = 0; i < 1000 ; i++) {
```

```
         new Thread(new Runnable() {

            public void run() {
               counter.increment();
            }
         }).start();
      }
      Thread.sleep(6000);
      System.out.println("Final number (should be 1000): " + counter.value());
   }
}
```

This will produce the following result.

Output

```
Final number (should be 1000): 1000
```

Java Concurrency - AtomicBoolean Class

A java.util.concurrent.atomic.AtomicBoolean class provides operations on underlying boolean value that can be read and written atomically, and also contains advanced atomic operations. AtomicBoolean supports atomic operations on underlying boolean variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicBoolean Methods

Following is the list of important methods available in the AtomicBoolean class.

Sr.No.	Method & Description
1	**public boolean compareAndSet(boolean expect, boolean update)** Atomically sets the value to the given updated value if the current value == the expected value.
2	**public boolean get()** Returns the current value.
3	**public boolean getAndSet(boolean newValue)** Atomically sets to the given value and returns the previous value.
4	**public void lazySet(boolean newValue)** Eventually sets to the given value.

5	**public void set(boolean newValue)** Unconditionally sets to the given value.
6	**public String toString()** Returns the String representation of the current value.
7	**public boolean weakCompareAndSet(boolean expect, boolean update)** Atomically sets the value to the given updated value if the current value == the expected value.

Example

The following TestThread program shows usage of AtomicBoolean variable in thread based environment.

Live Demo

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

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      final AtomicBoolean atomicBoolean = new AtomicBoolean(false);

      new Thread("Thread 1") {

         public void run() {

            while(true) {
               System.out.println(Thread.currentThread().getName()
                  +" Waiting for Thread 2 to set Atomic variable to true. Current value is "
                  + atomicBoolean.get());

               if(atomicBoolean.compareAndSet(true, false)) {
                  System.out.println("Done!");
                  break;
               }
            }
         };
      }.start();

      new Thread("Thread 2") {

         public void run() {
            System.out.println(Thread.currentThread().getName() +
               ", Atomic Variable: " +atomicBoolean.get());
            System.out.println(Thread.currentThread().getName() +
               " is setting the variable to true ");
            atomicBoolean.set(true);
            System.out.println(Thread.currentThread().getName() +
               ", Atomic Variable: " +atomicBoolean.get());
         };
      }.start();
```

```
    }
}
```

This will produce the following result.

Output

```
Thread 1 Waiting for Thread 2 to set Atomic variable to true. Current value is false
Thread 1 Waiting for Thread 2 to set Atomic variable to true. Current value is false
Thread 1 Waiting for Thread 2 to set Atomic variable to true. Current value is false
Thread 2, Atomic Variable: false
Thread 1 Waiting for Thread 2 to set Atomic variable to true. Current value is false
Thread 2 is setting the variable to true
Thread 2, Atomic Variable: true
Thread 1 Waiting for Thread 2 to set Atomic variable to true. Current value is false
Done!
```

Java Concurrency - AtomicReference Class

A java.util.concurrent.atomic.AtomicReference class provides operations on underlying object reference that can be read and written atomically, and also contains advanced atomic operations. AtomicReference supports atomic operations on underlying object reference variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicReference Methods

Following is the list of important methods available in the AtomicReference class.

Sr.No.	Method & Description
1	**public boolean compareAndSet(V expect, V update)** Atomically sets the value to the given updated value if the current value == the expected value.
2	**public boolean get()** Returns the current value.
3	**public boolean getAndSet(V newValue)** Atomically sets to the given value and returns the previous value.
4	**public void lazySet(V newValue)**

	Eventually sets to the given value.
5	**public void set(V newValue)** Unconditionally sets to the given value.
6	**public String toString()** Returns the String representation of the current value.
7	**public boolean weakCompareAndSet(V expect, V update)** Atomically sets the value to the given updated value if the current value == the expected value.

Example

The following TestThread program shows usage of AtomicReference variable in thread based environment.

Live Demo

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

public class TestThread {
   private static String message = "hello";
   private static AtomicReference<String> atomicReference;

   public static void main(final String[] arguments) throws InterruptedException {
      atomicReference = new AtomicReference<String>(message);

      new Thread("Thread 1") {

         public void run() {
            atomicReference.compareAndSet(message, "Thread 1");
            message = message.concat("-Thread 1!");
         };
      }.start();

      System.out.println("Message is: " + message);
      System.out.println("Atomic Reference of Message is: " + atomicReference.get());
   }
}
```

This will produce the following result.

Output

```
Message is: hello
Atomic Reference of Message is: Thread 1
```

Java Concurrency - AtomicIntegerArray Class

A java.util.concurrent.atomic.AtomicIntegerArray class provides operations on underlying int array that can be read and written atomically, and also contains advanced atomic operations. AtomicIntegerArray supports atomic operations on underlying int array variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicIntegerArray Methods

Following is the list of important methods available in the AtomicIntegerArray class.

Sr.No.	Method & Description
1	**public int addAndGet(int i, int delta)** Atomically adds the given value to the element at index i.
2	**public boolean compareAndSet(int i, int expect, int update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.
3	**public int decrementAndGet(int i)** Atomically decrements by one the element at index i.
4	**public int get(int i)** Gets the current value at position i.
5	**public int getAndAdd(int i, int delta)** Atomically adds the given value to the element at index i.
6	**public int getAndDecrement(int i)** Atomically decrements by one the element at index i.
7	**public int getAndIncrement(int i)** Atomically increments by one the element at index i.
8	**public int getAndSet(int i, int newValue)** Atomically sets the element at position i to the given value and returns the old value.

9	**public int incrementAndGet(int i)** Atomically increments by one the element at index i.
10	**public void lazySet(int i, int newValue)** Eventually sets the element at position i to the given value.
11	**public int length()** Returns the length of the array.
12	**public void set(int i, int newValue)** Sets the element at position i to the given value.
13	**public String toString()** Returns the String representation of the current values of array.
14	**public boolean weakCompareAndSet(int i, int expect, int update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.

Example

The following TestThread program shows usage of AtomicIntegerArray variable in thread based environment.

Live Demo

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

public class TestThread {
   private static AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(10);

   public static void main(final String[] arguments) throws InterruptedException {

      for (int i = 0; i<atomicIntegerArray.length(); i++) {
         atomicIntegerArray.set(i, 1);
      }

      Thread t1 = new Thread(new Increment());
      Thread t2 = new Thread(new Compare());
      t1.start();
      t2.start();

      t1.join();
      t2.join();

      System.out.println("Values: ");
```

```
      for (int i = 0; i<atomicIntegerArray.length(); i++) {
         System.out.print(atomicIntegerArray.get(i) + " ");
      }
   }

   static class Increment implements Runnable {

      public void run() {

         for(int i = 0; i<atomicIntegerArray.length(); i++) {
            int add = atomicIntegerArray.incrementAndGet(i);
            System.out.println("Thread " + Thread.currentThread().getId()
               + ", index " +i + ", value: "+ add);
         }
      }
   }

   static class Compare implements Runnable {

      public void run() {

         for(int i = 0; i<atomicIntegerArray.length(); i++) {
            boolean swapped = atomicIntegerArray.compareAndSet(i, 2, 3);

            if(swapped) {
               System.out.println("Thread " + Thread.currentThread().getId()
                  + ", index " +i + ", value: 3");
            }
         }
      }
   }
}
```

This will produce the following result.

Output

```
Thread 10, index 0, value: 2
Thread 10, index 1, value: 2
Thread 10, index 2, value: 2
Thread 11, index 0, value: 3
Thread 10, index 3, value: 2
Thread 11, index 1, value: 3
Thread 11, index 2, value: 3
Thread 10, index 4, value: 2
Thread 11, index 3, value: 3
Thread 10, index 5, value: 2
Thread 10, index 6, value: 2
Thread 11, index 4, value: 3
Thread 10, index 7, value: 2
Thread 11, index 5, value: 3
Thread 10, index 8, value: 2
Thread 11, index 6, value: 3
```

```
Thread 10, index 9, value: 2
Thread 11, index 7, value: 3
Thread 11, index 8, value: 3
Thread 11, index 9, value: 3
Values:
3 3 3 3 3 3 3 3 3 3
```

Java Concurrency - AtomicLongArray Class

A java.util.concurrent.atomic.AtomicLongArray class provides operations on underlying long array that can be read and written atomically, and also contains advanced atomic operations. AtomicLongArray supports atomic operations on underlying long array variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicLongArray Methods

Following is the list of important methods available in the AtomicLongArray class.

Sr.No.	Method & Description
1	**public long addAndGet(int i, long delta)** Atomically adds the given value to the element at index i.
2	**public boolean compareAndSet(int i, long expect, long update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.
3	**public long decrementAndGet(int i)** Atomically decrements by one the element at index i.
4	**public long get(int i)** Gets the current value at position i.
5	**public long getAndAdd(int i, long delta)** Atomically adds the given value to the element at index i.
6	**public long getAndDecrement(int i)** Atomically decrements by one the element at index i.

7	**public long getAndIncrement(int i)** Atomically increments by one the element at index i.
8	**public long getAndSet(int i, long newValue)** Atomically sets the element at position i to the given value and returns the old value.
9	**public long incrementAndGet(int i)** Atomically increments by one the element at index i.
10	**public void lazySet(int i, long newValue)** Eventually sets the element at position i to the given value.
11	**public int length()** Returns the length of the array.
12	**public void set(int i, long newValue)** Sets the element at position i to the given value.
13	**public String toString()** Returns the String representation of the current values of array.
14	**public boolean weakCompareAndSet(int i, long expect, long update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.

Example

The following TestThread program shows usage of AtomicIntegerArray variable in thread based environment.

Live Demo

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

public class TestThread {
   private static AtomicLongArray atomicLongArray = new AtomicLongArray(10);

   public static void main(final String[] arguments) throws InterruptedException {

      for (int i = 0; i<atomicLongArray.length(); i++) {
```

```
         atomicLongArray.set(i, 1);
      }

      Thread t1 = new Thread(new Increment());
      Thread t2 = new Thread(new Compare());
      t1.start();
      t2.start();

      t1.join();
      t2.join();

      System.out.println("Values: ");

      for (int i = 0; i<atomicLongArray.length(); i++) {
         System.out.print(atomicLongArray.get(i) + " ");
      }
   }

   static class Increment implements Runnable {

      public void run() {

         for(int i = 0; i<atomicLongArray.length(); i++) {
            long add = atomicLongArray.incrementAndGet(i);
            System.out.println("Thread " + Thread.currentThread().getId()
               + ", index " +i + ", value: "+ add);
         }
      }
   }

   static class Compare implements Runnable {

      public void run() {

         for(int i = 0; i<atomicLongArray.length(); i++) {
            boolean swapped = atomicLongArray.compareAndSet(i, 2, 3);

            if(swapped) {
               System.out.println("Thread " + Thread.currentThread().getId()
                  + ", index " +i + ", value: 3");
            }
         }
      }
   }
}
```

This will produce the following result.

Output

```
Thread 9, index 0, value: 2
Thread 10, index 0, value: 3
Thread 9, index 1, value: 2
Thread 9, index 2, value: 2
Thread 9, index 3, value: 2
Thread 9, index 4, value: 2
Thread 10, index 1, value: 3
```

```
Thread 9, index 5, value: 2
Thread 10, index 2, value: 3
Thread 9, index 6, value: 2
Thread 10, index 3, value: 3
Thread 9, index 7, value: 2
Thread 10, index 4, value: 3
Thread 9, index 8, value: 2
Thread 9, index 9, value: 2
Thread 10, index 5, value: 3
Thread 10, index 6, value: 3
Thread 10, index 7, value: 3
Thread 10, index 8, value: 3
Thread 10, index 9, value: 3
Values:
3 3 3 3 3 3 3 3 3 3
```

AtomicReferenceArray Class

A java.util.concurrent.atomic.AtomicReferenceArray class provides operations on underlying reference array that can be read and written atomically, and also contains advanced atomic operations. AtomicReferenceArray supports atomic operations on underlying reference array variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features.

AtomicReferenceArray Methods

Following is the list of important methods available in the AtomicReferenceArray class.

Sr.No.	Method & Description
1	**public boolean compareAndSet(int i, E expect, E update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.
2	**public E get(int i)** Gets the current value at position i.
3	**public E getAndSet(int i, E newValue)** Atomically sets the element at position i to the given value and returns the old value.

4	**public void lazySet(int i, E newValue)** Eventually sets the element at position i to the given value.
5	**public int length()** Returns the length of the array.
6	**public void set(int i, E newValue)** Sets the element at position i to the given value.
7	**public String toString()** Returns the String representation of the current values of array.
8	**public boolean weakCompareAndSet(int i, E expect, E update)** Atomically sets the element at position i to the given updated value if the current value == the expected value.

Example

The following TestThread program shows usage of AtomicReferenceArray variable in thread based environment.

Live Demo

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

public class TestThread {
   private static String[] source = new String[10];
   private static AtomicReferenceArray<String> atomicReferenceArray
      = new AtomicReferenceArray<String>(source);

   public static void main(final String[] arguments) throws InterruptedException {

      for (int i = 0; i<atomicReferenceArray.length(); i++) {
         atomicReferenceArray.set(i, "item-2");
      }

      Thread t1 = new Thread(new Increment());
      Thread t2 = new Thread(new Compare());
      t1.start();
      t2.start();

      t1.join();
      t2.join();
   }

   static class Increment implements Runnable {

      public void run() {
```

```
         for(int i = 0; i<atomicReferenceArray.length(); i++) {
            String add = atomicReferenceArray.getAndSet(i,"item-"+ (i+1));
            System.out.println("Thread " + Thread.currentThread().getId()
               + ", index " +i + ", value: "+ add);
         }
      }
   }

   static class Compare implements Runnable {

      public void run() {

         for(int i = 0; i<atomicReferenceArray.length(); i++) {
            System.out.println("Thread " + Thread.currentThread().getId()
               + ", index " +i + ", value: "+ atomicReferenceArray.get(i));
            boolean swapped = atomicReferenceArray.compareAndSet(i, "item-2", "updated-item-2");
            System.out.println("Item swapped: " + swapped);

            if(swapped) {
               System.out.println("Thread " + Thread.currentThread().getId()
                  + ", index " +i + ", updated-item-2");
            }
         }
      }
   }
}
```

This will produce the following result.

Output

```
Thread 9, index 0, value: item-2
Thread 10, index 0, value: item-1
Item swapped: false
Thread 10, index 1, value: item-2
Item swapped: true
Thread 9, index 1, value: updated-item-2
Thread 10, index 1, updated-item-2
Thread 10, index 2, value: item-3
Item swapped: false
Thread 10, index 3, value: item-2
Item swapped: true
Thread 10, index 3, updated-item-2
Thread 10, index 4, value: item-2
Item swapped: true
Thread 10, index 4, updated-item-2
Thread 10, index 5, value: item-2
Item swapped: true
Thread 10, index 5, updated-item-2
Thread 10, index 6, value: item-2
Thread 9, index 2, value: item-2
Item swapped: true
```

```
Thread 9, index 3, value: updated-item-2
Thread 10, index 6, updated-item-2
Thread 10, index 7, value: item-2
Thread 9, index 4, value: updated-item-2
Item swapped: true
Thread 9, index 5, value: updated-item-2
Thread 10, index 7, updated-item-2
Thread 9, index 6, value: updated-item-2
Thread 10, index 8, value: item-2
Thread 9, index 7, value: updated-item-2
Item swapped: true
Thread 9, index 8, value: updated-item-2
Thread 10, index 8, updated-item-2
Thread 9, index 9, value: item-2
Thread 10, index 9, value: item-10
Item swapped: false
```

Java Concurrency - Executor Interface

A java.util.concurrent.Executor interface is a simple interface to support launching new tasks.

ExecutorService Methods

Sr.No.	Method & Description
1	**void execute(Runnable command)** Executes the given command at some time in the future.

Example

The following TestThread program shows usage of Executor interface in thread based environment.

Live Demo

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      Executor executor = Executors.newCachedThreadPool();
      executor.execute(new Task());
      ThreadPoolExecutor pool = (ThreadPoolExecutor)executor;
      pool.shutdown();
   }
```

```
   static class Task implements Runnable {

      public void run() {

         try {
            Long duration = (long) (Math.random() * 5);
            System.out.println("Running Task!");
            TimeUnit.SECONDS.sleep(duration);
            System.out.println("Task Completed");
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Running Task!
Task Completed
```

ExecutorService Interface

A java.util.concurrent.ExecutorService interface is a subinterface of Executor interface, and adds features to manage the lifecycle, both of the individual tasks and of the executor itself.

ExecutorService Methods

Sr.No.	Method & Description
1	**boolean awaitTermination(long timeout, TimeUnit unit)** Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
2	**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)** Executes the given tasks, returning a list of Futures holding their status and results when all complete.
3	**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)** Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.

4	**<T> T invokeAny(Collection<? extends Callable<T>> tasks)** Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.
5	**<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)** Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
6	**boolean isShutdown()** Returns true if this executor has been shut down.
7	**boolean isTerminated()** Returns true if all tasks have completed following shut down.
8	**void shutdown()** Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
9	**List<Runnable> shutdownNow()** Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
10	**<T> Future<T> submit(Callable<T> task)** Submits a value-returning task for execution and returns a Future representing the pending results of the task.
11	**Future<?> submit(Runnable task)** Submits a Runnable task for execution and returns a Future representing that task.
12	**<T> Future<T> submit(Runnable task, T result)** Submits a Runnable task for execution and returns a Future representing that task.

Example

The following TestThread program shows usage of ExecutorService interface in thread based environment.

Live Demo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();

      try {
         executor.submit(new Task());
         System.out.println("Shutdown executor");
         executor.shutdown();
         executor.awaitTermination(5, TimeUnit.SECONDS);
      } catch (InterruptedException e) {
         System.err.println("tasks interrupted");
      } finally {

         if (!executor.isTerminated()) {
            System.err.println("cancel non-finished tasks");
         }
         executor.shutdownNow();
         System.out.println("shutdown finished");
      }
   }

   static class Task implements Runnable {

      public void run() {

         try {
            Long duration = (long) (Math.random() * 20);
            System.out.println("Running Task!");
            TimeUnit.SECONDS.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Shutdown executor
Running Task!
shutdown finished
cancel non-finished tasks
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:302)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:328)
```

```
        at TestThread$Task.run(TestThread.java:39)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:439)
        at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
        at java.util.concurrent.FutureTask.run(FutureTask.java:138)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
        at java.lang.Thread.run(Thread.java:662)
```

ScheduledExecutorService Interface

A java.util.concurrent.ScheduledExecutorService interface is a subinterface of ExecutorService interface, and supports future and/or periodic execution of tasks.

ScheduledExecutorService Methods

Sr.No.	Method & Description
1	**<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)** Creates and executes a ScheduledFuture that becomes enabled after the given delay.
2	**ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)** Creates and executes a one-shot action that becomes enabled after the given delay.
3	**ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 * period, and so on.
4	**ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)** Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.

Example

The following TestThread program shows usage of ScheduledExecutorService interface in thread based environment.

Live Demo

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

      final ScheduledFuture<?> beepHandler =
         scheduler.scheduleAtFixedRate(new BeepTask(), 2, 2, TimeUnit.SECONDS);

      scheduler.schedule(new Runnable() {

         @Override
         public void run() {
            beepHandler.cancel(true);
            scheduler.shutdown();
         }
      }, 10, TimeUnit.SECONDS);
   }

   static class BeepTask implements Runnable {

      public void run() {
         System.out.println("beep");
      }
   }
}
```

This will produce the following result.

Output

```
beep
beep
beep
beep
```

newFixedThreadPool Method

A fixed thread pool can be obtainted by calling the static newFixedThreadPool() method of Executors class.

Syntax

```
ExecutorService fixedPool = Executors.newFixedThreadPool(2);
```

where

Maximum 2 threads will be active to process tasks.

If more than 2 threads are submitted then they are held in a queue until threads become available.

A new thread is created to take its place if a thread terminates due to failure during execution shutdown on executor is not yet called.

Any thread exists till the pool is shutdown.

Example

The following TestThread program shows usage of newFixedThreadPool method in thread based environment.

Live Demo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      ExecutorService executor = Executors.newFixedThreadPool(2);

      // Cast the object to its class type
      ThreadPoolExecutor pool = (ThreadPoolExecutor) executor;

      //Stats before tasks execution
      System.out.println("Largest executions: "
         + pool.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + pool.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + pool.getPoolSize());
      System.out.println("Currently executing threads: "
         + pool.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + pool.getTaskCount());

      executor.submit(new Task());
      executor.submit(new Task());

      //Stats after tasks execution
      System.out.println("Core threads: " + pool.getCorePoolSize());
      System.out.println("Largest executions: "
         + pool.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + pool.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + pool.getPoolSize());
      System.out.println("Currently executing threads: "
         + pool.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + pool.getTaskCount());

      executor.shutdown();
```

```
   }

   static class Task implements Runnable {

      public void run() {

         try {
            Long duration = (long) (Math.random() * 5);
            System.out.println("Running Task! Thread Name: " +
               Thread.currentThread().getName());
               TimeUnit.SECONDS.sleep(duration);

            System.out.println("Task Completed! Thread Name: " +
               Thread.currentThread().getName());
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Largest executions: 0
Maximum allowed threads: 2
Current threads in pool: 0
Currently executing threads: 0
Total number of threads(ever scheduled): 0
Core threads: 2
Largest executions: 2
Maximum allowed threads: 2
Current threads in pool: 2
Currently executing threads: 1
Total number of threads(ever scheduled): 2
Running Task! Thread Name: pool-1-thread-1
Running Task! Thread Name: pool-1-thread-2
Task Completed! Thread Name: pool-1-thread-2
Task Completed! Thread Name: pool-1-thread-1
```

newCachedThreadPool Method

A cached thread pool can be obtainted by calling the static newCachedThreadPool() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newCachedThreadPool();
```

where

newCachedThreadPool method creates an executor having an expandable thread pool.

Such an executor is suitable for applications that launch many short-lived tasks.

Example

The following TestThread program shows usage of newCachedThreadPool method in thread based environment.

Live Demo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      ExecutorService executor = Executors.newCachedThreadPool();

      // Cast the object to its class type
      ThreadPoolExecutor pool = (ThreadPoolExecutor) executor;

      //Stats before tasks execution
      System.out.println("Largest executions: "
         + pool.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + pool.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + pool.getPoolSize());
      System.out.println("Currently executing threads: "
         + pool.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + pool.getTaskCount());

      executor.submit(new Task());
      executor.submit(new Task());

      //Stats after tasks execution
      System.out.println("Core threads: " + pool.getCorePoolSize());
      System.out.println("Largest executions: "
         + pool.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + pool.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + pool.getPoolSize());
      System.out.println("Currently executing threads: "
         + pool.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + pool.getTaskCount());

      executor.shutdown();
   }

   static class Task implements Runnable {
```

```
      public void run() {

         try {
            Long duration = (long) (Math.random() * 5);
            System.out.println("Running Task! Thread Name: " +
               Thread.currentThread().getName());
               TimeUnit.SECONDS.sleep(duration);
            System.out.println("Task Completed! Thread Name: " +
               Thread.currentThread().getName());
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Largest executions: 0
Maximum allowed threads: 2147483647
Current threads in pool: 0
Currently executing threads: 0
Total number of threads(ever scheduled): 0
Core threads: 0
Largest executions: 2
Maximum allowed threads: 2147483647
Current threads in pool: 2
Currently executing threads: 2
Total number of threads(ever scheduled): 2
Running Task! Thread Name: pool-1-thread-1
Running Task! Thread Name: pool-1-thread-2
Task Completed! Thread Name: pool-1-thread-2
Task Completed! Thread Name: pool-1-thread-1
```

newScheduledThreadPool Method

A scheduled thread pool can be obtainted by calling the static newScheduledThreadPool() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newScheduledThreadPool(1);
```

Example

The following TestThread program shows usage of newScheduledThreadPool method in thread based environment.

Live Demo

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

      final ScheduledFuture<?> beepHandler =
         scheduler.scheduleAtFixedRate(new BeepTask(), 2, 2, TimeUnit.SECONDS);

      scheduler.schedule(new Runnable() {

         @Override
         public void run() {
            beepHandler.cancel(true);
            scheduler.shutdown();
         }
      }, 10, TimeUnit.SECONDS);
   }

   static class BeepTask implements Runnable {

      public void run() {
         System.out.println("beep");
      }
   }
}
```

This will produce the following result.

Output

```
beep
beep
beep
beep
```

newSingleThreadExecutor Method

A single thread pool can be obtainted by calling the static newSingleThreadExecutor() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Where newSingleThreadExecutor method creates an executor that executes a single task at a time.

Example

The following TestThread program shows usage of newSingleThreadExecutor method in thread based environment.

Live Demo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();

      try {
         executor.submit(new Task());
         System.out.println("Shutdown executor");
         executor.shutdown();
         executor.awaitTermination(5, TimeUnit.SECONDS);
      } catch (InterruptedException e) {
         System.err.println("tasks interrupted");
      } finally {

         if (!executor.isTerminated()) {
            System.err.println("cancel non-finished tasks");
         }
         executor.shutdownNow();
         System.out.println("shutdown finished");
      }
   }

   static class Task implements Runnable {

      public void run() {

         try {
            Long duration = (long) (Math.random() * 20);
            System.out.println("Running Task!");
            TimeUnit.SECONDS.sleep(duration);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Shutdown executor
Running Task!
```

```
shutdown finished
cancel non-finished tasks
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:302)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:328)
        at TestThread$Task.run(TestThread.java:39)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:439)
        at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
        at java.util.concurrent.FutureTask.run(FutureTask.java:138)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
        at java.lang.Thread.run(Thread.java:662)
```

ThreadPoolExecutor Class

java.util.concurrent.ThreadPoolExecutor is an ExecutorService to execute each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods. It also provides various utility methods to check current threads statistics and control them.

ThreadPoolExecutor Methods

Sr.No.	Method & Description
1	**protected void afterExecute(Runnable r, Throwable t)** Method invoked upon completion of execution of the given Runnable.
2	**void allowCoreThreadTimeOut(boolean value)** Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.
3	**boolean allowsCoreThreadTimeOut()** Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive.
4	**boolean awaitTermination(long timeout, TimeUnit unit)** Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

5	**protected void beforeExecute(Thread t, Runnable r)** Method invoked prior to executing the given Runnable in the given thread.
6	**void execute(Runnable command)** Executes the given task sometime in the future.
7	**protected void finalize()** Invokes shutdown when this executor is no longer referenced and it has no threads.
8	**int getActiveCount()** Returns the approximate number of threads that are actively executing tasks.
9	**long getCompletedTaskCount()** Returns the approximate total number of tasks that have completed execution.
10	**int getCorePoolSize()** Returns the core number of threads.
11	**long getKeepAliveTime(TimeUnit unit)** Returns the thread keep-alive time, which is the amount of time that threads in excess of the core pool size may remain idle before being terminated.
12	**int getLargestPoolSize()** Returns the largest number of threads that have ever simultaneously been in the pool.
13	**int getMaximumPoolSize()** Returns the maximum allowed number of threads.
14	**int getPoolSize()** Returns the current number of threads in the pool.
15	**BlockingQueue getQueue()** Returns the task queue used by this executor.

15	**RejectedExecutionHandler getRejectedExecutionHandler()** Returns the current handler for unexecutable tasks.
16	**long getTaskCount()** Returns the approximate total number of tasks that have ever been scheduled for execution.
17	**ThreadFactory getThreadFactory()** Returns the thread factory used to create new threads.
18	**boolean isShutdown()** Returns true if this executor has been shut down.
19	**boolean isTerminated()** Returns true if all tasks have completed following shut down.
20	**boolean isTerminating()** Returns true if this executor is in the process of terminating after shutdown() or shutdownNow() but has not completely terminated.
21	**int prestartAllCoreThreads()** Starts all core threads, causing them to idly wait for work.
22	**boolean prestartCoreThread()** Starts a core thread, causing it to idly wait for work.
23	**void purge()** Tries to remove from the work queue all Future tasks that have been cancelled.
24	**boolean remove(Runnable task)** Removes this task from the executor's internal queue if it is present, thus causing it not to be run if it has not already started.
25	**void setCorePoolSize(int corePoolSize)**

	Sets the core number of threads.
26	**void setKeepAliveTime(long time, TimeUnit unit)** Sets the time limit for which threads may remain idle before being terminated.
27	**void setMaximumPoolSize(int maximumPoolSize)** Sets the maximum allowed number of threads.
28	**void setRejectedExecutionHandler(RejectedExecutionHandler handler)** Sets a new handler for unexecutable tasks.
29	**void setThreadFactory(ThreadFactory threadFactory)** Sets the thread factory used to create new threads.
30	**void shutdown()** Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
31	**List<Runnable> shutdownNow()** Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
32	**protected void terminated()** Method invoked when the Executor has terminated.
33	**String toString()** Returns a string identifying this pool, as well as its state, including indications of run state and estimated worker and task counts.

Example

The following TestThread program shows usage of ThreadPoolExecutor interface in thread based environment.

Live Demo

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class TestThread {
```

```
   public static void main(final String[] arguments) throws InterruptedException {
      ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.newCachedThreadPool();

      //Stats before tasks execution
      System.out.println("Largest executions: "
         + executor.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + executor.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + executor.getPoolSize());
      System.out.println("Currently executing threads: "
         + executor.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + executor.getTaskCount());

      executor.submit(new Task());
      executor.submit(new Task());

      //Stats after tasks execution
      System.out.println("Core threads: " + executor.getCorePoolSize());
      System.out.println("Largest executions: "
         + executor.getLargestPoolSize());
      System.out.println("Maximum allowed threads: "
         + executor.getMaximumPoolSize());
      System.out.println("Current threads in pool: "
         + executor.getPoolSize());
      System.out.println("Currently executing threads: "
         + executor.getActiveCount());
      System.out.println("Total number of threads(ever scheduled): "
         + executor.getTaskCount());

      executor.shutdown();
   }

   static class Task implements Runnable {

      public void run() {

         try {
            Long duration = (long) (Math.random() * 5);
            System.out.println("Running Task! Thread Name: " +
               Thread.currentThread().getName());
            TimeUnit.SECONDS.sleep(duration);
            System.out.println("Task Completed! Thread Name: " +
               Thread.currentThread().getName());
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Largest executions: 0
Maximum allowed threads: 2147483647
```

```
Current threads in pool: 0
Currently executing threads: 0
Total number of threads(ever scheduled): 0
Core threads: 0
Largest executions: 2
Maximum allowed threads: 2147483647
Current threads in pool: 2
Currently executing threads: 2
Total number of threads(ever scheduled): 2
Running Task! Thread Name: pool-1-thread-2
Running Task! Thread Name: pool-1-thread-1
Task Completed! Thread Name: pool-1-thread-1
Task Completed! Thread Name: pool-1-thread-2
```

ScheduledThreadPoolExecutor Class

java.util.concurrent.ScheduledThreadPoolExecutor is a subclass of ThreadPoolExecutor and can additionally schedule commands to run after a given delay, or to execute periodically.

ScheduledThreadPoolExecutor Methods

Sr.No.	Method & Description
1	**protected <V> RunnableScheduledFuture<V> decorateTask(Callable<V> callable, RunnableScheduledFuture<V> task)** Modifies or replaces the task used to execute a callable.
2	**protected <V> RunnableScheduledFuture<V> decorateTask(Runnable runnable, RunnableScheduledFuture<V> task)** Modifies or replaces the task used to execute a runnable.
3	**void execute(Runnable command)** Executes command with zero required delay.
4	**boolean getContinueExistingPeriodicTasksAfterShutdownPolicy()** Gets the policy on whether to continue executing existing periodic tasks even when this executor has been shutdown.
5	**boolean getExecuteExistingDelayedTasksAfterShutdownPolicy()**

	Gets the policy on whether to execute existing delayed tasks even when this executor has been shutdown.
6	**BlockingQueue<Runnable> getQueue()** Returns the task queue used by this executor.
7	**boolean getRemoveOnCancelPolicy()** Gets the policy on whether cancelled tasks should be immediately removed from the work queue at time of cancellation.
8	**<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)** Creates and executes a ScheduledFuture that becomes enabled after the given delay.
9	**ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)** Creates and executes a one-shot action that becomes enabled after the given delay.
10	**ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 * period, and so on.
11	**ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)** Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.
12	**void setContinueExistingPeriodicTasksAfterShutdownPolicy (boolean value)** Sets the policy on whether to continue executing existing periodic tasks even when this executor has been shutdown.

13	**void setExecuteExistingDelayedTasksAfterShutdownPolicy (boolean value)** Sets the policy on whether to execute existing delayed tasks even when this executor has been shutdown.
14	**void setRemoveOnCancelPolicy(boolean value)** Sets the policy on whether cancelled tasks should be immediately removed from the work queue at time of cancellation.
15	**void shutdown()** Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
16	**List<Runnable> shutdownNow()** Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
17	**<T> Future<T> submit(Callable<T> task)** Submits a value-returning task for execution and returns a Future representing the pending results of the task.
18	**Future<?> submit(Runnable task)** Submits a Runnable task for execution and returns a Future representing that task.
19	**<T> Future<T> submit(Runnable task, T result)** Submits a Runnable task for execution and returns a Future representing that task.

Example

The following TestThread program shows usage of ScheduledThreadPoolExecutor interface in thread based environment.

Live Demo

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
```

```
public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      final ScheduledThreadPoolExecutor scheduler =
         (ScheduledThreadPoolExecutor)Executors.newScheduledThreadPool(1);

      final ScheduledFuture<?> beepHandler =
         scheduler.scheduleAtFixedRate(new BeepTask(), 2, 2, TimeUnit.SECONDS);

      scheduler.schedule(new Runnable() {

         @Override
         public void run() {
            beepHandler.cancel(true);
            scheduler.shutdown();
         }
      }, 10, TimeUnit.SECONDS);
   }

   static class BeepTask implements Runnable {

      public void run() {
         System.out.println("beep");
      }
   }
}
```

This will produce the following result.

Output

```
beep
beep
beep
beep
```

Java Concurrency - Futures and Callables

java.util.concurrent.Callable object can return the computed result done by a thread in contrast to runnable interface which can only run the thread. The Callable object returns Future object which provides methods to monitor the progress of a task being executed by a thread. Future object can be used to check the status of a Callable and then retrieve the result from the Callable once the thread is done. It also provides timeout functionality.

Syntax

```
//submit the callable using ThreadExecutor
//and get the result as a Future object
Future<Long> result10 = executor.submit(new FactorialService(10));


//get the result using get method of the Future object
```

```
//get method waits till the thread execution and then return the result of the execution.
Long factorial10 = result10.get();
```

Example

The following TestThread program shows usage of Futures and Callables in thread based environment.

Live Demo

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException,
      ExecutionException {

      ExecutorService executor = Executors.newSingleThreadExecutor();

      System.out.println("Factorial Service called for 10!");
      Future<Long> result10 = executor.submit(new FactorialService(10));

      System.out.println("Factorial Service called for 20!");
      Future<Long> result20 = executor.submit(new FactorialService(20));

      Long factorial10 = result10.get();
      System.out.println("10! = " + factorial10);

      Long factorial20 = result20.get();
      System.out.println("20! = " + factorial20);

      executor.shutdown();
   }

   static class FactorialService implements Callable<Long> {
      private int number;

      public FactorialService(int number) {
         this.number = number;
      }

      @Override
      public Long call() throws Exception {
         return factorial();
      }

      private Long factorial() throws InterruptedException {
         long result = 1;

         while (number != 0) {
            result = number * result;
            number--;
            Thread.sleep(100);
         }
         return result;
      }
```

```
    }
}
```

This will produce the following result.

Output

```
Factorial Service called for 10!
Factorial Service called for 20!
10! = 3628800
20! = 2432902008176640000
```

Java Concurrency - Fork-Join framework

The fork-join framework allows to break a certain task on several workers and then wait for the result to combine them. It leverages multi-processor machine's capacity to great extent. Following are the core concepts and objects used in fork-join framework.

Fork

Fork is a process in which a task splits itself into smaller and independent sub-tasks which can be executed concurrently.

Syntax

```
Sum left  = new Sum(array, low, mid);
left.fork();
```

Here Sum is a subclass of RecursiveTask and left.fork() spilts the task into sub-tasks.

Join

Join is a process in which a task join all the results of sub-tasks once the subtasks have finished executing, otherwise it keeps waiting.

Syntax

```
left.join();
```

Here left is an object of Sum class.

ForkJoinPool

it is a special thread pool designed to work with fork-and-join task splitting.

Syntax

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

Here a new ForkJoinPool with a parallelism level of 4 CPUs.

RecursiveAction

RecursiveAction represents a task which does not return any value.

Syntax

```
class Writer extends RecursiveAction {
   @Override
   protected void compute() { }
}
```

RecursiveTask

RecursiveTask represents a task which returns a value.

Syntax

```
class Sum extends RecursiveTask<Long> {
   @Override
   protected Long compute() { return null; }
}
```

Example

The following TestThread program shows usage of Fork-Join framework in thread based environment.

Live Demo

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException,
      ExecutionException {

      int nThreads = Runtime.getRuntime().availableProcessors();
      System.out.println(nThreads);

      int[] numbers = new int[1000];

      for(int i = 0; i < numbers.length; i++) {
         numbers[i] = i;
      }

      ForkJoinPool forkJoinPool = new ForkJoinPool(nThreads);
      Long result = forkJoinPool.invoke(new Sum(numbers,0,numbers.length));
      System.out.println(result);
   }

   static class Sum extends RecursiveTask<Long> {
      int low;
      int high;
```

```
      int[] array;

      Sum(int[] array, int low, int high) {
         this.array = array;
         this.low   = low;
         this.high  = high;
      }

      protected Long compute() {

         if(high - low <= 10) {
            long sum = 0;

            for(int i = low; i < high; ++i)
               sum += array[i];
               return sum;
         } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightResult = right.compute();
            long leftResult  = left.join();
            return leftResult + rightResult;
         }
      }
   }
}
```

This will produce the following result.

Output

```
32
499500
```

Java Concurrency - BlockingQueue Interface

A java.util.concurrent.BlockingQueue interface is a subinterface of Queue interface, and additionally supports operations such as waiting for the queue to become non-empty before retrieving an element, and wait for space to become available in the queue before storing an element.

BlockingQueue Methods

Sr.No.	Method & Description
1	**boolean add(E e)** Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.

2	**boolean contains(Object o)** Returns true if this queue contains the specified element.
3	**int drainTo(Collection<? super E> c)** Removes all available elements from this queue and adds them to the given collection.
4	**int drainTo(Collection<? super E> c, int maxElements)** Removes at most the given number of available elements from this queue and adds them to the given collection.
5	**boolean offer(E e)** Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available.
6	**boolean offer(E e, long timeout, TimeUnit unit)** Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
7	**E poll(long timeout, TimeUnit unit)** Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
8	**void put(E e)** Inserts the specified element into this queue, waiting if necessary for space to become available.
9	**int remainingCapacity()** Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or Integer.MAX_VALUE if there is no intrinsic limit.
10	**boolean remove(Object o)** Removes a single instance of the specified element from this queue, if it is present.

11	**E take()** Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Example

The following TestThread program shows usage of BlockingQueue interface in thread based environment.

Live Demo

```
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class TestThread {

   public static void main(final String[] arguments) throws InterruptedException {
      BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(10);

      Producer producer = new Producer(queue);
      Consumer consumer = new Consumer(queue);

      new Thread(producer).start();
      new Thread(consumer).start();

      Thread.sleep(4000);
   }


   static class Producer implements Runnable {
      private BlockingQueue<Integer> queue;

      public Producer(BlockingQueue queue) {
         this.queue = queue;
      }

      @Override
      public void run() {
         Random random = new Random();

         try {
            int result = random.nextInt(100);
            Thread.sleep(1000);
            queue.put(result);
            System.out.println("Added: " + result);

            result = random.nextInt(100);
            Thread.sleep(1000);
            queue.put(result);
            System.out.println("Added: " + result);

            result = random.nextInt(100);
            Thread.sleep(1000);
            queue.put(result);
            System.out.println("Added: " + result);
         } catch (InterruptedException e) {
            e.printStackTrace();
```

```
         }
      }
   }

   static class Consumer implements Runnable {
      private BlockingQueue<Integer> queue;

      public Consumer(BlockingQueue queue) {
         this.queue = queue;
      }

      @Override
      public void run() {

         try {
            System.out.println("Removed: " + queue.take());
            System.out.println("Removed: " + queue.take());
            System.out.println("Removed: " + queue.take());
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

This will produce the following result.

Output

```
Added: 52
Removed: 52
Added: 70
Removed: 70
Added: 27
Removed: 27
```

Java Concurrency - ConcurrentMap Interface

A java.util.concurrent.ConcurrentMap interface is a subinterface of Map interface, supports atomic operations on underlying map variable. It have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. This interface ensures thread safety and atomicity guarantees.

ConcurrentMap Methods

Sr.No.	Method & Description
1	**default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)**

	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
2	**default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)** If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
3	**default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)** If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
4	**default void forEach(BiConsumer<? super K,? super V> action)** Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
5	**default V getOrDefault(Object key, V defaultValue)** Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
6	**default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)** If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
7	**V putIfAbsent(K key, V value)** If the specified key is not already associated with a value, associate it with the given value.
8	**boolean remove(Object key, Object value)** Removes the entry for a key only if currently mapped to a given value.
9	**V replace(K key, V value)** Replaces the entry for a key only if currently mapped to some value.

10	**boolean replace(K key, V oldValue, V newValue)** Replaces the entry for a key only if currently mapped to a given value.
11	**default void replaceAll(BiFunction<? super K,? super V,? extends V> function)** Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Example

The following TestThread program shows usage of ConcurrentMap vs HashMap.

Live Demo

```
import java.util.ConcurrentModificationException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class TestThread {

   public static void main(final String[] arguments) {
      Map<String,String> map = new ConcurrentHashMap<String, String>();

      map.put("1", "One");
      map.put("2", "Two");
      map.put("3", "Three");
      map.put("5", "Five");
      map.put("6", "Six");

      System.out.println("Initial ConcurrentHashMap: " + map);
      Iterator<String> iterator = map.keySet().iterator();

      try {

         while(iterator.hasNext()) {
            String key = iterator.next();

            if(key.equals("3")) {
               map.put("4", "Four");
            }
         }
      } catch(ConcurrentModificationException cme) {
         cme.printStackTrace();
      }
      System.out.println("ConcurrentHashMap after modification: " + map);

      map = new HashMap<String, String>();

      map.put("1", "One");
      map.put("2", "Two");
      map.put("3", "Three");
      map.put("5", "Five");
```

```
      map.put("6", "Six");

      System.out.println("Initial HashMap: " + map);
      iterator = map.keySet().iterator();

      try {

         while(iterator.hasNext()) {
            String key = iterator.next();

            if(key.equals("3")) {
               map.put("4", "Four");
            }
         }
         System.out.println("HashMap after modification: " + map);
      } catch(ConcurrentModificationException cme) {
         cme.printStackTrace();
      }
   }
}
```

This will produce the following result.

Output

```
Initial ConcurrentHashMap: {1 = One, 2 = Two, 3 = Three, 5 = Five, 6 = Six}
ConcurrentHashMap after modification: {1 = One, 2 = Two, 3 = Three, 4 = Four, 5 = Five, 6 = Six}
Initial HashMap: {1 = One, 2 = Two, 3 = Three, 5 = Five, 6 = Six}
java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextNode(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at TestThread.main(TestThread.java:48)
```

ConcurrentNavigableMap Interface

A java.util.concurrent.ConcurrentNavigableMap interface is a subinterface of ConcurrentMap interface, and supports NavigableMap operations, and recursively so for its navigable sub-maps, and approximate matches.

ConcurrentMap Methods

Sr.No.	Method & Description
1	**NavigableSet<K> descendingKeySet()** Returns a reverse order NavigableSet view of the keys contained in this map.
2	**ConcurrentNavigableMap<K,V> descendingMap()** Returns a reverse order view of the mappings contained in this map.

3	**ConcurrentNavigableMap<K,V> headMap(K toKey)** Returns a view of the portion of this map whose keys are strictly less than toKey.
4	**ConcurrentNavigableMap<K,V> headMap(K toKey, boolean inclusive)** Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
5	**NavigableSet<K> keySet()** Returns a NavigableSet view of the keys contained in this map.
6	**NavigableSet<K> navigableKeySet()** Returns a NavigableSet view of the keys contained in this map.
7	**ConcurrentNavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)** Returns a view of the portion of this map whose keys range from fromKey to toKey.
8	**ConcurrentNavigableMap<K,V> subMap(K fromKey, K toKey)** Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
9	**ConcurrentNavigableMap<K,V> tailMap(K fromKey)** Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
10	**ConcurrentNavigableMap<K,V> tailMap(K fromKey, boolean inclusive)** Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.

Example

The following TestThread program shows usage of ConcurrentNavigableMap.

Live Demo

```
import java.util.concurrent.ConcurrentNavigableMap;
import java.util.concurrent.ConcurrentSkipListMap;
```

```
public class TestThread {

   public static void main(final String[] arguments) {
      ConcurrentNavigableMap<String,String> map =
         new ConcurrentSkipListMap<String, String>();

      map.put("1", "One");
      map.put("2", "Two");
      map.put("3", "Three");
      map.put("5", "Five");
      map.put("6", "Six");

      System.out.println("Initial ConcurrentHashMap: "+map);
      System.out.println("HeadMap(\"2\") of ConcurrentHashMap: "+map.headMap("2"));
      System.out.println("TailMap(\"2\") of ConcurrentHashMap: "+map.tailMap("2"));
      System.out.println(
         "SubMap(\"2\", \"4\") of ConcurrentHashMap: "+map.subMap("2","4"));
   }
}
```

This will produce the following result.

Output

```
Initial ConcurrentHashMap: {1 = One, 2 = Two, 3 = Three, 5 = Five, 6 = Six}
HeadMap("2") of ConcurrentHashMap: {1 = One}
TailMap("2") of ConcurrentHashMap: {2 = Two, 3 = Three, 5 = Five, 6 = Six}
SubMap("2", "4") of ConcurrentHashMap: {2 = Two, 3 = Three}
```

Previous Page Next Page

Enter email for newsletter go