

## Multithreading in java

### Thread

It's a lightweight process that does some work

#### What is Multithreading in Java

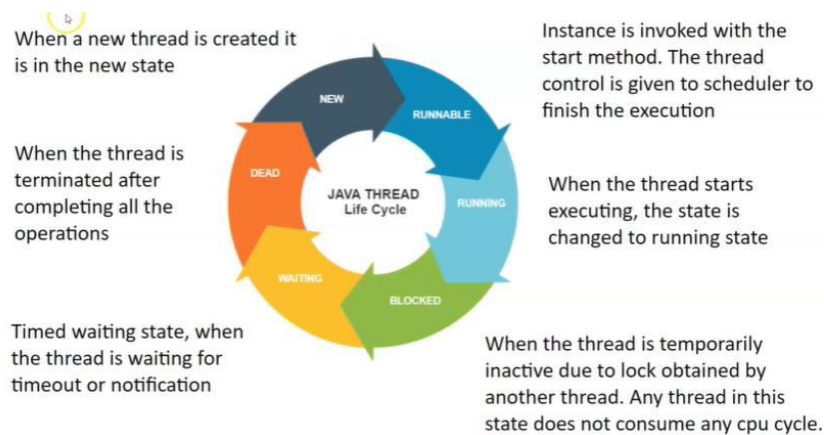
Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU

There are 2 types of threads - userthread and daemon thread.

When an application first begins, a user thread is created.

From this thread we can spun as many user and daemon threads we need.

Daemon threads are used for the cleanup tasks in background.



#### 1. Sequential programming

Create a supervisor class and 2 workers. Supervisor should start the worker1 and when it finished worker 2 should be started by supervisor's order

```
public class SupervisorExample {
    public static void main(String[] args){
        Worker1 worker1 = new Worker1();
        Worker2 worker2 = new Worker2();
        try {
            worker1.executeWork();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        worker2.executeWork();
    }
}

class Worker1 {
    public void executeWork() throws InterruptedException {
        for (int i=0; i<10; i++){
            Thread.sleep(100);
            System.out.println("Worker 1 is executing task");
        }
    }
}
```

Worker2 has been written with try catch . So supervisor is not using try catch for it

#### 2. Parallel programming

It can be done both by class Thread and Runnable interface

Thread code is written inside run() and start() is used to call it

```

public class SupervisorExampleWithThread {
    public static void main(String[] args) {
        ParallelWorker1 parallelWorker1 = new ParallelWorker1();
        ParallelWorker2 parallelWorker2 = new ParallelWorker2();

        parallelWorker1.start();
        parallelWorker2.start();
    }
}

class ParallelWorker1 extends Thread{
    @Override
    public void run() {
        for (int i=0;i<10;i++){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("The worker is executing task: " + i);
        }
    }
}

```

## Lambda expression

A lambda expression in Java has these main parts:

- **No name** – as this is anonymous function there is no name needed
- **Parameter list**
- **Body** – This is the main part of the function.
- **No return type** – You don't need to mention the return type in lambda's expression. The java 8+ compiler is able to infer the return type by checking the code.

## Functional interface

Lambda is mainly used to implement functional interfaces. Any interface with a SAM(Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

```

@FunctionalInterface
public interface Runnable {
    public abstract void run();
}

@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}

```

To call a functional interface's method, lambda can be used. It reduces the code

```

public class HelloWorldLambda {
    public static void main(String[] args) {
        //implementing sayHelloWorld Using Lambda
        HelloWorldInterface helloWorldInterface = () -> {
            return "Hello World";
        };

        System.out.println(helloWorldInterface.sayHelloWorld());
    }
}

```

As HelloWorldInterface(It's a SAM) has only 1 method, so using lambda, the program knows that func's implementation is given.

Notice implements Interfacename is also not given

## Runnable

Traditional way:

```

public class RunnableExample {
    public static void main(String[] args) {
        //Runnable Traditional example
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                int sum=0;
                for (int i=0; i<10; i++)
                    sum += i;
                System.out.println("Traditional: " + sum);
            }
        };
        new Thread(runnable).start();
    }
}

```

Using lambda

```

//Implement using Lambda
Runnable runnable1 = () -> {
    int sum =0;
    for (int i=0; i<10; i++) {
        sum += i;
    }
    System.out.println("Runnable Lambda: " + sum);
};

new Thread (runnable1).start();

```

Even a shorter way to create and run thread using lambda with thread

```

//Implement using Thread with Lambda
new Thread ( () -> {
    int sum=0;
    for (int i=0; i<10; i++)
        sum = sum + i;
    System.out.println("Thread Lambda: " + sum);
}).start();

```

## Callable

Callable can return something unlike runnable

```
Callable callable = () -> {
    int sum = 0;
    for (int i=array.length/2; i<array.length;i++){
        sum = sum + array[i];
    }
    return sum;
};
```

```
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
List<Callable<Integer>> taskList = Arrays.asList(callable1, callable2);
List<Future<Integer>> results = executorService.invokeAll(taskList);
```

```
int k=0;
int sum=0;
for (Future<Integer> result: results){
    sum = sum + result.get();
    System.out.println("Sum of " + ++k + " is: " + result.get());
}
```

## Parallel processing using Runnable

Supervisor and workers way

```
class Worker2Parallel implements Runnable {
    int[] array;
    int sum=0;
    @Override
    public void run() {
        for (int i=array.length/2; i<array.length; i++){
            sum = sum + array[i];
        }
        SumOfNumberUsingRunnableExample.add(sum);
    }
}
```

The above worker has one parameterized constructor also

```

public static void main(String[] args) throws InterruptedException {
    Worker1Parallel worker1Parallel = new Worker1Parallel(numbers);
    Worker2Parallel worker2Parallel = new Worker2Parallel(numbers);

    Thread thread1 = new Thread(worker1Parallel);
    Thread thread2 = new Thread(worker2Parallel);

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    System.out.println("Sum of 5000 integers in parallel is : " + sum);
}

```

Join() is used to wait for the threads to finish its execution before moving to next line

Now, let's use lambda to reduce the lines of code

```

Thread thread1 = new Thread(() -> {
    for (int i=0; i<numbers.length/2; i++){
        add(numbers[i]);
    }
});

Thread thread2 = new Thread(() -> {
    for (int i=numbers.length/2; i<numbers.length; i++){
        add(numbers[i]);
    }
});

thread1.start();
thread2.start();
thread1.join();
thread2.join();

public synchronized static void add (int toAdd){
    sum = sum + toAdd;
}

```

Synchronised keyword serializes the threads i.e if thread1 enters this method, then let it finish then only thread2 will be allowed to enter

## JOINS

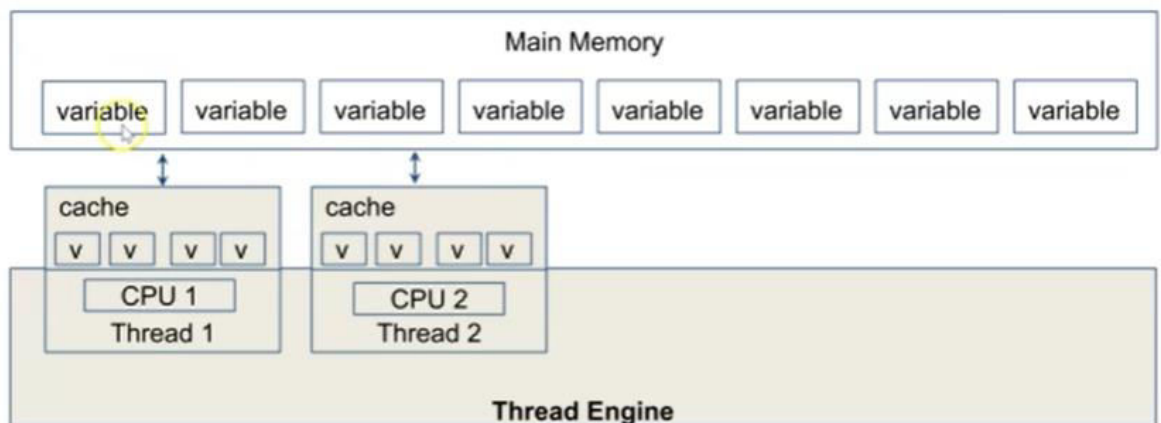


1. The join method allows one thread to wait for the completion of another thread
2. When we invoke the join() on a thread, the calling thread goes into a waiting state. It remains in the waiting state until the referenced thread terminates
3. The join method will keep waiting if the referenced thread is blocked which can become an issue as the calling thread will become non-responsive. Java provided two overloaded version of the join() method that takes timeout period
4. Join throws InterruptedException if the referenced thread is interrupted
5. If the referenced thread is unable to start or already terminated join will return immediately

## Volatile

### volatile

Volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory



06

Every read of volatile variable is from RAM, don't make a variable volatile if you don't have to. Cache is much faster than RAM

## Volatile

1. Variable can be defined as volatile, methods and classes cannot be volatile
2. Volatile keywords guarantee that the variable will be read from main memory (RAM) and not from thread local cache
3. In Java reads and writes are atomic for all variables declared volatile
4. Java volatile keyword doesn't mean atomic operation, it's a common misconception that declaring variable volatile will make all operation on the variable atomic. You still need to ensure exclusive access using synchronized method or block
5. If a variable is not shared between threads, don't use volatile
6. Every read of volatile variable is from RAM, don't make a variable volatile if you don't have to. Cache is much faster than RAM

## Deadlock

Dono eke k lock rk k baithe hai

Livelock is like deadlock in the sense that two (or more) processes are blocking each other but with livelock, each process is actively trying to resolve the problem on its own (like reverting back and retry). A livelock happens when the combination of these processes efforts to resolve the problem make it impossible for them to ever terminate.

An example of livelock is; a husband and wife are trying to eat soup, but only have one spoon between them. Each spouse is too polite, and will pass the spoon if the other has not yet eaten.

## Solution – Synchronization

### Thread Synchronization

There are 2 types of thread synchronization

1. Mutual Exclusive
  - a. Synchronized Method
  - b. Synchronization block
  - c. Static synchronization
2. Cooperation (Inter-thread communication)
  - a. Wait ()
  - b. Notify()
  - c. NotifyAll()

If an object or a block is declared as synchronized than only one thread can access that object or block at a time. No other thread can take the object or block until it is available.

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.

```
public class FastFoodRestaurantSynchronizedMethod {
    private String lastClientName;
    private int numberOfBurgersSold;

    public void synchronized buyBurger(String clientName){
        alongRunningProcess();
        this.lastClientName=clientName;
        numberOfBurgersSold++;
        System.out.println(clientName + "bought a burger");
    }

    public void alongRunningProcess(){
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

4 threads are calling buyBurger(). And it is using the instance variable-lastClientName and numberOfBurgersSold for this. So, we r using synchronized here. But the issue is it makes the execution a bit slow bcz everytime a thread calls this method, it has to wait for that thread's execution to get over.

So, change it to synchronised block withonly required lines like below

```

public class FastFoodRestaurantSynchronizedBlock {
    private String lastClientName;
    private int numberOfBurgersSold;

    public void buyBurger(String clientName){
        alongRunningProcess();
        System.out.println(clientName + "bought a burger");
        synchronized (this) {
            this.lastClientName = clientName;
            numberOfBurgersSold++;
        }
    }
}

```

## Wait, notify

These methods need to be called from synchronized context, otherwise an exception `IllegalMonitorStateException` is thrown.

**wait():** When you call wait on the object it tells the thread to give up the lock and go to sleep state until some other threads enters in same sold and calls notify or notify all.

**notify():** When you call notify on the object it wakes one thread waiting for the object. If multiple threads are waiting for the object it will call one of them. Depending on the OS implementation it will call one of the waiting thread.

**notifyAll():** notifyAll will wake up all threads waiting on the object. Which thread will be called first is depending on the thread priority and OS implementation.

```

new Thread () ->{
    synchronized (course){
        System.out.println(Thread.currentThread().getName() + " is waiting for the course: "
            + course.getTitle());
        try {
            course.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " the course: " + course.getTitle()
            + " is now completed");
    }
}, name: "StudentB").start();

```

course is a final variable in this class  
The notifier thread is written below

```

new Thread () ->{
    synchronized (course){
        System.out.println(Thread.currentThread().getName() + " is starting a new course: "
            + course.getTitle());
        try {
            Thread.sleep( millis: 4000);
            course.notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}, name: "Syed Ahmed").start();

```



But we need to make sure that notifier thread is started after the threads waiting for the course. We can add `thread.sleep(200)` before starting notifier Thread.

## Locks

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java synchronized blocks.

Since lock is an interface, we need to use one of its implementation to use locks in applications. `ReentrantLock` is one such implementation of lock interface.

### Difference between Lock Interface and synchronized keyword

The main difference between a lock and synchronized block are:

1. Synchronized block does not provide a timeout. Using `Lock.tryLock(long timeout, TimeUnit timeUnit)` it is possible
2. The synchronized block must be fully implemented in a single method. A lock can have its call to lock and unlock in separate methods.

## Reentrant Lock

- Reentrant lock allow thread 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. When the thread unlocks the resource the lock would go to the thread which has been waiting for the longest time. This fairness mode is setup by passing `true` to the constructor of the lock.

# Reentrant Lock

## Reentrant lock methods:

**lock():** call to the lock() method increments the hold count by 1 and gives the lock if the shared resources is free

**unlock():** call to the unlock() method decrements the hold count by 1, when this count is reached to zero the resource is released

**trylock():** if the shared resource is available the trylock will lock the resource and increment the hold count to 1. If the shared resource is not available it will exit instead of waiting

**trylock(long timeout, TimeUnit unit):** the thread wait for certain amount of time specified as a parameter before exiting if the shared resource is not available

**lockInterruptibly():** If a thread is waiting for a lock but some other thread request the lock, then the current thread will be interrupted and return immediately without acquiring lock

```
public class BankTransfer {
    private double balance;
    private int id;
    private String accountName;
    final Lock lock = new ReentrantLock();

    public BankTransfer (int id, double balance, String accountName){
        this.id=id;
        this.balance=balance;
        this.accountName=accountName;
    }

    public boolean withdraw (double amount) throws InterruptedException {
        if (this.lock.tryLock()){
            Thread.sleep( millis: 100);
            balance -= amount;
            this.lock.unlock();
            return true;
        }
        return false;
    }

    public boolean deposit (double amount){
        if(this.lock.tryLock()){
            Thread.sleep( millis: 100);
            balance += amount;
            this.lock.unlock();
            return true;
        }
        return false;
    }
}
```

```

public boolean transfer (BankTransfer to, double amount) throws InterruptedException {
    if (withdraw(amount)){
        System.out.println("Withdrawing amount: " + amount + " from: " + accountName);
        if(to.deposit(amount)){
            System.out.println("Depositing amount: " + amount + " to: " + to.accountName);
            return true;
        }
        else {
            System.out.println("Failed to acquire both locks: refunding " + amount + " to: " + accountName);
            while (!deposit(amount))
                continue;
        }
    }
    return false;
}

public static void main(String[] args) {
    BankTransfer studentBankAccount = new BankTransfer( id: 1, balance: 50000, accountName: "StudentA");
    BankTransfer universityBankAccount = new BankTransfer( id: 1, balance: 100000, accountName: "University");

    System.out.println("Starting balance of account are: University: " + universityBankAccount.balance
        + " student: " + studentBankAccount.balance);

    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    Thread t = new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + " says :: Executing Transfer");
        try {
            while (!studentBankAccount.transfer(universityBankAccount, amount: 1000)) {
                Thread.sleep( millis: 100);
                continue;
            }
        }catch (InterruptedException ie){
            ie.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " says transfer is successful");
    });

    for (int i=0;i<20; i++){
        service.submit(t);
    }
}

```

Even after shutdown there might be deadlock. So to handle that below code is written to terminate after 24 hrs

```

service.shutdown();
try {
    while (!service.awaitTermination( timeout: 24L, TimeUnit.HOURS)) {
        System.out.println("Not Yet. still waiting for termination");
    }
}catch (InterruptedException e){
    e.printStackTrace();
}

```

## Semaphores

Semaphore is a class in java.util.concurrent package.

Semaphores are generally used to restrict the number of threads to access resources.

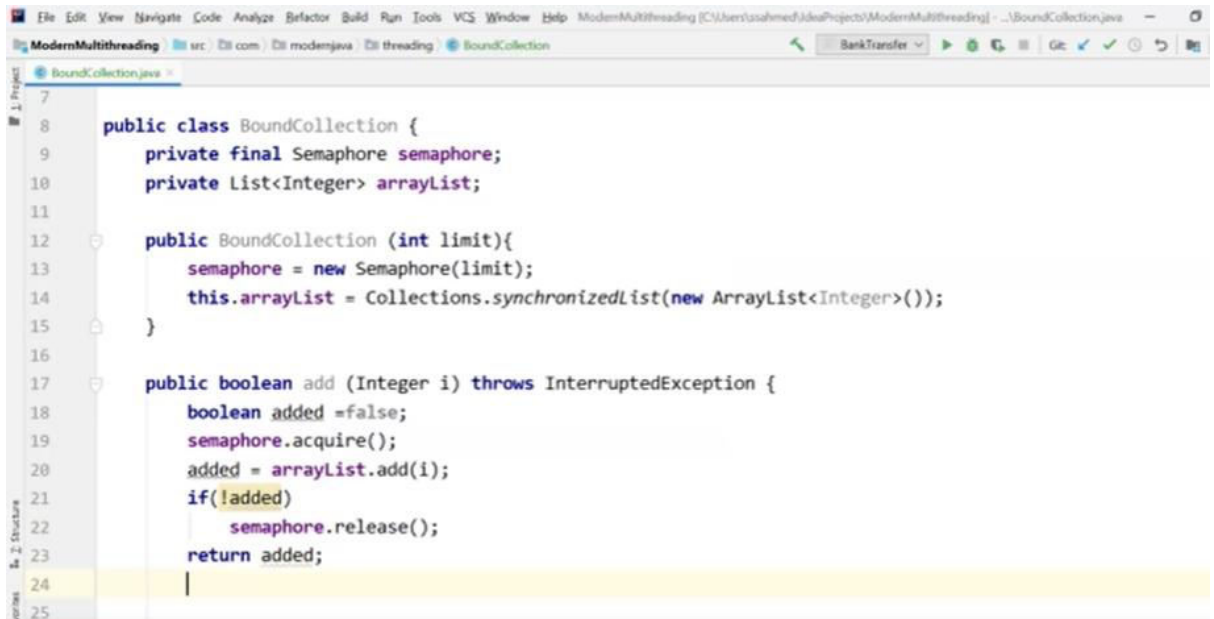
Semaphore basically maintains a set of permits through the use of counter.

To access the resource, a thread must be granted a permit from the semaphore.

If the counter is greater than zero, then access is allowed. If it is zero, then access is denied.

### Examples of Semaphores:

1. Restricting number of connections to the database
2. Bounding a collection



```
7
8 public class BoundCollection {
9     private final Semaphore semaphore;
10    private List<Integer> arrayList;
11
12    public BoundCollection (int limit){
13        semaphore = new Semaphore(limit);
14        this.arrayList = Collections.synchronizedList(new ArrayList<Integer>());
15    }
16
17    public boolean add (Integer i) throws InterruptedException {
18        boolean added =false;
19        semaphore.acquire();
20        added = arrayList.add(i);
21        if(!added)
22            semaphore.release();
23        return added;
24    }
25
```

Doubts in code

```
public static void main(String[] args) {
    final BoundCollection boundCollection = new BoundCollection( limit: 10);

    new Thread(() -> {
        for (int i=0;i<20;i++){
            try {
                if (boundCollection.add(i))
                    System.out.println(Thread.currentThread().getName() + " adding value: " + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Executors

# Executor

## How does Executor work

- The work of an executor is to execute tasks.
- The executor picks up a thread from the threadpool to execute a task.
- If a thread is not available and new threads cannot be created then the executor stores these tasks in a queue. A task can also be removed from the queue

## ThreadPoolExecutor

- Executor is the base interface of the executor framework and has only one method `execute(Runnable Command)`.
- `ExecutorService` and `ScheduledExecutorService` are two other interface which extends executor.
- `ThreadPoolExecutor` is an implementation of the `ExecutorService` interface. The `ThreadPoolExecutor` executes the given tasks using one of its internally pool threads.

1:31 / 4:57

# Thread pool executors

There are five types of thread pool executors with pre-build methods in Executors interface

1. **Fixed thread pool executor:** Creates a thread pool of fixed number of threads. Any task over the number of threads will wait in the queue until the thread is available

```
ThreadPoolExecutor executor = Executors.newFixedThreadPool(5);
```

2. **Cached thread pool executor:** Creates a thread pool that creates new threads as needed. It will reuse the previously constructed threads when they are available. Use this thread pool with caution as this can bring down system if the number of threads goes beyond what the system can handle

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

3. **Scheduled thread pool executor:** Creates a thread pool that can schedule commands to run after a given delay or to execute periodically

```
ThreadPoolExecutor executor = Executors.newScheduledThreadPool(10);
```

4. **Single thread pool executor:** Creates single thread to execute all tasks, use it when you have one task to execute

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

5. **Work stealing thread pool executor:** Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at a single point of time, in multiprocessor machines



There are a few different ways to delegate tasks to an ExecutorService.

1. **execute (Runnable)** : It takes Runnable implementation and execute it asynchronously
2. **submit (Runnable)** : It takes Runnable implementation and returns a future object. The future object can be used to check if the Runnable has finished executing
3. **Submit (Callable)** : It takes Callable implementation and returns a future object
4. **invokeAny()** : It takes collection of callable objects. Invoking this method does not return any future but returns the result of one of the callable objects
5. **invokeAll()** : This method invokes all callable objects passed as parameters. It returns the future objects which can be used to get the results of the execution of each Callable

```
Runnable runnable = () ->{
    try {
        TimeUnit.MILLISECONDS.sleep( timeout: 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " Finished executing at: " + LocalDateTime.now());
};
```

```
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
System.out.println("First Example - executing task with execute() method ");
executor.execute(runnable);
```

```
System.out.println("Second Example - executing task with submit() method");
Future<String> result = executor.submit(runnable, result: "COMPLETED");
```

```
while (!result.isDone()){
    System.out.println("The method return value: " + result.get());
}
```

```
Executor.shutdown()
```

```

Callable<String> callable = () ->{
    TimeUnit.MILLISECONDS.sleep( timeout: 1000);
    return "Current Time is: " + LocalDateTime.now();
};

ExecutorService executor = Executors.newFixedThreadPool( nThreads: 1);
List<Callable<String>> taskList = Arrays.asList(callable, callable);

System.out.println("First example using invokeAll");
List<Future<String>> results = executor.invokeAll(taskList);

for (Future<String> result: results){
    System.out.println();
}
System.out.println("Executing callable using submit");
Future<String> result = executor.submit(callable);

while (!result.isDone()){
    System.out.println("The method return value: " + result.get());
}

executor.shutdown();

```

## CountDownLatch

### CountDownLatch

CountDownLatch is a synchronization aid which allows one Thread to wait for one or more Threads before starts processing. Simply put, a CountDownLatch has a counter field, which you decrement as required and use it to block a calling thread until it's counted down to zero.

While doing parallel processing we will initiate the CountDownLatch with the same value of the counter as the number of threads we want it to wait for. Then, we will call await() method in the main thread and call the countdown() to decrease the counter inside each thread the main thread waiting for to complete.

### CountDownLatch

#### Methods of CountDownLatch

**void await():** Waits until the latch has counted down to zero

**boolean await(long timeout, TimeUnit unit):** similar to wait but waits only for given time and returns false when not reached zero

**void countdown():** Decrements the number in the latch. Notifies awaiting threads when reached zero

**long getCount():** Returns current value of latch

```

final static int total = IntStream.rangeClosed(0,5000).sum();
final static CountDownLatch countDownLatch = new CountDownLatch(2);

public static void main(String[] args) {
    Callable callable1 = () -> {
        int sum =0;
        for (int i=0;i<array.length/2; i++){
            sum = sum + array[i];
        }
        countDownLatch.countDown();
        return sum;
    };
}

```

Callable2 is also created in void main()

```

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
Future<Integer> sum1 = executorService.submit(callable1);
Future<Integer> sum2 = executorService.submit(callable2);

System.out.println("Count Down latch count before calling the await: " + countDownLatch.
countDownLatch.await();
System.out.println("Count DownLatch count after await: " + countDownLatch.getCount());

int sum= sum1.get() + sum2.get();

```

Executorservice.shutdown();

## CyclicBarrier

CyclicBarrier is synchronisation aid that allows a set of threads to wait for each other to reach a common barrier point. The threads who reached the barrier point has to wait for other threads to reach. As soon as all the threads have reached the barrier point, all of them are released to continue.

A CyclicBarrier is reusable, so it's more like a racing tour where everyone meets at a waypoint before proceeding on the next leg of the tour. Cyclicbarrier can be reused after the waiting threads are released and that is where it is different than CountdownLatch. We can reuse CyclicBarrier by calling reset() method which resets the barrier to its initial state.

**Example of CyclicBarrier:** A multithreaded download manager. The download manager will start multiple threads to download each part of the file simultaneously. Suppose that you want the integrity check for the downloaded pieces to be done after a particular time interval. Here cyclicbarrier plays an important role. After each time interval, each thread will wait at the barrier so that thread associated with cyclicbarrier can do the integrity check. This integrity check can be done multiple times thanks to CyclicBarrier

## CyclicBarrier Example

1. We will be implementing the sum of 15000 numbers with CyclicBarrier in three steps
2. In the first step we will sum numbers from 0-5000 using 2 thread
3. In the second step we will sum numbers from 5001 - 10000 using 2 threads
4. In the third step we will sum numbers from 10001 - 15000 using 2 threads
5. At the end of each step all the thread will wait for each other to complete and then proceed to the next step using CyclicBarrier awaits() method
6. We print the total sum after the 3 steps

```
final static CyclicBarrier cyclicBarrier = new CyclicBarrier( parties: 3);
```

```
public static void main(String[] args) {
```

```
    Callable callable1 = () -> {
```

```
        int sum=0;
```

```
        sum = sum + calculateSum( start: 0, end: array1.length/2, array1);
```

```
        cyclicBarrier.await();
```

```
        sum = sum + calculateSum( start: 0, end: array2.length/2, array2);
```

```
        cyclicBarrier.await();
```

```
        sum = sum + calculateSum( start: 0, end: array3.length/2, array3);
```

```
        return sum;
```

```
    }; I
```

```
}
```

```
Callable callable1 = () -> {
```

```
    int sum=0;
```

```
    sum = sum + calculateSum( start: 0, end: array1.length/2, array1);
```

```
    cyclicBarrier.await();
```

```
    sum = sum + calculateSum( start: 0, end: array2.length/2, array2);
```

```
    cyclicBarrier.await();
```

```
    sum = sum + calculateSum( start: 0, end: array3.length/2, array3);
```

```
    return sum;
```

```
I
```

```
};
```

```
Callable callable2 = () -> {
```

```
    int sum =0;
```

```
    sum = sum + calculateSum( start: array1.length/2, array1.length,array1);
```

```
    cyclicBarrier.await();
```

```
    sum = sum + calculateSum( start: array2.length/2, array2.length,array2);
```

```
    cyclicBarrier.await();
```

```
    sum = sum + calculateSum( start: array2.length/2, array3.length, array3);
```

```
    cyclicBarrier.await();
```

```
    return sum;
```





```

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
Future<Integer> sum1 = executorService.submit(callable1);
Future<Integer> sum2 = executorService.submit(callable2);

System.out.println("Calculating first sum ");
cyclicBarrier.await();
System.out.println("First sum is calculated");

System.out.println("Calculating second sum ");
cyclicBarrier.await();
System.out.println("Second sum is calculated");

System.out.println("Calculating third sum ");
cyclicBarrier.await();
System.out.println("Third sum is calculated");

```

## Blocking Queue

# BlockingQueue

The Java BlockingQueue interface, `java.util.concurrent.BlockingQueue`, represents a queue which is thread safe to put elements into, and take elements out of from. In other words, multiple threads can be inserting (enqueue) and taking elements (dequeue) concurrently from a Java BlockingQueue, without any concurrency issues arising.

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. If a thread tries to take an element and there are none left in the queue, the thread can be blocked until there is an element to take and similarly the thread is blocked if the queue is full and it tries to add an element to the queue. Whether or not the calling thread is blocked depends on what methods you call on the BlockingQueue.

## BlockingQueue Methods

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future. It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a `NullPointerException`.

	Throws Exception	Special Value	Blocks	Times Out
<b>Insert</b>	<code>add(o)</code>	<code>offer(o)</code>	<code>put(o)</code>	<code>offer(o, timeout, timeunit)</code>
<b>Remove</b>	<code>remove(o)</code>	<code>poll()</code>	<code>take()</code>	<code>poll(timeout, timeunit)</code>
<b>Examine</b>	<code>element()</code>	<code>peek()</code>		

1. **Throws Exception:** If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value:** If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks:** If the attempted operation is not possible immediately, the method call blocks until it is.
4. **Times Out:** If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).



BlockingQueue is an interface so you need to use one of its implementation. The java.util.concurrent package has the following implementation of BlockingQueue:

1. **ArrayBlockingQueue:** ArrayBlockingQueue class is backed by an array and it is a bounded blocking queue. By bounded it means that the size of the Queue is fixed. Once created, the capacity cannot be changed
2. **DelayQueue:** DelayQueue is a specialized Priority Queue that orders elements based on their delay time. It means that only those elements can be taken from the queue whose time has expired
3. **LinkedBlockingQueue:** LinkedBlockingQueue is an optionally-bounded blocking queue based on linked nodes. It means that the LinkedBlockingQueue can be bounded, if its capacity is given, else the LinkedBlockingQueue will be unbounded
4. **PriorityBlockingQueue:** PriorityBlockingQueue is an unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations
5. **SynchronousQueue:** SynchronousQueue is special kind of BlockingQueue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa

Array blocking queue

- Attempts to put an element into a full queue will result in the operation blocking. Similarly attempts to take an element from an empty queue will also be blocked.

It follows FIFO

- ArrayBlockingQueue supports an optional fairness policy for ordering waiting producer and consumer threads. With fairness set to true, the queue grants threads access in FIFO order.
1. We will be implementing a producer consumer example with ArrayBlockingQueue
  2. We will be creating two Runnable's one for producer and other for consumer
  3. Producer will be putting numbers to the ArrayBlockingQueue using put()
  4. Consumer will be taking numbers from the ArrayBlockingQueue using take()
  5. Both put() and take() are blocking so if the queue is empty the thread will be blocked and similarly if the queue is full put() will be blocked

```

ArrayBlockingQueue<Integer> arrayBlockingQueue = new ArrayBlockingQueue<>( capacity: 5);

Runnable producer = () -> {
    int i=0;
    while (true){
        try {
            arrayBlockingQueue.put(++i);
            System.out.println("Added: " + i);
            TimeUnit.MILLISECONDS.sleep( timeout: 1000);
        }catch (InterruptedException ie){
            ie.printStackTrace();
        }
    }
}

Runnable consumer = () -> {
    while (true){
        try {
            Integer poll;
            poll = arrayBlockingQueue.take();
            System.out.println("Polled: " + poll);
            TimeUnit.MILLISECONDS.sleep( timeout: 1000);
        }catch (Exception ie){
            ie.printStackTrace();
        }
    }
}

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
executorService.submit(producer);
executorService.submit(consumer);
executorService.shutdown();

```

Both insertion and removal is going on parallel, After 5, to add more, atleast 1 element should be polled.

## Delay queue

# DelayQueue

- DelayQueue class is an unbounded blocking queue of delayed elements, in which an element can only be taken when its delay has expired.
- DelayQueue class is part of java.util.concurrent package.
- The head of the queue is that Delayed element whose delay expired furthest in the past.
- If no delay has expired there is no head and poll will return null.

Expiration occurs when an element's `getDelay(TimeUnit.NANOSECONDS)` method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using `take` or `poll`, they are otherwise treated as normal elements. For example, the `size` method returns the count of both expired and unexpired elements.

This queue does not permit null elements.

### What is delay element?

A delay element implements `java.util.concurrent.Delayed` interface and its `getDelay()` method return a zero or negative value which indicate that the delay has already elapsed

For clarity, we can consider that each element stores its activation date/time. As soon as, this timestamp reaches, element is ready to be picked up from queue.

## DelayQueue

### What is delay element?

A delay element implements `java.util.concurrent.Delayed` interface and its `getDelay()` method return a zero or negative value which indicate that the delay has already elapsed

For clarity, we can consider that each element stores its activation date/time. As soon as, this timestamp reaches, element is ready to be picked up from queue.

An implementation of `Delayed` interface must define a `compareTo()` method that provides an ordering consistent with its `getDelay()` method.

`compareTo(Delayed o)` method does not return the actual timestamp, generally. It returns a value less than zero if the object that is executing the method has a delay smaller than the object passed as a parameter – otherwise a positive value greater than zero. It will return zero if both the objects have the same delay.

1. We will be implementing a producer consumer example with Delay Queue
2. We will create a `DelayTask` class which implements `Delayed` interface
3. We will override `getDelay()` and `compareTo()`
4. We will be creating two `Runnable`s one for producer and other for consumer
5. Producer will be putting `DelayTask` objects to the `DelayQueue` using `put()` and with a Random expiration time
6. Consumer will be consuming `DelayTask` from the `DelayQueue` using `take()`

```
class DelayTask implements Delayed {  
    private String name;  
    private long delayTime;  
  
    public DelayTask (String name, long delayTime){  
        this.name=name;  
        this.delayTime= System.currentTimeMillis() + delayTime;  
    }  
}
```

```

@Override
public int compareTo(Delayed o) {
    if (this.delayTime < ((DelayTask)o).delayTime)
        return -1;
    if (this.delayTime > ((DelayTask)o).delayTime)
        return 1;
    return 0;
}

@Override
public long getDelay(TimeUnit unit) {
    long difference = delayTime - System.currentTimeMillis();
    return unit.convert(difference, TimeUnit.MILLISECONDS);
}

public class DelayQueueExample {
    static int taskCount;
    public static void main(String[] args) {
        DelayQueue<DelayTask> delayQueue = new DelayQueue<>();
    }
}

```

Here, we are producing messages infinitely. That's why while(true)

```

Runnable producer = () -> {
    while (true){
        long delayTime = new Random().nextInt( bound: 10000);
        //
        Date expirationTime = new Date(System.currentTimeMillis() + delayTime);
        String taskName = "Task " + taskCount++;
        delayQueue.put(new DelayTask(taskName, delayTime));
        System.out.println("Producing task: " + taskName + " with expiration time of: " + expirationTime);
        try {
            TimeUnit.MILLISECONDS.sleep( timeout: 5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

Runnable consumer = () -> {
    while (true){
        DelayTask poll;
        try {
            poll = delayQueue.take();
            System.out.println("Consumed task: " + poll.getName() + " with expiration of: " +
                new Date(poll.getDelayTime()));
            TimeUnit.MILLISECONDS.sleep( timeout: 2000);
        } catch (InterruptedException ie){
            ie.printStackTrace();
        }
    }
}

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
executorService.submit(producer);
executorService.submit(consumer);
executorService.shutdown();

```

## Linked Blocking Queue

- Linked blocking queue is based on linked nodes
- This queue orders elements FIFO (first-in-first-out)
- The head of the element is that element that has been on the queue the longest time
- The tail of the queue is that element that has been on the queue the shortest time
- New elements are inserted at the tail of the queue
- The queue retrieval operation is from the head of the queue



Code is same as ArrayBlocking queue. Just change the classname to LinkedBlockingQueue

## PriorityBlockingQueue

Java PriorityBlockingQueue class is a concurrent blocking queue data structure implementation in which objects are processed based on their priority. The “blocking” part of the name is added to imply the thread will block until there’s an item available on the queue.

In a priority blocking queue, the elements are ordered as per their natural ordering or based on a custom Comparator supplied at the time of creation.

### PriorityBlockingQueue Features:

1. A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The default initial capacity is '11' which can be overridden using initialCapacity parameter in appropriate constructor
  2. The objects of the priority queue are ordered by default in natural order
  3. It does not permit null elements
  4. It does not permit insertion of non-comparable objects
  5. PriorityBlockingQueue is thread safe
  6. PriorityBlockingQueue class and its iterator implements all of the optional methods of the Collection and Iterator interfaces
1. We will be implementing a producer consumer example with PriorityBlockingQueue
  2. We will be creating two Runnable's one for producer and other for consumer
  3. We will be creating an array of Strings (names) and producer will put() them in the queue
  4. Consumer will be taking names from the queue as per their natural ordering using take()
  5. Both put() and take() are blocking so if the queue is empty the thread will be blocked and similarly if the queue is full put() will be blocked

```
final String[] names = {"Mike", "Syed", "Jean", "Jenny", "Kajeev", "Henry"};  
final PriorityBlockingQueue<String> queue = new PriorityBlockingQueue<>();
```

```
Runnable producer = () -> {  
    for (String name:names)  
        queue.put(name);  
};
```

```
Runnable consumer = () -> {  
    while (true){  
        try {  
            System.out.println(queue.take());  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
};
```

```
ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);  
executorService.submit(producer);  
executorService.submit(consumer);  
executorService.shutdown();
```

By default, it is sorted in alphabetical order of names here.



# SynchronousQueue

- Synchronous queue is blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.
- A synchronous queue does not have any internal capacity, not even a capacity of one. You cannot peek at a synchronous queue because an element is only present when you try to remove it
- You cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate.
- The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null.

```
final String[] names = {"Mike", "Syed", "Jean", "Jenny", "Rajeev", "Henry"};
final SynchronousQueue<String> queue = new SynchronousQueue<>();

Runnable producer = () -> {
    for (String name: names){
        try {
            queue.put(name);
            System.out.println("Inserted: " + name + " in the queue");
            TimeUnit.MILLISECONDS.sleep( timeout: 1000);
        }catch (InterruptedException ie){
            ie.printStackTrace();
        }
    }
};

Runnable consumer = () -> {
    while (true){
        try {
            System.out.println("Retrieved: " + queue.take() + " from the queue");
            TimeUnit.MILLISECONDS.sleep( timeout: 2000);
        }catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
};
```

int nThreads

Same executor service for using above 2 runnables