```
Current Thread : main
First Thread
Current Thread : main
Third Thread
Current Thread : main
Second Thread
Exiting from Current Thread : main
```

In the code, if you don't write t2.join(), then current thread will not wait from the t2 thread to die, see the output below when t2.join() statement is commented from the code :

```
Current Thread : main
First Thread
Current Thread : main
Third Thread
Current Thread : main
Exiting from Current Thread : main
Second Thread
```

There are overloaded versions of join() method also,

- join(long milliseconds) : when this method is called, then the current thread will wait at most for the specified milliseconds
- join(long milliseconds, long nanoseconds) : when this method is called, then the current thread will wait at most for the specified milliseconds plus nanoseconds.

These join methods are dependent on the underlying Operating system for timing. So, you should not assume that join() will wait exactly as long as you specify.

You can execute threads in a sequence using CountDownLatch also.

Question 80: yield() method

Answer: yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.

When the yielded thread will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn't guarantee that the current thread will pause or stop but it guarantees that CPU will be relinquished by current Thread as a result of a call to Thread.yield() method in java.

Question 81: Tell something about synchronized keyword

Answer: synchronized keyword in java is used to control the access of multiple threads to any shared resource, so that any consistency problem can be avoided.

We can make the entire method as synchronized or just the part where the shared resource is getting used, to do this synchronized blocks are used.

Synchronized method/block can only have one thread executing inside it, all the other threads trying to enter into the synchronized method/block will get blocked until the thread inside

finishes its execution. When the thread exits the synchronized method/block then Java guarantees that changes to the state of the object is visible to all the threads. This eliminates the memory inconsistency errors.

Question 82: What is static synchronization?

Answer: When synchronized keyword is used with a static method, then that is called static synchronization. In this, lock will be on the class not the object. This means only one thread can access the class at a time.

The purpose of static synchronization is to make the static data thread-safe.

Let's look at some programs:

Here, we have a Hello class which has a synchronized method:

```
package com.multithreading.demo;

class Hello {

    synchronized void sayHello() {
        System.out.println("in sayHello() method " +
                                    Thread.currentThread().getName());
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName() + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

A Task class which implements Runnable and its run() method simply calls the synchronized method of Hello class:

```
class Task implements Runnable {

    Hello h;

    Task(Hello h) {
        this.h = h;
    }

    @Override
    public void run() {
        h.sayHello();
    }

}
```

Our main class:

```java
public class SynchronizationDemo {
    public static void main(String[] args) {
        Hello obj1 = new Hello();
        Hello obj2 = new Hello();

        Thread task1 = new Thread(new Task(obj1));
        task1.setName("First Thread");
        Thread task2 = new Thread(new Task(obj1));
        task2.setName("Second Thread");
        Thread task3 = new Thread(new Task(obj2));
        task3.setName("Third Thread");
        Thread task4 = new Thread(new Task(obj2));
        task4.setName("Fourth Thread");

        task1.start();
        task2.start();
        task3.start();
        task4.start();
    }
}
```

```
in sayHello() method First Thread
in sayHello() method Third Thread
First Thread , i = 1
Third Thread , i = 1
First Thread , i = 2
Third Thread , i = 2
First Thread , i = 3
Third Thread , i = 3
Third Thread , i = 4
First Thread , i = 4
First Thread , i = 5
Third Thread , i = 5
in sayHello() method Second Thread
```

We have 2 objects of our Hello class, one object is shared among First and Second thread, and one object is shared among Third and Fourth thread, and we are starting these threads.

Output:

```
in sayHello() method Second Thread
Second Thread , i = 1
in sayHello() method Fourth Thread
Fourth Thread , i = 1
Second Thread , i = 2
Fourth Thread , i = 2
Second Thread , i = 3
Fourth Thread , i = 3
Fourth Thread , i = 4
Second Thread , i = 4
Second Thread , i = 5
Fourth Thread , i = 5
```

As you can see from the output, the First and Second thread are not having any thread interference. Same way, Third and Fourth thread does not have any thread interference but First and Third thread are entering the synchronized method at the same time with their own object locks (Hello obj1 and obj2).

Lock which is hold by First thread will only stop the Second thread from entering the synchronized block, because they are working on the same instance i.e. obj1, but it cannot stop Third or Fourth thread as they are working on another instance i.e. obj2.

If we want our synchronized method to be accessed by only one thread at a time then we have to use a ***static synchronized method/block*** to have the synchronization on the class level rather than on the instance level.

```
class Hello {

    static synchronized void sayHello() {
        System.out.println("in sayHello() method " +
                                        Thread.currentThread().getName());
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName() + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Let's see the output now:

```
in sayHello() method First Thread
First Thread , i = 1
First Thread , i = 2
First Thread , i = 3
First Thread , i = 4
First Thread , i = 5
in sayHello() method Fourth Thread
Fourth Thread , i = 1
Fourth Thread , i = 2
Fourth Thread , i = 3
Fourth Thread , i = 4
Fourth Thread , i = 5
```

```
in sayHello() method Third Thread
Third Thread , i = 1
Third Thread , i = 2
Third Thread , i = 3
Third Thread , i = 4
Third Thread , i = 5
in sayHello() method Second Thread
Second Thread , i = 1
Second Thread , i = 2
Second Thread , i = 3
Second Thread , i = 4
Second Thread , i = 5
```

Here, only one thread is accessing the static synchronized method.

Same can be done by synchronized block also:

```
class Hello {

    static void sayHello() {
        synchronized(Hello.class) {
        System.out.println("in sayHello() method " +
                                    Thread.currentThread().getName());
            for(int i=1; i<=5; i++) {
                System.out.println(Thread.currentThread().getName() + " , i = " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Question 83: What will be output of below program where one synchronized method is calling another synchronized method?

```
package com.multithreading.demo;

class Demo {
    public synchronized void m1() {
        m2();
        System.out.println("inside m1()");
    }

    public synchronized void m2() {
        m3();
        System.out.println("inside m2()");
    }

    public synchronized void m3() {
        System.out.println("inside m3()");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Demo d1 = new Demo();

        d1.m1();
    }
}
```

Output:

```
inside m3()
inside m2()
inside m1()
```

One thing to remember here is that Java synchronized keyword is re-entrant in nature, it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding the lock can enter into that method without acquiring lock.

Question 84: Programs related to synchronized and static synchronized methods

There are some confusing programs that interviewer can ask where some methods are static synchronized and some methods are non-static synchronized, and sometimes they are calling each other, so let's discuss those.

Scenario 1: There are 2 threads that are calling 2 different static synchronized methods.

Here, these 2 threads will block each other, as only one lock per class exists. So, these 2 static synchronized methods will not be executed at the same time.

Program:

```
package com.multithreading.demo;

class Demo {
    public static synchronized void m1() {
        System.out.println("inside m1()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                                        + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    public static synchronized void m2() {
        System.out.println("inside m2()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                                        + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Demo d1 = new Demo();
        Demo d2 = new Demo();

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                d1.m1();
            }
        });
        t1.setName("First thread");

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                d2.m2();
            }
        });
        t2.setName("Second thread");

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

Output:

```
inside m1()
First thread , i = 1
First thread , i = 2
First thread , i = 3
First thread , i = 4
First thread , i = 5
inside m2()
Second thread , i = 1
Second thread , i = 2
Second thread , i = 3
Second thread , i = 4
Second thread , i = 5
```

Scenario 2: There are 2 threads, one is calling static synchronized method on one object, and the other thread is calling non-static synchronized method on another object.

Here, these 2 threads will not block each other and will be executed concurrently as both locks are different, the thread executing the static synchronized method holds a lock on the class and the thread executing the non-static synchronized method holds the lock on the object on which the method has been called.

In short, static synchronized method do not block a non-static synchronized method.

Program:

```
package com.multithreading.demo;

class Demo {
    public static synchronized void m1() {
        System.out.println("inside m1()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                    + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public synchronized void m2() {
        System.out.println("inside m2()");
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().getName()
                    + " , i = " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

(Main class is same as Scenario 1)

Output:

```
inside m2()
inside m1()
Second thread , i = 1
First thread , i = 1
Second thread , i = 2
First thread , i = 2
Second thread , i = 3
First thread , i = 3
First thread , i = 4
Second thread , i = 4
First thread , i = 5
Second thread , i = 5
```

Question 85: What is Callable Interface?

Answer: Callable interface represents an asynchronous task which can be executed by a separate thread, it has one call() method, which returns a Future object.

A Runnable can be executed by passing it into the Thread class constructor but Thread class does not have a constructor that accepts a Callable, so Callable can be executed only by submit() method of ExecutorService Interface.

Callable's call() method can throw checked exceptions, the exceptions are collected in Future object which can be checked by making a call to Future.get() method, an ExecutionException is thrown which wraps the original exception. We can get the original checked exception by making a call to getCause() method on the exception object thrown. However, if we don't call Future.get() method then the exception will not be reported back and the task will be marked as completed.

One thing you should remember is that the Future.get() method blocks the execution, so timeouts should be used when using this method to avoid unexpected waits.

Program showing how Callable is used and how the result is returned in Future object:

```java
class Task implements Callable<Integer> {
    private int num;

    public Task(int num) {
        this.num = num;
    }

    @Override
    public Integer call() throws Exception {
        if(num < 0) {
            throw new InvalidParameterException("Negative number not allowed");
        }
        return num*num;
    }
}
```

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

        Task task = new Task(5);

        ExecutorService es = Executors.newFixedThreadPool(2);
        Future<Integer> f = es.submit(task);

        try {
            System.out.println("Result: " + f.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        es.shutdownNow();
    }
}
```

Output:

```
Result: 25
```

Let's pass a negative number, so that exception will be thrown:

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

        Task task = new Task(-10);

        ExecutorService es = Executors.newFixedThreadPool(2);
        Future<Integer> f = es.submit(task);

        try {
            System.out.println("Result: " + f.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        es.shutdownNow();
    }
}
```

Output:

```
java.util.concurrent.ExecutionException: java.security.InvalidParameterException: Negative number not allowed
        at java.util.concurrent.FutureTask.report(Unknown Source)
        at java.util.concurrent.FutureTask.get(Unknown Source)
        at com.callable.demo.CallableDemo.main(CallableDemo.java:35)
Caused by: java.security.InvalidParameterException: Negative number not allowed
        at com.callable.demo.Task.call(CallableDemo.java:20)
        at com.callable.demo.Task.call(CallableDemo.java:1)
        at java.util.concurrent.FutureTask.run(Unknown Source)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
        at java.lang.Thread.run(Unknown Source)
```

You can call e.getCause() to get the original exception.

Question 86: How to convert a Runnable to Callable

Answer: We have a utility method in "Executors" class:

- callable(Runnable task) : Returns a Callable object that, when called, runs the given task and returns null.
- callable(Runnable task, T result) : Returns a Callable object that, when called, runs the given task and returns the given result

Question 87: Difference between Runnable and Callable

Answer: The difference is:

- Runnable tasks can be executed by using Thread class or ExecutorService interface whereas Callable tasks can be executed by using ExecutorService interface only
- Return type of Runnable's run() method is void whereas Callable's call() method returns Future object
- Runnable's run() method does not throw checked exceptions whereas Callable's call() method can throw checked exceptions

Question 88: What is Executor Framework in Java, its different types and how to create these executors?

Answer: Executor Framework is an abstraction to managing multiple threads by yourself. So, it decouples the execution of a task and the actual task itself. Now, we just have to focus on the task that means, only implement the Runnables and submit them to executor. Then these runnables will be managed by the executor framework. It is available from Java 1.5 onwards.

Also, we don't have to create new threads every time. With executor framework, we use Thread pools. Think of Thread Pool as a user-defined number of threads which are called worker threads, these are kept alive and reused. The tasks that are submitted to the executor will be executed by these worker threads. If there are more tasks than the threads in the pool, they can be added in a Queue and as soon as one of thread is finished with a task, it can pick the next one from this Queue or else, it will be added back in the pool waiting for a task to be assigned.

So, it saves the overhead of creating a new thread for each task. If you are thinking about what is the problem with creating a new thread every time we want to execute a task, then you should know that creating a thread is an expensive operation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead and new threads without any throttling will lead to the creation of large number of threads. These threads will cause wastage of resources.

There are 2 main interfaces that you must know, one is *Executor* and the other is *ExecutorService*.

Executor interface contains execute(Runnable task) method through which you can execute only Runnables. Also, the return type of execute() method is void, since you are passing a Runnable to it and it does not return any result back.

ExecutorService interface contains the submit() method which can take both Runnable and Callable, and its return type is Future object. ExecutorService extends the Executor Interface, so it also has the execute() method.

Now, we have an idea of what is an Executor Framework, let's look at different types of Executors:

SingleThreadExecutor:

This executor has only one thread and is used to execute tasks in a sequential manner. If the thread dies due to an exception while executing the task, a new thread is created to replace the old thread and the subsequent tasks are executed in the new thread.

How to create a SingleThreadExecutor:

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

Executors is a utility class which contains many factory methods to create different types of ExecutorService, like the one called SingleThreadExecutor, we just created.

FixedThreadPoolExecutor:

As its name suggests, this is an executor with a fixed number of threads. The tasks submitted to this executor are executed by the specified number of threads and if there are more tasks than the number of threads, then those tasks will be added in a queue (e.g. LinkedBlockingQueue).

How to create a FixedThreadPoolExecutor:

ExecutorService executor = Executors.*newFixedThreadPool*(5);

Here, we have created a thread pool executor of 5 threads, that means at any given time, 5 tasks can be managed by this executor. If there are more active tasks, they will be added to a queue until one of the 5 threads becomes free.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

CachedThreadPoolExecutor:

This executor is mainly used when there are many short-lived tasks to be executed. If you compare this with the fixed thread pool, here the number of threads of this executor pool is not bounded. If all the threads are busy executing the assigned tasks and if there is a new task, then a new thread will be created and added to the pool. If a thread remains idle for close to sixty seconds, it is terminated and removed from the cache.

Use this one, if you are sure that the tasks will be short-lived, otherwise there will be a lot of threads in the pool which will lead to performance issues.

How to create a CachedThreadPoolExecutor:

ExecutorService executor = Executors.*newCachedThreadPool*();

ScheduledExecutor:

Use this executor, when you want to schedule your tasks, like run them at regular intervals or run them after a given delay. There are 2 methods which are used for scheduling tasks: *scheduleAtFixedRate* and *scheduleWithFixedDelay*.

How to create ScheduledExecutor:

ExecutorService executor = Executors.*newScheduledThreadPool*(4);

ScheduledExecutorService interface extends the ExecutorService interface.

Now, apart from using Executors class to create executors, you can use ThreadPoolExecutor and ScheduledThreadPoolExecutor class also. Using these classes, you can manually configure and fine-tune various parameters of the executor according to your need. Let's see at some of those parameters:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

Core and Max Pool sizes:

A ThreadPoolExecutor will automatically adjust the pool size according to the bounds set by corePoolSize and maximumPoolSize

When a new task is submitted to the executor then:

- If the number of threads running are less than the corePoolSize, a new thread is created to handle the request
- If the number of threads running are more than corePoolSize but less than maximumPoolSize then a new thread will be created only if the queue is full

Let's understand this with an example:

You have defined the core pool size as 5, maximum pool size as 10 and the queue capacity as 100. Now as tasks are coming in, new threads will be created up to 5, then other new tasks will be added to queue until it reaches 100. Now when the queue is full and if new tasks are coming in, threads will be created up to the maximumPoolSize i.e. 10. Once all the threads are in use and the queue is also full, the new tasks will be rejected. As the queue reduces, so does the number of active threads.

Keep Alive Time and TimeUnit:

When the number of threads are greater than the core size, this is the maximum time that excess idle threads will wait for new tasks before terminating. It is used to avoid the overhead of creating a new thread.

Let's understand this with an example:

You have defined the core pool size as 5 and maximum pool size as 15 and all the 15 threads are getting used at the moment. Now when the threads are getting finished with their work, the excess 10 threads (15-5) become idle and eventually die. To avoid these 10 threads being killed too quickly, we can specify the keep alive time for these by using the keepAliveTime parameter in the ThreadPoolExecutor constructor. If you have given its value as 1 and time unit as TimeUnit.MINUTE, each thread will wait for 1 min after it had finished executing a task. Basically, it is waiting for a new task to be assigned. If it is not given any task, it would let itself complete. And in the end, the executor will be left with the core threads (5).

BlockingQueue:

The queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method, you can use a ArrayBlockingQueue or LinkedBlockingQueue like:

```
BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);
```

ThreadFactory:

The factory to use when the executor creates a new thread. Using thread factories removes hardwiring of calls to *new Thread*, enabling applications to use special thread subclasses, priorities, etc.

RejectedExecutionHandler:

This handler is used when a task is rejected by the executor because all the threads are busy and the queue is full.

When this handler is not provided and the task submitted to execute() method is rejected, then an unchecked *RejectedExecutionException* is thrown.

But adding a handler is a good practice to follow, there is a method:

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
```

This method will be invoked by ThreadPoolExecutor when execute() cannot accept a task.

Putting it all together:

```
BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);

ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 15, 50, TimeUnit.SECONDS, queue);

executor.setRejectedExecutionHandler(new RejectedExecutionHandler() {

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        //run this code when task is rejected
    }
});
```

Question 89: Tell something about awaitTermination() method in executor

Answer: This method blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

```
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
```

It returns true if this executor is terminated and false if the timeout is elapsed before termination.

Question 90: Difference between shutdown() and shutdownNow() methods of executor

Answer: An executor will not shut down automatically even when there is no task to process. It will stay alive and wait for new work. It will keep the JVM running.

When shutdown() method is called on an executor, then the executor will not accept new tasks and it will wait for the currently executing tasks to finish.

When shutdownNow() is called, it tries to interrupt the running threads and shutdown the executor immediately. However, there is no guarantee that all the running threads will be stopped at the same time.

One good way to shutdown an executor is to use both of these methods along with awaitTermination(). With this approach, the executor will stop accepting new tasks and waits up to the specified duration for all running tasks to be completed. If the time expires, it will shutdown immediately.

```
executor.shutdown();

try {
    if(executor.awaitTermination(5, TimeUnit.MINUTES)) {
        executor.shutdownNow();
    }
} catch (InterruptedException e) {
    executor.shutdownNow();
}
```

Question 91: What is Count down latch in Java?

Answer: CountDownLatch is used in requirements where you want one or more tasks to wait for some other tasks to finish before it starts its own execution. One example can be a server which is dependent on some services to be up and running before it can start processing requests.

How to use: When we create an object of CountDownLatch, then we specify the number of threads that it should wait for, then waiting thread calls the countDownlatch.await() method and until all the specified thread calls countDownLatch.countDown() method, the waiting thread will not start its execution.

For example, there are 3 services which a server is dependent on, before the server accepts any request, these services should be up and running. We will create a CountDownLatch by specifying 3 threads that the main thread should wait for, then main thread will call await() method means it will wait for all 3 threads. Once the threads are complete they will call countdown() method, decreasing the count by 1. The main thread will start its execution only when count reaches to zero.

```java
package com.countdownlatch.demo;

import java.util.concurrent.CountDownLatch;

class Task implements Runnable {
    String service;
    CountDownLatch latch;

    public Task(String service, CountDownLatch latch) {
        this.service = service;
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(service + " is up");
        latch.countDown();
    }
}
```

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

        CountDownLatch latch = new CountDownLatch(3);

        Thread t1 = new Thread(new Task("Service1", latch));
        Thread t2 = new Thread(new Task("Service2", latch));
        Thread t3 = new Thread(new Task("Service3", latch));

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

        try {
            latch.await();
            System.out.println("All services are up, "
                    + "Starting Main Application now");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Service1 is up
Service2 is up
Service3 is up
All services are up, Starting Main Application now
```

CountDownLatch can also be used to start multiple threads at the same time, you can create a CountDownLatch of size 1, make all the other threads wait by calling countDownLatch.await(),

then a single call to countDownLatch.countDown() method will resume execution for all the waiting threads at the same time.

CountDownLatch cannot be reused once the count reaches to zero, therefore in those scenarios, CyclicBarrier is used.

Question 92: What is Cyclic Barrier?

Answer: CyclicBarrier is used to make multiple threads wait for each other to reach a common barrier point. It is used mainly when multiple threads perform some calculation and the result of these threads needs to be combined to form the final output.

All the threads that wait for each other to reach the barrier are called parties, CyclicBarrier is created with a number of parties to wait for, and the threads wait for each other at the barrier by calling cyclicBarrier.await() method which is a blocking method and it blocks until all threads have called await().

The barrier is called Cyclic because it can be re-used after the waiting threads are released, by calling cyclicBarrier.reset() method which resets barrier to the initial state.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

```
public CyclicBarrier(int parties, Runnable barrierAction)
```

The CyclicBarrier uses an all-or-none breakage model for failed synchronization attempts: If a thread leaves a barrier point prematurely because of interruption, failure, or timeout, all other threads waiting at that barrier point will also leave abnormally via BrokenBarrierException (or InterruptedException if they too were interrupted at about the same time).

Example: One thread is adding first 5 natural numbers to the list, the other thread is adding next 5 numbers to the list and we will perform an addition of all these numbers to compute the sum of first 10 natural numbers

```
package com.cyclicbarrier.demo;

import java.util.List;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CyclicBarrier;

class Task1 implements Runnable {
    String name;
    List<Integer> numbers;
    CyclicBarrier barrier;

    public Task1(String name, List<Integer> numbers, CyclicBarrier barrier) {
        this.name = name;
        this.numbers = numbers;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        System.out.println(name + " is running");

        for(int i=1; i<=5; i++) {
            numbers.add(i);
        }

        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(name + " has crossed the barrier");
    }
}
```

```
class Task2 implements Runnable {
    String name;
    List<Integer> numbers;
    CyclicBarrier barrier;

    public Task2(String name, List<Integer> numbers, CyclicBarrier barrier) {
        this.name = name;
        this.numbers = numbers;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        System.out.println(name + " is running");

        for(int i=6; i<=10; i++) {
            numbers.add(i);
        }

        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(name + " has crossed the barrier");
    }
}
```

```
class FinalTask implements Runnable {
    String name;
    List<Integer> numbers;

    public FinalTask(String name, List<Integer> numbers) {
        this.name = name;
        this.numbers = numbers;
    }
    @Override
    public void run() {
        int sum = 0;
        for(Integer i : numbers) {
            sum = sum + i;
        }
        System.out.println("Sum of first 10 natural numbers : " + sum);
    }
}

public class CyclicBarrierDemo {
    public static void main(String[] args) {
        List<Integer> numbers = new CopyOnWriteArrayList<Integer>();

        CyclicBarrier barrier
                = new CyclicBarrier(2, new FinalTask("Final Thread", numbers));

        Thread t1 = new Thread(new Task1("First Thread", numbers, barrier));
        Thread t2 = new Thread(new Task2("Second Thread", numbers, barrier));

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

Output:

```
First Thread is running
Second Thread is running
Sum of first 10 natural numbers : 55
Second Thread has crossed the barrier
First Thread has crossed the barrier
```

Question 93: Atomic classes

Answer: The java.util.concurrent.atomic package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set call has a happens-before relationship with any subsequent get call on the same variable.

For example, consider below code:

```
class Task implements Runnable {
    private int count;

    public int getCount() {
        return this.count;
    }

    @Override
    public void run() {
        for(int i=1; i<=50; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count++;
        }
    }
}
```

In the above runnable task, we are just incrementing an integer 50 times,

```
public class AtomicDemo {
    public static void main(String[] args) {
        Task task = new Task();

        Thread t1 = new Thread(task);
        //Thread t2 = new Thread(task);

        t1.start();
        //t2.start();

        try {
            t1.join();
            //t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Count is : " + task.getCount());
    }
}
```

We are using join() method so that the main thread will wait for the thread t1 to die, otherwise the sysout will be executed before t1 has finished execution. Let's see what will be the output when only one thread is running:

Output:

```
Count is : 50
```

As, you can see, the output is as expected because only one thread is accessing the count, let's see what will happen in case the count variable is accessed by more than one thread, un-comment the code regarding second thread t2 and run the main class:

Output:

```
Count is : 83
```

The expected output was 100 but we got a different output, if you run the above program you will see a different output and it will be anywhere between 50 and 100. The reason for this is that 2 threads are accessing a mutable variable without any synchronization. One solution that will be coming to your mind will be using synchronization block, and yes this problem can be solved using that but it will have a performance impact, as threads will acquire the lock, update the value and release the lock, and then giving other threads access to the shared mutable variable.

But java has provided Atomic wrapper classes for this purpose that can be used to achieve this atomic operation without using Synchronization.

Let's see the change in our Runnable:

```
class Task implements Runnable {
    private AtomicInteger count = new AtomicInteger();

    public int getCount() {
        return this.count.get();
    }

    @Override
    public void run() {
        for(int i=1; i<=50; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count.incrementAndGet();
        }
    }
}
```