**Java Concurrency**

**Concurrency** is the ability to run several programs or several parts of a program in parallel. Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware. e.g. modern computers has several CPU’s or several cores within one CPU, program can utilize all cores for some part of processing; thus completing task much before in time in comparison to sequential processing.

The backbone of **java concurrency** are threads. A thread is a lightweight process, which has its own call stack, but can access shared data of other threads in the same process. A Java application runs by default in one process. Within a Java application, you can work with many threads to achieve parallel processing or concurrency.

Synchronization in java is achieved using ***synchronized*** keyword.***You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.***

Synchronization import points:

1. Synchronization in java guarantees that no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
4. java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Synchronized block is null. For example, in above code sample if lock is initialized as null, the synchronized (lock) will throw NullPointerException Java Synchronization will throw NullPointerException if object used in java
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
7. It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.

[**Java executor framework**](https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html)

[Java executor framework](https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html) (java.util.concurrent.Executor), released with the JDK 5 is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads.

Creating a thread in java is a very expensive process which includes memory overhead also. So, it’s a good idea if we can re-use these threads once created, to run our future runnables.

**Thread Pools:** Creating new thread for every job (runnable objects) may create performance and memory problem, to overcome this we should go for thread pool.

Thread pool is pool of already created threads ready to do our job, java 1.5 introduced thread pool framework to implements threads pools.

Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

Thread pool framework also known as executors framework.

**We can Thread pool created as follows:**

ExecutorService executor = Executors.newFixedThreadPool(3);

**We can submit a runnable job by using a submit method.**

executor.submit(job)

**Shut down execute service**

executor.shutdown()

**Why Use a Thread Pool?**

Creating and starting a thread can be an expensive process. By repeating this process every time we need to execute a task, we’re incurring a significant performance cost – which is exactly what we were attempting to improve by using threads.

For a better understanding of the cost of creating and starting a thread, let’s see what the JVM actually does behind the scenes:

* it allocates memory for a thread stack that holds a frame for every thread method invocation
* each frame consists of a local variable array, return value, operand stack and constant pool

Basically, the executor replaces the explicit creation and management of a thread.

**The factory methods in the Executors class can create several types of thread pools:**

* newSingleThreadExecutor() – a thread pool with only one thread with an unbounded queue, which only executes one task at a time
* newFixedThreadPool() – a thread pool with a fixed number of threads which share an unbounded queue; if all threads are active when a new task is submitted, they will wait in queue until a thread becomes available
* newCachedThreadPool() – a thread pool that creates new threads as they are needed
* newWorkStealingThreadPool() – a thread pool based on a “work-stealing” algorithm which will be detailed more in a later section

**Note:**

While designing webserver and application servers we can use thread pool concepts.

Default thread pool size: 60

Example:

|  |
| --- |
| **class** PrintJobThread **implements** Runnable {  String name;    **public** PrintJobThread(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  System.***out***.println(name + " job started by: " + Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(5000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(name + " job stoped by: " + Thread.*currentThread*().getName());  }  }  **public** **class** PrintJob {  **public** **static** **void** main(String[] args) {  PrintJobThread[] jobs = { **new** PrintJobThread("Nag"), **new** PrintJobThread("Scala"), **new** PrintJobThread("Java"),  **new** PrintJobThread("angular")};    ExecutorService service = Executors.*newFixedThreadPool*(3);  **for**(PrintJobThread job : jobs) {  service.submit(job);  }  service.shutdown();  }  } |

**Callable Interface**

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable

**Callable**

Callable interface has the call() method. In this method, we have to implement the logic of a task. The Callable interface is a parameterized interface, meaning we have to indicate the type of data the call() method will return.

The Callable interface is a generic interface containing a single call() method – which returns a generic value V:

public interface Callable<V> {

    V call() throws Exception;

}

**Runnable:**

The Runnable interface is a functional interface and has a single run() method which doesn’t accept any parameters and does not return any values

This is suitable for situations where we are not looking for a result of the thread execution, for example, incoming events logging:

public interface Runnable {

    public void run();

}

**Runnable and Callable**

Main difference between Runnable and Callable is that Runnable cannot return any value back to caller but Callable can return value. Another difference is that call() method from Callable can also throw checked exception which was not possible by run() method of Runnable interface

public Object call() throws Exception;

#### Future

Future interface has methods to obtain the result generated by a Callable object and to manage its state.

Callable object to be executed in an executor using the submit()method. This method receives a Callable object as a parameter and returns a Futureobject that we can use with two main objectives –

1. **We can control the status of the task** – we can cancel the task and check if it has finished. For this purpose, we have used the isDone() method to check if the tasks had finished.
2. **We can get the result returned by the call() method**. For this purpose, we have used the get() method. This method waits until the Callable object has finished the execution of the call() method and has returned its result.

If the thread is interrupted while the get() method is waiting for the result, it throws an InterruptedException exception. If the call() method throws an exception, this method throws an ExecutionException exception.

The Future interface provides another version of the get() method i.e. **get(longtimeout,TimeUnitunit)**. This version of the get method, if the result of the task isn’t available, waits for it for the specified time. If the specified period of time passes and the result isn’t yet available, the method returns a null value

Observe that Callable and Future do two different things – Callable is similar to Runnable, in that it encapsulates a task that is meant to run on another thread, whereas a Future is used to store a result obtained from a different thread. In fact, the Future can be made to work with Runnable as well, which is something that will become clear when Executors come into the picture.

* public boolean cancel(boolean mayInterrupt): Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
* public Object get() throws InterruptedException, ExecutionException: Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
* public boolean isDone(): Returns true if the task is complete and false otherwise

**Example:**

|  |
| --- |
| **package** com.mng.thread;  **import** java.util.ArrayList;  **import** java.util.List;  **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;  **class** MyCallable **implements** Callable<Long> {  **private** Long no;  **private** String name;  **public** MyCallable(Long n, String name) {  **this**.no = n;  **this**.name = name;  }  @Override  **public** Long call() **throws** Exception {  System.***out***.println(name + " job started by " + Thread.*currentThread*().getName());  Long result = factorialNos(no);  System.***out***.println(name + " job stoped by " + Thread.*currentThread*().getName());  **return** result;  }  **public** Long factorialNos(Long n) **throws** InterruptedException {  **long** result = 1;  **while** (n != 0) {  result = n \* result;  n = n - 1;  Thread.*sleep*(500);  }  **return** result;  }  }  **public** **class** CallableDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  MyCallable[] myCallables = { **new** MyCallable(4l, "First Thread"), **new** MyCallable(5l, "Second Thread"),**new** MyCallable(1l, "Third Thread"), **new** MyCallable(10l, "fourth Thread") };  ExecutorService executorService = Executors.*newFixedThreadPool*(5);  List<Future<Long>> results = **new** ArrayList<Future<Long>>();  **for** (MyCallable job : myCallables) {  Future<Long> result = executorService.submit(job);  System.***out***.println("return type result: " + result.get()+" And Task done is " + result.isDone());  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  results.add(result);  }    System.***out***.println("future result set process");  **for** (Future<Long> res : results) {  **try** {  System.***out***.println("return type result: " + res.get()+" And Task done is " + res.isDone());  } **catch** (InterruptedException | ExecutionException e) {  e.printStackTrace();  }  }  executorService.shutdown();  }  }  **Output:**  First Thread job started by pool-1-thread-1  First Thread job stoped by pool-1-thread-1  return type result: 24 And Task done is true  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Second Thread job started by pool-1-thread-2  Second Thread job stoped by pool-1-thread-2  return type result: 120 And Task done is true  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Third Thread job started by pool-1-thread-3  Third Thread job stoped by pool-1-thread-3  return type result: 1 And Task done is true  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  fourth Thread job started by pool-1-thread-4  fourth Thread job stoped by pool-1-thread-4  return type result: 3628800 And Task done is true  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  future result set process  return type result: 24 And Task done is true  return type result: 120 And Task done is true  return type result: 1 And Task done is true  return type result: 3628800 And Task done is true |

In the case of runnable job thread doesn’t return anything after completing the job.

If thread is required to return some results after execution then we should go for callable.

Callable interface contains only one method.

**Public Object call() throws Exception;**

If a submitting the callable object to executor then after completing the job thread returns object of type future.

Future object can used to retrieve the results from callable job.

**Difference between Runnable/Callable interfaces:**

|  |  |
| --- | --- |
| **Runnable** | **Callable** |
| If thread not required returning anything after completing the job then we should go for runnable.  Runnable interface contains only one method run ().  run () method return type is void.  Within the run() if there is any chance of checked exception complier we should handle using try catch because we can’t use throws keyword run().  Runnable present in java.lang pacakge | If thread required returning something of the completing job then we should go for callable.  Callable interface contains only one method call ().  Call () method return type is Object.  Within call () if there is any chance of checked exception, we are not requires to handle by using try catch because call() method already throws exception.  Callabe present in java.util.concurent pacakge |

**ThreadLocal**

Java ThreadLocal is used to create thread local variables. We know that all threads of an Object share it’s variables, so the variable is not thread safe. We can use synchronization for thread safety but if we want to avoid synchronization, we can use ThreadLocal variables.

**When to use ThreadLocal?**

For example, consider you are working on a eCommerce application. You have a requirement to generate a unique transaction id for each and every customer request this controller process and you need to pass this transaction id to the business methods in manager/DAO classes for logging purpose. One solution could be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.

To solve that, here you can use ThreadLocal variable. You can generate a transaction id in controller OR any pre-processor interceptor; and set this transaction id in the ThreadLocal. After this, whatever the methods, that this controller calls, they all can access this transaction id from the threadlocal. Also note that application controller will be servicing more that one request at a time and since each request is processed in separate thread at framework level, the transaction id will be unique to each thread and will be accessible from all over the thread’s execution path.

Thread Local class provides thread local variables thread local class maintains values for thread bases

Each thread local object maintains a separate value like, user id, transaction id etc. for each thread that access the object.

Thread can access its local value, can manipulate the value even can remove its value, every part of the code which is executed by the thread, we can access it’s a local variable.

**Example:**

Consider a servle which invoke some business methods we have requirement to generate a unique transaction id for every request and we have to pass this transaction id to the business method for this requirement, we can use thread local to maintain a separate transaction id for ever request, that is for ever thread.

**Note:**  
Thread local interduced in version 2.0 and enhanced in 1.5 version,

Thread local can be associated with thread scope, total code which is executed by thread as accessed to corresponding thread local variable.

A Thread can accessed its own local variable and can’t access others threads local variables.

Once thread entered to dead, state its entire local variable by default eligible for garbage collection.

ThreadLocal tl = new ThreadLocal()

**This class has following methods:**

1. **get()** : Returns the value in the current thread’s copy of this thread-local variable.
2. **initialValue()** : Returns the current thread’s “initial value” for this thread-local variable.
3. **remove()** : Removes the current thread’s value for this thread-local variable.
4. **set(T value)** : Sets the current thread’s copy of this thread-local variable to the specified value.

**Exammple**:

|  |
| --- |
| **package com.mng.thread;**  **import java.text.SimpleDateFormat;**  **import java.util.concurrent.ExecutorService;**  **import java.util.concurrent.Executors;**  **class PrintJobThread implements Runnable {**  **String name;**    **// Used for effective and expiration date**  **private ThreadLocal<SimpleDateFormat> dateParser = new ThreadLocal<SimpleDateFormat>()**  **{**  **@Override**  **protected SimpleDateFormat initialValue()**  **{**  **return new SimpleDateFormat("MM/dd/yyyy");**  **}**  **};**    **public PrintJobThread(String name) {**  **this.name = name;**  **}**  **@Override**  **public void run() {**  **// TODO Auto-generated method stub**  **System.out.println("Thread Name= "+Thread.currentThread().getName()+" default Formatter = "+dateParser.get().toPattern());**  **try {**  **Thread.sleep(5000);**  **} catch (InterruptedException e) {**  **// TODO Auto-generated catch block**  **e.printStackTrace();**  **}**  **//formatter pattern is changed here by thread, but it won't reflect to other threads**  **dateParser.set(new SimpleDateFormat());**  **System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter = "+dateParser.get().toPattern());**  **}**  **}**  **public class PrintJob {**  **public static void main(String[] args) {**  **PrintJobThread[] jobs = { new PrintJobThread("Nag"), new PrintJobThread("Scala"), new PrintJobThread("Java"),**  **new PrintJobThread("angular")};**    **ExecutorService service = Executors.newFixedThreadPool(3);**  **for(PrintJobThread job : jobs) {**  **service.submit(job);**  **}**  **service.shutdown();**  **}**  **}**  **OutPut:**  Thread Name= pool-1-thread-1 default Formatter = MM/dd/yyyy  Thread Name= pool-1-thread-3 default Formatter = MM/dd/yyyy  Thread Name= pool-1-thread-2 default Formatter = MM/dd/yyyy  Thread Name= pool-1-thread-2 formatter = M/d/yy h:mm a  Thread Name= pool-1-thread-3 formatter = M/d/yy h:mm a  Thread Name= pool-1-thread-2 default Formatter = MM/dd/yyyy  Thread Name= pool-1-thread-1 formatter = M/d/yy h:mm a  Thread Name= pool-1-thread-2 formatter = M/d/yy h:mm a |

**Update**: ThreadLocal class is extend in [Java 8](https://www.journaldev.com/2389/java-8-features-with-examples) with a new method withInitial() that takes Supplier functional interface as argument. So we can use lambda expressions to easily create the ThreadLocal instance. For example, above formatter ThreadLocal variable can be defined in one line as below:

**private static final ThreadLocal<SimpleDateFormat> formatter =**

**ThreadLocal.<SimpleDateFormat>withInitial**

**(() -> {return new SimpleDateFormat("yyyyMMdd HHmm");});**

**Inherence Thread Local:**

In child thread if accessing parent thread local variable we will get null value or initial value not parent assigning thread local variable in run method.

**InheritableThreadLocal:**

If use InheritableThreadLocal in child class we will get parent thread local value.

InheritableThreadLocal<String> inheritableThreadLocal = new InheritableThreadLocal<>();

If we want separate value for child thread we want to override childValue() method.

InheritableThreadLocal<String> inheritableThreadLocal = **new** InheritableThreadLocal<String>() {

@Override

**protected** String childValue(String parentValue) {

// **TODO** Auto-generated method stub

**return** "child value";

}

};

This method is called(overridden) within the parent thread before the child thread is started.

* If we want to make parent thread, thread local variable value available to the child thread, then we should go for InheritableThreadLocal class.
* By default, child thread value is exactly the same as parent thread value. But we can provide our own customized value for child thread by overriding childValue method.

|  |
| --- |
| /Java program to illustrate parent thread, ThreadLocal variable  //by default not available to child thread  **class** ParentThread **extends** Thread {  **public** **static** ThreadLocal *gfg\_tl* = **new** ThreadLocal();  **public** **void** run()  {  // setting the new value  *gfg\_tl*.set("parent data");  // returns the ThreadLocal value associated with current thread  System.***out***.println("Parent Thread Value :" + *gfg\_tl*.get());  ChildThread gfg\_ct = **new** ChildThread();  gfg\_ct.start();  }  }  **class** ChildThread **extends** Thread {  **public** **void** run()  {  // returns the ThreadLocal value associated with current thread  System.***out***.println("Child Thread Value :" + ParentThread.*gfg\_tl*.get());  /\* null (parent thread variable  thread local value is not available to child thread ) \*/  }  }  **class** ThreadLocalDemo {  **public** **static** **void** main(String[] args)  {  ParentThread gfg\_pt = **new** ParentThread();  gfg\_pt.start();  }  }  **Output:**  Parent Thread Value:parent data  Child Thread Value:**null** (by **default** initialValue is **null**)  //Java program to illustrate inheritance of customized value  //from parent thread to child thread  **class** ParentThread **extends** Thread {  // anonymous inner class for overriding childValue method.  **public** **static** InheritableThreadLocal *gfg\_tl* = **new** InheritableThreadLocal() {  **public** Object childValue(Object parentValue)  {  **return** "child data";  }  };  **public** **void** run()  {  // setting the new value  *gfg\_tl*.set("parent data");  // parent data  System.***out***.println("Parent Thread Value :" + *gfg\_tl*.get());  ChildThread gfg\_ct = **new** ChildThread();  gfg\_ct.start();  }  }  **class** ChildThread **extends** Thread {  **public** **void** run()  {  // child data  System.***out***.println("Child Thread Value :" + ParentThread.*gfg\_tl*.get());  }  }  **class** ThreadLocalDemo {  **public** **static** **void** main(String[] args)  {  ParentThread gfg\_pt = **new** ParentThread();  gfg\_pt.start();  }  }  Output:  Parent Thread Value:parent data  Child Thread Value:child data |

**Java.util.concurrent.Locks**

**Lock:**

This is the base interface for Lock API. It provides all the features of synchronized keyword with additional ways to create different Conditions for locking, providing timeout for thread to wait for lock. Some of the important methods are lock() to acquire the lock, unlock() to release the lock, tryLock() to wait for lock for a certain period of time, newCondition() to create the Condition etc.

The problem with traditional synchronized keyword we are not having any flexibility to try for lock without waiting thread.

Synchronized blocks don’t offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock

Three is no way maximum waiting time for thread to get lock so that thread will wait until getting the lock, which may create performance problem, which may cause dead lock

If thread a releases a lock then which waiting thread will get that lock we are not having any control this.

There is no api to list out all waiting for a lock.

The synchronized keyword must be used either method level or within the method and it not possible to use across multiple methods.

To overcome above problems sun peoples introduced java.util.concurent.locks package in 1.5 version.

It also several enhancements to the programmer to provide more control on concurrency.

### Java Lock vs synchronized

1. Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
4. synchronized keyword doesn’t provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
5. We can create different conditions for Lock and different thread can await() for different conditions.

Lock object similar to implicit lock acquired by thread to execute synchronized method or synchronized block

Lock implementation provide more extensive operation than traditional implicit locks.

* **void lock()** – acquire the lock if it’s available; if the lock is not available a thread gets blocked until the lock is released

If lock is available, immediately current thread will get that the lock, if lock not alerday avilabe then it will wait until getting the lock

It is exactly same behavior of traditional synchronized keyword.

* **boolean tryLock()** – this is a non-blocking version of lock() method; it attempts to acquire the lock immediately, return true if locking succeeds

To aquire the lock without waiting, if lock is available then thread acquire the lock and returns true.

If lock is not available then this method returns false can continue its execution without waiting, in this case thread never be waiting sate.

if(l.tryLock(){

perform safe operations.

}else{

Perform alternative operation

}

* **boolean tryLock(long timeout, TimeUnit timeUnit)** – this is similar to tryLock(), except it waits up the given timeout before giving up trying to acquire the Lock

If lock is available then the thread will get the lock and continue its execution.

If lock is not available then the thread will wait until specified amount of time, still lock is not available then thread continue execution

If (l.tryLock(1000,Timeunit.MILISECONDS){}

* **void lockInterruptibly()** – this is similar to the lock(), but it allows the blocked thread to be interrupted and resume the execution through a rown java.lang.InterruptedException.

Acquire the lock if it is available and return immediately

If lock is not available then it will wait while waiting if thread is interrupted then thread will not get the lock.

If the current thread is waiting for lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring lock.

* **void unlock() –** unlocks the Lock instance.

To call this method current thread should be owner of the lock otherwise we will get runtime exception illegalMonitostateException

**ReentrantLock**

ReentrantLock is implementation of lock interface and direct child of objects,

Reentrant means a thread can acquire same lock multiple times without any issue internally reentrant lock increments thread personal count whenever we call lock and decrement count value whenever thread call unlock method and lock will be released whenever count reaches.

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method.

ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

**Constructor:**

ReentrantLock reentrantLock = new ReentrantLock()

ReentrantLock reentrantLock = new ReentrantLock(Boolean fairness)

Create reentrant locks with given Fairness policy

If the fairness is true then longest waiting thread can acquire the lock, if it is available that is it is follows first come first serve (FCFS) policy, if fairness is false then which waiting thread will get the chance we can’t expect.

Note: Default value for fairness is false.

|  |
| --- |
| **public void some\_method()**  **{**  **reentrantlock.lock();**  **try**  **{**  **//Do some work**  **}**  **catch(Exception e)**  **{**  **e.printStackTrace();**  **}**  **finally**  **{**  **reentrantlock.unlock();**  **}**    **}** |

The unlock statement is always called in the finally block to ensure that the lock is released even if an exception is thrown in the method body(try block).

**ReentrantLock Methods:**

* **lock():** Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.
* **unlock():** Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.
* **tryLock():** If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free then the method returns false and the thread is not blocked but it exits.
* **tryLock(long timeout, TimeUnit unit):** As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.
* **lockInterruptibly():** This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring lock.
* **getHoldCount():** This method returns the count of the number of locks held on the resource.
* **isHeldByCurrentThread():** This method returns true if the lock on the resource is held by the current thread.
* **Int getQueueLength():** returns how many threads waiting to get the lock.
* **Collection getQueuedThreads():** return collection of thread which are waiting the lock.
* **Boolean hasQueuedThreads():** returns true if any threads waiting to the lock.
* **Boolean isLcoked():** returns true if the lock is acquired by some thread**.**
* **Boolean isFair():** returns true if fairness policy is a set with true value.
* **Boolean getOwner():** returns the thread which acquire the lock.

|  |
| --- |
| **public** **class** ReentrantLockDemo {  **public** **static** **void** main(String[] args) {  ReentrantLock reentrantLock = **new** ReentrantLock();  reentrantLock.lock();  reentrantLock.lock();  System.***out***.println(reentrantLock.isLocked()); // true  System.***out***.println(reentrantLock.isHeldByCurrentThread()); //true  System.***out***.println(reentrantLock.getQueueLength()); // 0  System.***out***.println(reentrantLock.isLocked()); // true  reentrantLock.unlock();;  System.***out***.println(reentrantLock.getHoldCount()); // 1  System.***out***.println(reentrantLock.isLocked()); // true  reentrantLock.unlock();  System.***out***.println(reentrantLock.isLocked()); // false  System.***out***.println(reentrantLock.isFair()); // false  }  }  **Output:**  true  true  0  true  1  true  false  false |

**How to achieve thread safety:**

|  |
| --- |
| **Lock method of ReentrantLock**  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** Display1 {  ReentrantLock reentrantLock = **new** ReentrantLock();  **public** **void** wish(String name) {  reentrantLock.lock();  **for**(**int** i=0; i<= 10; i++) {  System.***out***.println("Good Morning");  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(name);  }  reentrantLock.unlock();  }  }  **class** MyThread1 **extends** Thread {  Display1 d;  String name;  **public** MyThread1(Display1 d, String name) {  **this**.d = d;  **this**.name = name;  }  @Override  **public** **void** run() {  d.wish(name);  }  }  **package** com.mng.thread;  **public** **class** ThreadWithoutSynchronized {  **public** **static** **void** main(String[] args) {  Display1 d = **new** Display1();    MyThread1 m1 = **new** MyThread1(d, "Nag");  MyThread1 m2 = **new** MyThread1(d, "Nagendra");  MyThread1 m3 = **new** MyThread1(d, "scala");    m1.start();  m2.start();  m3.start();  }  }  **TryLock Method example:**  **class** MyThread1 **extends** Thread {  String name;  **static** ReentrantLock *lock* = **new** ReentrantLock();  **public** MyThread1(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  **if**(*lock*.tryLock()) {  System.***out***.println(name +".. got lock and performance safe operations");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  *lock*.unlock();  }finally{  lock.unlock();  } else {  System.***out***.println(name +".. unaable to get lock and hence performance alternative operations");  }  }  }  **public** **class** ThreadWithoutSynchronized {  **public** **static** **void** main(String[] args) {  MyThread1 m1 = **new** MyThread1("First Thread");  MyThread1 m2 = **new** MyThread1("Second Thread");  MyThread1 m3 = **new** MyThread1("Third Thread");    m1.start();  m2.start();  m3.start();  }  }  **OutPut:**  First Thread.. got lock and performance safe operations  Second Thread.. unaable to get lock and hence performance alternative operations  Third Thread.. unaable to get lock and hence performance alternative operations  tryLock(2000) method Example:  **import** java.util.concurrent.TimeUnit;  **import** java.util.concurrent.locks.ReentrantLock;  **class** MyThread1 **extends** Thread {  String name;  **static** ReentrantLock *lock* = **new** ReentrantLock();  **public** MyThread1(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  **do** {  **try** {  **if** (*lock*.tryLock(3000, TimeUnit.***MILLISECONDS***)) {  System.***out***.println(name + ".. got lock and performance safe operations");  Thread.*sleep*(25000);  System.***out***.println(name + ".. releases the lock");  *lock*.unlock();  **break**;  } **else** {  System.***out***.println(name + ".. unaable to get lock and will try again");  }  } **catch** (Exception e) {  e.printStackTrace();  }  } **while** (**true**);  }  }  **public** **class** ThreadWithoutSynchronized {  **public** **static** **void** main(String[] args) {  MyThread1 m1 = **new** MyThread1("First Thread");  MyThread1 m2 = **new** MyThread1("Second Thread");  MyThread1 m3 = **new** MyThread1("Third Thread");    m1.start();  m2.start();  m3.start();  }  }  **Output:**  First Thread.. got lock and performance safe operations  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Second Thread.. unaable to get lock and will try again  Third Thread.. unaable to get lock and will try again  First Thread.. releases the lock  **………..……**  **Note:**  If we want o fairness mechnisam use below constructor.  static ReentrantLock lock = new ReentrantLock(true); |

## Data Structure:

If the lock is already owned by a thread, any other thread trying to acquire the lock will put on a hold and will be waiting in a queue. The thread would be disabled for thread scheduling purpose. The queue here is a double linked list. The lock will know the owning thread, state (0 or 1), head and tail of the linked list. If the node’ successor node is waiting for the lock then its wait state property (-1) will reflect that.

## Waiting Queue:

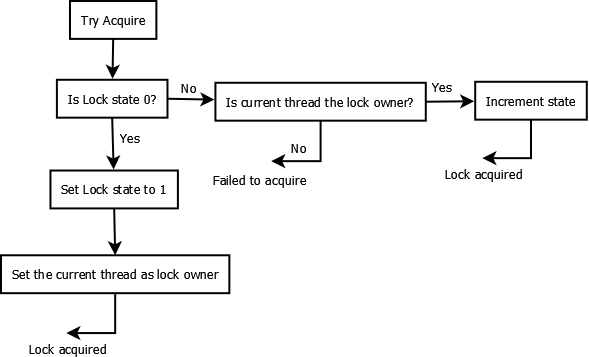
The thread fails to acquire the lock fails as the lock is already acquired by another thread. The thread is added to a double linked list queue where the header is a dummy node with a mode indicating that it is waiting for a signal and the current thread which failed to acquire the lock becomes the tail node and is the next node to header. Since the current thread fails to get the lock, it is parked as it remains blocked till some other thread unblocks it.

**Below example shows a non-fair lock scenario.**

1. Thread A acquires the lock
2. Thread B tries to acquire the lock. It fails as thread A already holds the lock.
3. Since the lock is not available, thread B is put on a queue.
4. Thread A releases the lock.
5. Thread A unparks Thread B so that it can try acquire the lock.
6. Thread C tries to acquire the lock.
7. Since it is still free and is not yet acquired by thread B, thread C manages to acquire it.
8. By the time thread B shows up to acquire the lock, thread C is done with its work and releases the lock.
9. Thread B acquires the lock

## ReentrantLock Recursive Example

If a thread is trying to acquire a lock already owned by some other thread then it is guaranteed to fail but if the owning thread is trying to acquire the same lock once again, its internal hold count will be incremented by one.

[](http://www.javarticles.com/wp-content/uploads/2016/05/tryAcquire.png)

# **Concurrent Collections**

**Need for Concurrent Collection:**

As we already know [Collections](https://www.geeksforgeeks.org/collections-in-java-2/) , which is nothing but collections of Objects, where we deals with the Objects using some pre-defined methods. but There are several problems which occurs when we use Collections concept in multi-threading. The problems, which occurs while using Collections in Multi-threaded application:

* In case of traditional collection, multiple threads can access traditional collection objects (like ArrayList, HashMap etc) simultaneously and there may be a chance of data inconsistency problem and hence these are not thread safe.
* Already existing thread safe collections (Vector, HashTable, synchronized List, synchronized Set, and synchronized Map) performance wise not up to the mark.
* Because for every operation even for read operation also total collection will be locked by only one thread at a time and increasing waiting time of threads.
* Another big problem with traditional collections is while one thread iterating collection, the other thread are not allowed to modify collection object simultaneously if we are trying to modify then we will get concurrentModificationexception.
* Hence, these traditional collection objects are not suitable for scalable multithreaded applications.
* To overcome these problem sun people introduced concurrent collection in version 5.

**Example ConcurrentModificationException:**

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **class** ConcurrentModificationExDemo **implements** Runnable {  **static** ArrayList<String> *list* = **new** ArrayList<>();  @Override  **public** **void** run() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  // Child thread trying to add new  // element in the Collection object  *list*.add("addValue");  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {    *list*.add("value1");  *list*.add("value2");  *list*.add("value3");    // We create a child thread that is  // going to modify ArrayList l.  ConcurrentModificationExDemo obj = **new** ConcurrentModificationExDemo();  Thread t = **new** Thread(obj);  t.start();    // Now we iterate through the ArrayList  // and get exception.  Iterator<String> iterator = *list*.iterator();  **while**(iterator.hasNext()) {  String str = iterator.next();  System.***out***.println(str);  Thread.*sleep*(6000);  }  System.***out***.println("list Values: "+*list*);  }  }  **Output:**  value1  Exception in thread "main" java.util.ConcurrentModificationException  at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)  at java.util.ArrayList$Itr.next(ArrayList.java:851)  at com.mng.thread.concurrent.ConcurrentModificationExDemo.main(ConcurrentModificationExDemo.java:39) |

# **Difference between Traditional Collections and Concurrent Collections in java**

* Concurrent collections are always thread safe.
* When compared with traditional thread safe collection performance is more because of default different locking mechanism using.
* While one, thread interacting collection the other threads are allowed to modify collection in safe manner.
* Hence, concurrent collection never threw concurrent modification exception.
* The important concurrent classes are:

concurrentHashMap.

copyOnWriteArrayList.

copyOnWriteArraySet.

# **Concurrent Collections:**

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

* [**BlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
* [**ConcurrentMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)is a **subinterface of [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html" \t "_blank) that** defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of **ConcurrentMap is [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html" \t "_blank)**, which is a concurrent analog of [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html" \t "_blank).
* [**ConcurrentNavigableMap**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is[ConcurrentSkipListMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), which is a concurrent analog of [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html" \t "_blank).

**ConurrentHashMap:**

* Underlying data structure is hash table.
* Concurrent hash Map allows concurrent read and thread safe update operations.
* To perform read operation thread won’t require any lock, but to perform update operation thread requires lock but it is the lock of only a particular part of map( bucket level lock).
* Instead of whole map concurrent update achieved by internally dividing map into smaller portion which is defined by concurrency level.
* The default concurrency level is 16.
* That is concurrentHashMap is allows simultaneous read operation and simultaneously 16 write (update) operation.
* For both key and value, null not allowed.
* While one, thread iterating the other thread can perform update operations and concurrentHashMap never throw concurrentModificationException.

**Constructors:**

1. Create as empty ConcurrentHashMap with default initial capcity 16 and default fill ration 0.75 and default concurrency level 16

ConcurrentHashMap m1 = new ConcurrentHashMap ();.

1. ConcurrentHashMap m1 = new ConcurrentHashMap (int initialCapcity, float fillRation);
2. ConcurrentHashMap m1 = new ConcurrentHashMap (int initialCapcity, float fillRation, int cuncurenceyLevel);
3. ConcurrentHashMap m1 = new ConcurrentHashMap (Map m);

**What are different between HashMap, HashTable and ConcurrentHashMap.**

**HashMap:**

* Hash map is not thread safe so multiple threads can access the object simultaneously there may be chance of data inconsistency problem.
* HashMap is null allowed both key and value.

**HashTable:**

* HashTable object thread safe at a time one thread allowed to access the object.
* In case of hashTable to perform any operation even read/update but only thread perform operation.
* If one thread perform operating other thread are wait until completing operation first thread.
* It increase waiting time of reaming threads and create performance problem.

HashMap and HashTable total map object will be locked, so multiple threads cannot access objet, only one thread possible.

While one thread iterating, the other thread trying can perform update operation it will throw concurrentModificationException.

To overcome above performance problem to use concept multithreaded scalable application we should go for concurrent hashMap.

**ConcurrentHashMap:**

* Underlying data structure is hash table.
* Concurrent hash map allows concurrent read and thread safe update operations.
* ConcurrentHashMap is thread safe, this map maintain lock concept at bucket level or segment level instead of total object locked.
* In case of hashTable to perform read/write operations, compulsory lock of total object is required.
* But In case of concurrent hashmap to perform read operation thread won’t require any lock so all threads reading object, but to perform write/update operation thread require lock, but lock is bucket level, at time 16 threads to perform update operation because default concurrency level 16.
* Concurrent update achieved by internally dividing map into smaller portions, which is depending concurrency level, The default concurrency level 16.
* ConcurrentHashMap allows any no of read operations, but 16 update operations at a time by default because concurrency level 16
* Null is not allowed both key and value.
* While one thread iterating, the other thread can perform update operation and never throw concurrentModificationException.

while Hashtable or synchronizedMap allows only one thread to work on the map at a time. More specifically, ConcurrentHashMap allows any number of concurrent reader threads and a limited number of concurrent writer threads, and both reader and writer threads can operate on the map simultaneously.

* Reader threads perform retrieval operations such as get, containsKey, size, isEmpty, and iterate over keys set of the map.
* Writer threads perform update operations such as put and remove.

HashMap is a non-threadsafe Map which should not be used by multiple threads.

Hashtable is a thread-safe Map that allows only one thread to execute a read/update operation at a time.

synchronized Map is a thread-safe wrapper on a Map implementation. It is generated by the Collections.synchronizedMap(Map)  factory method. A synchronizedMap also allows only a single thread to work on the map at a time, And ConcurrentHashMap is a thread-safe Map with greater flexibility and higher scalability as it uses a special locking mechanism that enables multiple threads to read/update the map concurrently.

Therefore, you can use ConcurrentHashMap to replace HashMap/Hastable/synchronizedMap for concurrency needs without locking the whole map.

**Concurrent Map interface conations 3 special methods:**

* **putIfAbsent(K key, V value)**: associates the specified key to the specified value if the key is not already associated with a value. This method is performed atomically, meaning that no other threads can intervene in the middle of checking absence and association.

**Note:**

**Put(Object key, Object value) :** if key is present then replaced the old value to new value

**putIfAbsent(object key, object value):** if key present don’t do anything, (doesn’t not add) not present then only add.

* **remove(Object key, Object value)**: removes the entry for a key only if currently mapped to some value. This method is performed atomically.

**If key is associated with value the only remove entry from collection.**

**This method removed only if key and value both matched.**

* **replace(K key, V oldValue, V newValue)**: replaces the entry for a key only if currently mapped to a given value. This method is performed atomically.

If Key present and matched with value then only replace the new value.

**Example above methods:**

|  |
| --- |
| **public** **class** ConcurrentHashMapDemo {  **public** **static** **void** main(String[] args) {  ConcurrentHashMap<Integer, String> map = **new** ConcurrentHashMap<>();  // put method example  map.put(101, "python");  map.put(102, "java");  map.put(103, "scala");  map.put(105, "kafka");  System.***out***.println("Before methods processing map entries");  System.***out***.println(map);    map.putIfAbsent(102, "demo");  System.***out***.println("After putIfAbsent method executed with key present");  System.***out***.println(map);    map.putIfAbsent(104, "angularJs");  System.***out***.println("After putIfAbsent method executed with different key");  System.***out***.println(map);    //remove method  map.remove(101, "C++");  System.***out***.println("After remove method executed with key with differnt value");  System.***out***.println(map);    map.remove(101, "python");  System.***out***.println("After remove method executed with key with associted value");  System.***out***.println(map);    //replace method  map.replace(105, "apache kafka", "spring kafka");  System.***out***.println("After replace method executed with key with differnt value along with new value");  System.***out***.println(map);    map.replace(105, "kafka", "spring kafka");  System.***out***.println("After replace method executed with key with associted value, and new value");  System.***out***.println(map);  }  }  **Output:**  Before methods processing map entries  {101=python, 102=java, 103=scala, 105=kafka}  After putIfAbsent method executed with key present  {101=python, 102=java, 103=scala, 105=kafka}  After putIfAbsent method executed with different key  {101=python, 102=java, 103=scala, 104=angularJs, 105=kafka}  After remove method executed with key with differnt value  {101=python, 102=java, 103=scala, 104=angularJs, 105=kafka}  After remove method executed with key with associted value  {102=java, 103=scala, 104=angularJs, 105=kafka}  After replace method executed with key with differnt value along with new value  {102=java, 103=scala, 104=angularJs, 105=kafka}  After replace method executed with key with associted value, and new value  {102=java, 103=scala, 104=angularJs, 105=spring kafka} |

Example 2:

|  |
| --- |
| **import** java.util.Iterator;  **import** java.util.Set;  **import** java.util.concurrent.ConcurrentHashMap;  **class** ConcurrentModificationExDemo **implements** Runnable {  **static** ConcurrentHashMap<Integer, String> *map* = **new** ConcurrentHashMap<>();  @Override  **public** **void** run() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  // Child thread trying to add new  // element in the Collection object  System.***out***.println("Chiled thread executed and update one entry");  *map*.put(104, "angularJs");  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  // put method example  *map*.put(101, "python");  *map*.put(102, "java");  *map*.put(103, "scala");  *map*.put(105, "kafka");  System.***out***.println("Before methods processing map entries");  System.***out***.println(*map*);    // We create a child thread that is  // going to modify ArrayList l.  ConcurrentModificationExDemo obj = **new** ConcurrentModificationExDemo();  Thread t = **new** Thread(obj);  t.start();    // Now we iterate through the ArrayList  // and get exception.  Set set = *map*.keySet();  Iterator iterator = set.iterator();  **while**(iterator.hasNext()) {  Integer key = (Integer) iterator.next();  System.***out***.println(key+":"+*map*.get(key));  Thread.*sleep*(3000);  }  System.***out***.println("After updating map Values: "+ *map*);  }  }  Output:  Before methods processing map entries  {101=python, 102=java, 103=scala, 105=kafka}  101:python  Chiled thread executed and update one entry  102:java  103:scala  104:angularJs  105:kafka  After updating map Values: {101=python, 102=java, 103=scala, 104=angularJs, 105=kafka} |

**What different between HashMap and ConcurrentHashMap**

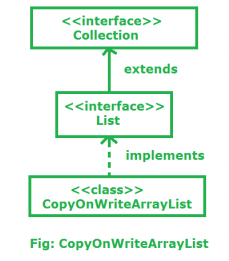
|  |  |
| --- | --- |
| **HashMap** | **ConcurrentHashMap** |
| It is not thread safe.  Relatively performance is high because threads are not required to wait to operate on hash Map.  While one thread iterating hash Map the other threads are not allowed to modify map object otherwise we will get runtime exception  Exception: ConcurrentModificationException.  Iterator of hashmap is fail-fast and it throws  ConcurrentModificationException.  Null is allowed for both Keys and Values.  Introduced in 1.2 version. | It is thread safe.  Relatively performance is low because sometimes threads are required to wait to operate on concurrentHashMap.  While one thread iterating hash Map the other threads are allowed to modify map object in safe manner it won’t throw runtime exception  Exception: ConcurrentModificationException.  Iterator of hashmap is fail-safe and it won’t throws  ConcurrentModificationException.  Null is not allowed for both key and values, otherwise we will get NullPointerException.  Introduced in 1.5 version. |

**What different between ConcurrentHashMap, Synchronized Map and HashTable.**

|  |  |  |
| --- | --- | --- |
| ConcurrentHashMap | SynchronizedMap | HashTable |
| We will get thread safety without locking total map object just with bucket level lock.  At time, multiple threads allowed to operation map object in safe manner.  Read operation can be performed without lock but write operation can be performed with bucket level lock.  While one thread iterating hash Map the other threads are allowed to modify map object in safe manner it won’t throw runtime exception  Exception: ConcurrentModificationException.  Iterator of hashmap is fail-safe and it won’t throws  ConcurrentModificationException.  Null is not allowed for both key and values, otherwise we will get NullPointerException.  Introduced in 1.5 version. | We will get thread safety by locking whole map object.  At time, only one thread allowed to perform any operation on map object.  Every read and write operations required total map object lock.  While one thread iterating hash Map the other threads are not allowed to modify map object otherwise we will get runtime exception  Exception: ConcurrentModificationException.  Iterator of hashmap is fail-fast and it throws ConcurrentModificationException.  Null is allowed for both Keys and Values.  Introduced in 1.2 version. | We will get thread safety by locking whole map object.  At time, only one thread allowed to perform any operation on map object.  Every read and write operations required total map object lock.  While one thread iterating hash Map the other threads are not allowed to modify map object otherwise we will get runtime exception  Exception: ConcurrentModificationException.  Iterator of hashmap is fail-fast and it throws ConcurrentModificationException.  Null is allowed for both Keys and Values.  Introduced in 1.0 version. |

**CopyOnWriteArrayList**

* CopyOnWriteArrayList is concurrent collection and thread safe version of arrayList object.



* CopyOnWriteArrayList is implementation of list interface which child of collect interface.
* Multiple Threads are able to perform update operation simultaneously but for every update operation a separate cloned copy is created, certain point both will synchronized automatically which is taken care by by jvm internally, Therefore there is no effect for threads which are performing read operation
* It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArrayList is the best choice if several read operation and less number of write operation are required to perform.
* It extends object and implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E> and RandomAccess.
* The underlined data structure is grow-able array.
* Insertion is preserved, duplicates are allowed and heterogeneous Objects are allowed.
* Nullinsertion is possiable.
* While one thread iterating CopyOnWriteArrayList, the other threads are allowed to modify and we won’t get concurrentModificationException, that is iterator is fail safe, because of update operation performed on separate cloned copy.
* Iteration of ArrayList can perform remover operation but iteration of CopyOnWriteArrayList cannot perform remove operation, otherwise we will get runtimeException-saying UnsupportedOperationException, because of maintaing separate

**Constructors Summary:**

* CopyOnWriteArrayList c = new CopyOnWriteArrayList(); :Creates an empty list.
* CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection obj);:Creates a list containing the elements of the specified collection, in the order they are returned by the collection’s iterator.
* CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj);:Creates a list holding a copy of the given array.

**Methods Summary:**

* public boolean addIfAbsent(Object obj): the element will be added if and only if list doesn’t contain this element.
* public int addAllAbsent(Collection C): the elements of collection will be added to the list if element are absent and returns no of elements added.

**Difference b/w ArrayList and CopyOnWriteArrayList.**

|  |  |
| --- | --- |
| **ArrayList** | **CopyOnWriteArrayList** |
| it is not thread safe.  While one thread iterating CopyOnWriteArrayList, the other threads are allowed to modify we will get concurrentModificationException, that is iterator is fail fast.  Iterator is fail-fast  Iteration of ArrayList can perform remove operation.  Introduced in 1.2 version | It is thread safe because every update operation will be performed on separate cloned copy will be created.  While one thread iterating CopyOnWriteArrayList, the other threads are allowed to modify and we won’t get concurrentModificationException, that is iterator is fail safe, because of update operation performed on separate cloned copy.  Iterator is fail-safe.  Iteration of copyOnArrayList can’t perform remove operation otherwise we will get runtime exception.  UnSupportedOperationException.  Interduced in 1.5 version. |

**What different CopyOnWriteArrayList, SynchronizedList and Vector:**

**Example 1:**

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.concurrent.CopyOnWriteArrayList;  **public** **class** CopyOnArrayListDemo {  **public** **static** **void** main(String[] args) {  ArrayList<String> list = **new** ArrayList<String> ();  list.add("A");  list.add("B");    CopyOnWriteArrayList<String> l1 = **new** CopyOnWriteArrayList<> ();  l1.addIfAbsent("A");  l1.addIfAbsent("C");  l1.addAll(list);    System.***out***.println("After adding list object into CopyOnWriteArrayList..");  System.***out***.println(l1);    ArrayList<String> l2 = **new** ArrayList<String> ();  l2.add("A");  l2.add("E");    l1.addAllAbsent(l2);  System.***out***.println("After adding l2 object into CopyOnWriteArrayList..");  System.***out***.println(l1);  }  }  **OutPut:**  After adding list object into CopyOnWriteArrayList..  [A, C, A, B]  After adding l2 object into CopyOnWriteArrayList..  [A, C, A, B, E] |

**Example 2:**

|  |
| --- |
| **public** **class** CopyOnArrayListDemo1 **extends** Thread {  **static** CopyOnWriteArrayList<String> *l* = **new** CopyOnWriteArrayList<String>();  **public** **void** run() {  // Child thread trying to  // add new element in the  // Collection object  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {}  *l*.add("D");  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  *l*.add("A");  *l*.add("B");  *l*.add("c");  // We create a child thread  // that is going to modify  // ArrayList l.  CopyOnArrayListDemo1 t = **new** CopyOnArrayListDemo1();  t.start();  // Now we iterate through  // the ArrayList and get  // exception.  Iterator itr = *l*.iterator();  **while** (itr.hasNext()) {  String s = (String) itr.next();  System.***out***.println(s);  Thread.*sleep*(1000);  }  System.***out***.println(*l*);  }  }  **OutPut:**  A  B  c  D  [A, B, c, D]  Note: If we replace in the place CopyOnWriteArrayList into normal ArraList, we will get concurrentModificationException. |

**UnsupportedOperationException Program:**

|  |
| --- |
| **public** **class** CopyOnArrayListDemo2 {  **public** **static** **void** main(String[] args) {  CopyOnWriteArrayList l = **new** CopyOnWriteArrayList();  l.add("A");  l.add("B");  l.add("C");  Iterator itr = l.iterator();    **while** (itr.hasNext())  {  String s = (String)itr.next();    **if** (s.equals("B"))  {  // Can remove  itr.remove();  }  }  System.***out***.println(l);  }  }  **OutPut:**  Exception in thread "main" java.lang.UnsupportedOperationException  at java.util.concurrent.CopyOnWriteArrayList$COWIterator.remove(CopyOnWriteArrayList.java:1176)  at com.mng.thread.concurrent.CopyOnArrayListDemo2.main(CopyOnArrayListDemo2.java:23)  **Note:**  If we replace in the place CopyOnWriteArrayList into normal ArrayList we won’t get any exception, so it removed element from list.  **ArrayList:**  Normal Arralist iterator can perform read and remove operation  **CopyOnWriteArrayList:**  iterator of cow Al only perfom read is allowed cannot perform remove operation. |

**Example 4:**

Every update operation will be performed on separate copy will be created, hence after

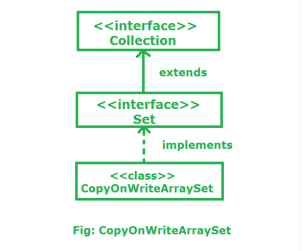
Getting iterator if we trying to perform any modification to the list it will not be reflected to the iterator.

In the case of ArrayList we will get concurrentModificationException>

|  |
| --- |
| **public** **class** CopyOnArrayListDemo2 {  **public** **static** **void** main(String[] args) {  CopyOnWriteArrayList l = **new** CopyOnWriteArrayList();  l.add("A");  l.add("B");  l.add("C");  Iterator itr = l.iterator();  l.add("D");  **while** (itr.hasNext())  {  String s = (String)itr.next();  System.***out***.println(s);  }  System.***out***.println(l);  }  }  Output:  A  B  C  Note: if replace CopyonWriteArrayList into normal ArrayList, we will get ConcurrentModificationException> |

**CopyOnWriteArraySet**

**CopyOnWriteArraySet** is a Set that uses an internal [CopyOnWriteArrayList](https://www.geeksforgeeks.org/copyonwritearraylist-in-java/) for all of its operations. It is introduced in JDK 1.5, we can say that it is thread-safe version of Set.



It share some properties of Set and also has its own properties:

* The internal implementation of CopyOnWriteArraySet is CopyOnWriteArrayList only.
* Insertion order is preserved and duplicates are allowed.
* Multiple Threads are able to perform update operation simultaneously but for every update operation a separate cloned copy is created. As for every update a new cloned copy will be created which is costly. Hence if multiple update operation are required then it is not recommended to use CopyOnWriteArraySet.
* While one thread iterating the Set, other threads can perform updation, here we wont get any runtime exception like ConcurrentModificationException.
* Iterator of CopyOnWriteArraySet class can perform only read only and wont perform deletion, otherwise we will get Run-time exception UnsupportedOperationException.

**Constructors of CopyOnWriteArraySet:**

CopyOnWriteArraySet c = new CopyOnWriteArraySet(): Creates an empty set.

CopyOnWriteArraySet c = new CopyOnWriteArraySet(Collection c): Creates a set containing all of the elements of the specified collection.

NOTE: All methods of CopyOnWriteArraySet is extended from the Collections only because Set interface does not contain any specific methods. CopyOnWriteArraySet does not contain any new methods.

**Note:**

All example are same above copyOnWriteArrayList only replace the copyOnWriteArraySet.

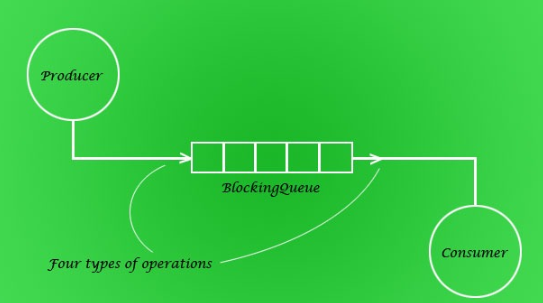
**BlockingQueue**

Java.util.concurrent.BlockingQueue is a java.util.Queue that additionally supports operations,

BlockingQueue is best used in multi-threading and producer-consumer scenarios. While adding an element to a BlockingQueue, if there is no space it can wait till it becomes available. Similarly while retrieving, it will wait till an element is available if it is empty.

BlockingQueue is excellent when you want to skip the complexity involved in wait–notify statements. This BlockingQueue can be used to solve the **producer-consumer proble.**

BlockingQueue slove the



BlockingQueue is helps to contain objects when few threads is inserting objects and other threads are taking objects out of it.

The threads will keep on inserting the objects until the max capacity of blocking queue reached, after which threads will be blocked and they will not be able to inserting further objects. Threads will be blocked state until few objects are taken out from blocking queue by other threads and there is space is available in the blcokingQueue.

The threads will keep on taking the objects out from blockingQueue until there is no objects left in the blockingQueue, after which the threads will be blocked and they will remain in blocked state until few objects are inserted into the blockingQueue by other threads.

The java.util.concurrent.BlockingQueue is an interface and comes with two ready-made implementations then [ArrayLinkedBlockingQueue](http://javarevisited.blogspot.com/2012/12/blocking-queue-in-java-example-ArrayBlockingQueue-LinkedBlockingQueue.html) and [LinkedBlockingQueue](http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html). As the name suggests, one is backed by an array while other is backed by linked list.  
  
BlockingQueue is an interface. Either we should go for custom implementation or choose the existing implementations from the Java JDK. Following the different implementations available for the BlockingQueue in Java.

* [ArrayBlockingQueue](https://javapapers.com/java/java-arrayblockingqueue/)

ArrayBlockingQueue is based on a bounded [Java array](https://javapapers.com/core-java/java-array/). Bounded means it will have a fixed size, once created it cannot be resized.

* [DelayQueue](https://javapapers.com/java/java-delayqueue/)
* [LinkedBlockingDeque](https://javapapers.com/java/java-linkedblockingdeque/)
* [LinkedBlockingQueue](https://javapapers.com/java/java-linkedblockingqueue/)
* [LinkedTransferQueue](https://javapapers.com/java/java-linkedtransferqueue/)
* [PriorityBlockingQueue](https://javapapers.com/java/java-priorityblockingqueue/)
* [SynchronousQueue](https://javapapers.com/java/java-synchronousqueue/).

Methods available in the BlockingQueue.

* **put(E e):** This method inserts the specified element into this queue, waiting if necessary for space to become available.
* **E take():** This method retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

**What is difference between ArrayBlockingQueue and LinkedBlockingQueue**

ArrayBlockingQueue is backed by an array that size will never change after creation. Setting the capacity to Integer.MAX\_VALUE would create a big array with high costs in space. ArrayBlockingQueue is always bounded.

LinkedBlockingQueue creates nodes dynamically until the capacity is reached. This is by default Integer.MAX\_VALUE. Using such a big capacity has no extra costs in space. LinkedBlockingQueue is optionally bounded.

**Example:**

|  |
| --- |
| **Producer program:**  **import** java.util.concurrent.BlockingQueue;  **public** **class** Producer **implements** Runnable {  **private** BlockingQueue<String> queue;  **public** Producer(BlockingQueue<String> queue) {  **this**.queue = queue;  }  @Override  **public** **void** run() {  **for**(**int** i=0; i<20; i++) {  **try** {  Thread.*sleep*(10);  String msg = "message "+ i;  queue.put(msg);  System.***out***.println("Produced "+ msg);  } **catch** (InterruptedException e) {}  }  **try** {  queue.put("exit");  } **catch** (InterruptedException e) {  }}}  **Consumer Program:**  **public** **class** Consumer **implements** Runnable {  **private** BlockingQueue<String> queue;  **public** Consumer(BlockingQueue<String> queue) {  **this**.queue = queue;  }  @Override  **public** **void** run() {  **try** {  **while**(!queue.contains("exit")){  Thread.*sleep*(1000);  System.***out***.println("consumed message: "+ queue.take());  }  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }}  **Main Running Program:**  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  **public** **class** BlockingQueueDemo {  **public** **static** **void** main(String[] args) {  BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<String>(10);  //BlockingQueue<String> linkedBlockingQueue = new LinkedBlockingQueue<String>();  Producer producer = **new** Producer(blockingQueue);  Consumer consumer = **new** Consumer(blockingQueue);  //starting producer to produce messages in queue  **new** Thread(producer).start();  //starting consumer to consume messages from queue  **new** Thread(consumer).start();  System.***out***.println("Producer and Consumer has been started");  }  }  **Console Output:**  Producer and Consumer has been started  Produced message 0  Produced message 1  Produced message 2  Produced message 3  Produced message 4  Produced message 5  Produced message 6  Produced message 7  Produced message 8  Produced message 9  consumed message: message 0  Produced message 10  consumed message: message 1  Produced message 11  consumed message: message 2  Produced message 12  consumed message: message 3  Produced message 13  consumed message: message 4  Produced message 14  consumed message: message 5  Produced message 15  consumed message: message 6  Produced message 16  consumed message: message 7  Produced message 17  consumed message: message 8  Produced message 18  consumed message: message 9  Produced message 19  consumed message: message 10 |

# **Volatile keyword in Java:**

# In java we cannot have synchronized variable, using synchronized keyword with a variable is illegal and will get the completion error instead of using the synchronized variable in java, we can use the java volatile variable which will instruct JVM threads to read the value of volatile variable from main memory and don’t cache it local

# If a variable is not shared between multiple threads then there is no need to use volatile keyword.

Consider below simple example.

**class** SharedObj

{

// Changes made to sharedVar in one thread

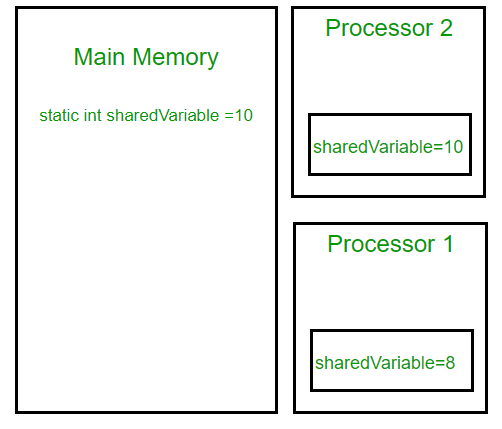
// may not immediately reflect in other thread

**static int** sharedVar = 6;

}

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](https://en.wikipedia.org/wiki/CPU_cache#Write_policies) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of **sharedVariable**may be different in different threads.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/volatile-keyword-in-java.png)

Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called [sequential consistency](https://en.wikipedia.org/wiki/Sequential_consistency)). Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it’s not a good practice to rely on hardware for to ‘fix’ a faulty application.

class SharedObj

{

// volatile keyword here makes sure that

// the changes made in one thread are

// immediately reflect in other thread

static **volatile** int sharedVar = 6;

}

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

|  |
| --- |
| **You need to add/delete an item in an ArrayList when you are iterating the list. You will receive the java.util.ConcurrentModificationException exception. For example, the following code will throw an exception after adding an item into list:**  public class Sample {    public static void main(String[] args) {      List<Integer>  iList = new ArrayList<Integer>();     for (int i = 0; i != 100; i++)       iList.add(i);      int addValue = 1000;     for (Integer i: iList) {       if (i%10 == 0) {         iList.add(addValue++);       }     }    }  **To avoid java.util.ConcurrentModificationException exception, we can add an item through the iterator of list. If we do the same as the above code, the next access item in list via the iterator will generate the same exception.**  public class Sample {    public static void main(String[] args) {      List<Integer>  iList = new ArrayList<Integer>();     for (int i = 0; i != 100; i++)       iList.add(i);      int addValue = 1000;      for (ListIterator<Integer> itr = iList.listIterator(); itr.hasNext();) {       Integer i = itr.next();       if (i%10 == 0) {         itr.add(addValue++);       }     }  } |

Read more: <http://www.java67.com/2015/12/producer-consumer-solution-using-blocking-queue-java.html#ixzz5SqINh6cV>

Read more:z

<https://www.codejava.net/java-core/concurrency?start=20>

<https://www.codejava.net/java-core/concurrency/java-concurrent-collection-concurrenthashmap-examples>

<https://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html#ixzz5SB2jYx3y>