Table of Contents

JAVA CONCURRENCY AND JAVA PROFILING 2
Executors<<class>> 2
Callable<<interface>> 4
Future<<interface>> 4
ThreadPoolExecutor<<class>> 5
Executor<<interface>> 8
ExecutorService<<interface>> 8
BlockingQueue<<interface>> 9
Producer Consumer Pattern 9
Using BlockingQueue 9
ConcurrentHashMap<<class>> 11
CountDownLatch<<class>> 13
CyclicBarrier <<class>> 14
Deadlock Condition and Solution 16
Race Condition 18
Semaphore 21
ForkJoin 23
INTERVIEW QUESTIONS 25

JAVA CONCURRENCY AND JAVA PROFILING

Working with the Thread class can be very tedious and error-prone. Due to this reason the Concurrency API has been added to Java 5. The API is located in the package **java.util.concurrent** and contains many useful classes for implementing **thread pooling** and handling **concurrent programming**.

Creating a new thread–per-every task is expensive because of memory overload i.e., poor resource management. Hence it is always recommended to use thread pooling.

The threads in the pool are reused but not recreated.

We can implement thread pooling effectively using either **Executors** or **ThreadPoolExecutor** classes in java.util.concurrent package. The difference between Executors and ThreadPoolExecutor is default vs customized configurations.

Note: Tasks are logical units of work, and threads are a mechanism by which tasks can run in parallel.

Executors<<class>>

This class provides convenient factory methods to implement thread pooling along with some **predefined configurations**.

The Executors are capable of running parallel tasks and typically manage a **pool of threads**, so we don't have to create new threads manually.

Executors has to be stopped explicitly using either shutdown() or shutdownNow() methods from ExecutorService<<interface>>, otherwise they keep listening for new tasks.

The shutdown() waits for currently running tasks to finish while shutdownNow() interrupts all running tasks and shut the executor down immediately.

The Executors class supports both java.lang.Runnable and java.util.concurrent.Callable interfaces. The Callable interface is same as Runnable but instead of being void they return a value.

The runnable tasks are submitted to the executor service using either execute() or submit() methods.

The callable tasks are always submitted to the executor service using submit() method.

This class contains following methods:

1. **public static ExecutorService newFixedThreadPool(int poolsize)**

   This method creates fixed-size thread pool as tasks are submitted, up to the poolSize, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected Exception). If additional tasks are submitted when all threads are busy, they will wait in the queue until a thread is available.

   Example #1:

```
public class RunnableOne implements Runnable{
        @Override
        public void run(){
                //business logic
                try {
                        Thread.sleep(20*1000);
                } catch (InterruptedException e) {  e.printStackTrace(); }
                System.out.println("*****");
        }
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ExecutorsDemo1 {
        public static void main(String[] args) {
                RunnableOne r = new RunnableOne(); //runnable obj
                ExecutorService threadPool = Executors.newFixedThreadPool(5);//put break point here
                threadPool.execute(r);
                threadPool.execute(r);

                threadPool.execute(r);
                threadPool.execute(r);
                threadPool.execute(r);
                threadPool.execute(r);
                threadPool.execute(r);
                threadPool.execute(r);

                threadPool.shutdown();
        }
}
```

2. **public static ExecutorService newCachedThreadPool()**
   A cached thread pool has more flexibility to remove idle threads (default time is sixty seconds) when the current pool size exceeds the demand for processing, and to add new threads when demand increases, **but places no bounds on the pool size**. The threads that have not been used for **sixty seconds** are terminated and removed from the pool.
   Example #2:

```
public class RunnableOne implements Runnable{
        @Override
        public void run(){
                //business logic
                try {
                        Thread.sleep(20*1000);
                } catch (InterruptedException e) { e.printStackTrace(); }
                System.out.println("*****");
        }
}

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ExecutorsDemo2 {
        public static void main(String[] args) {
                RunnableOne r = new RunnableOne(); //runnable obj
                ExecutorService threadPool = Executors.newCachedThreadPool();

                threadPool.execute(r);
                threadPool.execute(r);
                threadPool.execute(r);
```

```
                    threadPool.execute(r);
                    threadPool.execute(r);
                    threadPool.execute(r);
                    threadPool.execute(r);
                    threadPool.execute(r);
                    //threadPool.shutdown(); //not required because threads are automatically
                    removed from pool if they are idle for 60 seconds which is default value.
          }
     }
```

Callable<<interface>>

The Executors class also supports java.util.concurrent.Callable interfaces.
The Callable interface is same as Runnable but instead of being void they return a value.
This interface contains call() method:

public Object call() throws Exception

The call() returns result and throws checked exception.

Example:

```
import java.util.concurrent.Callable;
public class CallableOne implements Callable<Integer>{
     @Override
     public Integer call() throws Exception{
          //business logic
          try {
               Thread.sleep(20*1000);
          } catch (InterruptedException e) {  e.printStackTrace(); }
          System.out.println("*****");
          return 1234;
     }
}
```

Note: The callable tasks are always submitted to the executor service using **submit**() method.

Future<<interface>>

In case of callable task, the result is returned but the submit() doesn't wait until the task completes. The executor service cannot return the result of the callable directly. Instead the executor returns a special result of type **Future** which can be used to retrieve the actual result at a later point in time.

public Future<T> submit(Callable<T> task)

Example:

```
     Future<Integer> future = exeuctorService.submit(callableTask);
     System.out.println("future done? " + future.isDone()); //may be false
     Integer result = future.get(); //blocks the current thread and waits until the callable task
     completes before returning the actual result
     System.out.println("future done? " + future.isDone()); //always true
     System.out.print("result: " + result);
```

The task has four states: created, submitted, started and completed.

The Future represents the lifecycle of a task and provides methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task.
This interface contains following methods:

a) public boolean cancel(boolean)
b) public Boolean isCancelled()
c) public Boolean isDone()
d) public Object **get()** - Retrieves task result.

Example #3:

```
import java.util.concurrent.Callable;
public class CallableOne implements Callable<Integer>{
        @Override
        public Integer call()throws Exception{
                //business logic
                try {
                        Thread.sleep(20*1000);
                } catch (InterruptedException e) {  e.printStackTrace(); }
                System.out.println("*****");
                return 1234;
        }
}

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class ExecutorsDemo3 {
        public static void main(String[] args) throws Exception{
                CallableOne cobj = new CallableOne(); //callable obj
                ExecutorService threadPool = Executors.newCachedThreadPool(); //put break point here

                Future<Integer> f1 = threadPool.submit(cobj); //submits the task to executor service.
                Future<Integer> f2 = threadPool.submit(cobj); //submits the task to executor service.

                System.out.println("future done? " + f1.isDone()); //may be false
                System.out.println("future done? " + f2.isDone()); //may be false

                System.out.println(f1.get());
                System.out.println(f2.get());

                System.out.println("future done? " + f1.isDone()); //always true
                System.out.println("future done? " + f2.isDone()); //always true
        }
}
```

ThreadPoolExecutor<<class>>

This class provides **customized** thread pool implementation. We can specify the core pool size, maximum pool size, idle time, queue size, etc.
If additional tasks are submitted when all threads are busy and queue is full then executor service throws **RejectedExecutionHandler.**
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)
corePoolSize – The number of threads in pool created as tasks are submitted, and these threads remain in pool even though they are idle.
maximumPoolSize - The maximum number of threads created as tasks are submitted if and only if the queue is full and all core threads are busy.
keepAliveTime –When the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.
unit - The time unit for the keepAliveTime argument
workQueue - The queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute method.
Example #1:

```
public class RunnableOne implements Runnable{
	@Override
	public void run() {
		// business logic
		try {
			Thread.sleep(20 * 1000);
		} catch (InterruptedException e) { e.printStackTrace(); }
		System.out.println("runnable");
	}
}

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.RejectedExecutionException;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
public class ThreadPoolExecutorDemo1 {
	public static void main(String[] args) {
		BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(3);
		ExecutorService threadPool = new ThreadPoolExecutor(2, 4, 30, TimeUnit.SECONDS, queue);
		RunnableOne r = new RunnableOne();
		try {
			threadPool.execute(r);  //pool-1-thread-1
			threadPool.execute(r); ////pool-1-thread-2
```

```
                    threadPool.execute(r); //goes to queue
                    threadPool.execute(r); //goes to queue
                    threadPool.execute(r); //goes to queue

                    threadPool.execute(r); //pool-1-thread-3
                    threadPool.execute(r); //pool-1-thread-4

                    threadPool.execute(r); //causes RejectedExcecutionException
                    threadPool.execute(r);
                    threadPool.execute(r);
            } catch (RejectedExecutionException e) {
                    e.printStackTrace();
            } finally {
                    threadPool.shutdown();
            }
        }
}
```

We can provide implementation for **RejectedExecutionHandler** to handle the tasks that can't fit in the worker queue.

Example #2:

```
public class RunnableOne implements Runnable{
        @Override
        public void run() {
                // business logic
                try {
                        Thread.sleep(20 * 1000);
                } catch (InterruptedException e) { e.printStackTrace(); }
                System.out.println("runnable");
        }
}

import java.util.concurrent.*;
class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {
         @Override
         public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
            System.out.println(r.toString() + " is rejected");
         }
 }

public class ThreadPoolExecutorDemo2 {
        public static void main(String[] args) {
```

```
		BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(3);
		//RejectedExecutionHandler implementation
		RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();
		ExecutorService threadPool = new ThreadPoolExecutor(2, 4, 30, TimeUnit.SECONDS, queue, rejectionHandler);

		RunnableOne r = new RunnableOne();
		try {
			threadPool.execute(r);  //pool-1-thread-1
			threadPool.execute(r); ////pool-1-thread-2

			threadPool.execute(r); //goes to queue
			threadPool.execute(r); //goes to queue
			threadPool.execute(r); //goes to queue

			threadPool.execute(r); //pool-1-thread-3
			threadPool.execute(r); //pool-1-thread-4

			threadPool.execute(r); //handled by our rejection handler implementation
			threadPool.execute(r); //handled by our rejection handler implementation
			threadPool.execute(r); //handled by our rejection handler implementation
		} catch (RejectedExecutionException e) {
			e.printStackTrace();
		} finally {
			threadPool.shutdown();
		}
	}
}
```

Executor<<interface>>

It is used to execute Runnable tasks asynchronously means simultaneously.
This interface contains *execute(Runnable)* method.

ExecutorService<<interface>>

It is a sub interface of Executor<<interface>>.
The Concurrency API introduces the concept of an ExecutorService as a higher level replacement for working with threads directly.
An Executor implementation is likely to create threads for processing tasks. But the JVM can’t exit until all the (non-daemon) threads have terminated, so failing to shut down an Executor could prevent the JVM from exiting.
To address the above issue, the ExecutorService interface extends Executor, adding a number of methods for lifecycle management (as well as some convenience methods for task submission).

The lifecycle management methods of ExecutorService are:

1) ***Public void shutdown()***

   This method will wait and remove all threads from the pool if and only if threads are idle and queue is empty.

   This method initiates a **graceful shutdown** i.e., no new tasks are accepted but previously submitted tasks are allowed to complete including those that have not yet begun execution.

2) ***Public List<Runnable> shutdownNow()***

   Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

   This method initiates an **abrupt shutdown** i.e., it attempt to cancel outstanding tasks and does not start any tasks that are queued but not begun.

The lifecycle implied by ExecutorService has three states—running, shutting down and terminated.
The ExecutorServices are initially created in the running state.
The shutdown() initiates a graceful shutdown: no new tasks are accepted but previously submitted tasks are allowed to complete—including those that have not yet begun execution.
The shutdownNow method initiates an abrupt shutdown: it attempts to cancel outstanding tasks and does not start any tasks that are queued but not begun.

BlockingQueue<<interface>>

Queued pending tasks.
The subclasses are:

Java.util.concurrent.ArrayBlockingQueue<<class>>
Java.util.concurrent.LinkedBlockingQueue<<class>>
Java.util.concurrent.PriorityBlockingQueue<<class>>

Producer Consumer Pattern

Producer should wait if queue is full and Consumer should wait if queue is empty.
This problem can be implemented or solved by different ways in Java:

1. Using wait() and notify() methods
2. BlockingQueue in java.util.concurrent package [Simple hence Preferred]

Using BlockingQueue

It is much simpler to implement Producer Consumer problem using java.util.concurrent.BlockingQueue because it provides control implicitly by introducing blocking methods **put**() and **take**().
We don't require to use wait and notify to communicate between Producer and Consumer.
The BlockingQueue's put() method will automatically block if queue is full in case of bounded queue and take() will automatically block if queue is empty.
Example:
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ProducerConsumerPattern {
        public static void main(String args[]) {
                // Creating shared object
                BlockingQueue<Runnable> sharedQueue = new ArrayBlockingQueue<Runnable>(5);
                ExecutorService threadPool = Executors.newFixedThreadPool(2);

                // Executing Producer and Consumer tasks
                threadPool.execute(new Producer(sharedQueue));
                threadPool.execute(new Consumer(sharedQueue));
                threadPool.shutdown();
        }
}

// Producer Class in java
class Producer implements Runnable {
        private final BlockingQueue sharedQueue;
        public Producer(BlockingQueue sharedQueue) {  this.sharedQueue = sharedQueue; }

        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
                        try {
                                System.out.println("Produced: " + i);
                                sharedQueue.put(i);
                        } catch (InterruptedException ex) {
                                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
                        }
                }
        }
}

// Consumer Class in Java
class Consumer implements Runnable {
        private final BlockingQueue sharedQueue;
        public Consumer(BlockingQueue sharedQueue) {  this.sharedQueue = sharedQueue; }

        @Override
        public void run() {
```

```
                    while (true) {
                              try {
                                        System.out.println("Consumed: " + sharedQueue.take());
                              } catch (InterruptedException ex) {
                                        Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);
                              }
                    }
          }
}
```

ConcurrentHashMap<<class>>

The ConcurrentHashMap is very similar to HashMap but it provides better concurrency level.

We can synchronize HashMap using Collections.synchronizedMap(Map).

So what is the difference between ConcurrentHashMap and Collections.synchronizedMap(Map)?

In case of Collections.synchronizedMap(Map), it locks whole HashMap but in case of ConcurrentHashMap, it locks only part of it.

In case of synchronized version of HashMap, only one thread is allowed to read or modify.

But in case of ConcurrentHashMap, by default, **sixteen** threads are allowed at a time to read or modify.

ConcurrentHashMap allows concurrent modifications from several threads without blocking everything hence enhances performance. To better visualize the ConcurrentHashMap, let it consider as a group of hash maps.

In ConcurrentHashMap, the difference lies in internal structure to store these key-value pairs. ConcurrentHashMap has an additional concept of segments. There is an Segment[] array called segments which has default size of 16.

The ConcurrentHashMap class has inner class called Segment. We can assume that each segment is equivalent to one HashMap.

A ConcurrentHashMap is divided into number of segments [default 16] on initialization.

The ConcurrentHashMap allows similar number (default is 16) of threads to access these segments concurrently so that each thread works on a specific segment during high concurrency. This way, if when our key-value pair is stored in segment 10; code does not need to block other 15 segments additionally. This structure provides a very high level of concurrency.

In other words, the ConcurrentHashMap uses a multitude of locks, each lock controls one segment of the map. When setting data in a particular segment, the lock for that segment is obtained.

Example #1: Using synchronized version of HashMap

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class SynchronizedHashMapDemo {
          public static void main(String[] args) {
                    Map<String, String> map = new HashMap<String, String>(); //put break point here
```

```
		Collections.synchronizedMap(map); //becomes synchronized hashmap

		map.put("India", "Delhi");
		map.put("Japan", "Tokyo");
		map.put("France", "Paris");
		map.put("Russia", "Moscow");
		map.put("Pakistan","Islamabad");
		map.put("Srilanka","Colombo");

		Iterator<String> ittr = map.keySet().iterator();
		while (ittr.hasNext()) {
			String countryObj = ittr.next();
			//map.put("abc", "xyz"); //throws ConcurrentModificationException
			String capital = map.get(countryObj);
			System.out.println(countryObj + "----" + capital);
		}
	}
}
```

Example #2: Using ConcurrentHashMap

```
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
//use jdk1.7
public class ConcurrentHashMapDemo {
	public static void main(String[] args) {
		Map<String, String> map = new ConcurrentHashMap<String, String>(); //put break point here

		map.put("India", "Delhi");
		map.put("Japan", "Tokyo");
		map.put("France", "Paris");
		map.put("Russia", "Moscow");
		map.put("Pakistan","Islamabad");
		map.put("Srilanka","Colombo");

		Iterator<String> ittr = map.keySet().iterator(); //put break point here
		while (ittr.hasNext()) {
			String countryObj = ittr.next();
			map.put("abc", "xyz"); //Ok.
			String capital = map.get(countryObj);
			System.out.println(countryObj + "----" + capital);
		}
```

```
        }
}
```

CountDownLatch<<class>>

This is used to disable (means wait) main processing thread for thread scheduling until other thread(s) complete their processing.

Note: We can also implement same functionality using wait() and notify() methods but it requires lot of code. But With CountDownLatch it can be done in just few lines.

The CountDownLatch works in latch principle i.e., the main processing thread will wait at gate/door until latch is open.

Main processing thread waits for 'n' number of dependent threads specified while creating CountDownLatch in Java. Any thread (i.e., main processing thread) which calls **CountDownLatch.await()** will disable means wait until count reaches zero or its interrupted by another thread.

All other threads are required to do **count down** by calling **CountDownLatch.countDown()** once they are completed or ready to the job.

As soon as count reaches zero, then latch is opened so that main processing thread starts running.

Example:

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.*;
import java.util.logging.*;
/**
 * Java program to demonstrate how to use CountDownLatch in Java. CountDownLatch is useful if we
 * want to start main processing thread once its dependency is completed as illustrated in this
 * CountDownLatch Example.
 */
public class CountDownLatchDemo {
        /**
         * Application should not start processing any thread until all services are up and ready to do
         *their job. The main thread will start with count 3 and wait until count reaches zero. Each
         * thread once up and read will do a count down. This will ensure that main thread is not started
         * processing until all services are up. Count is 3 since we have 3 threads means services.
         */
        public static void main(String args[]) {
                ExecutorService threadPool = Executors.newFixedThreadPool(3);
                final CountDownLatch latch = new CountDownLatch(3);
                threadPool.execute(new Service("CacheService", latch));
                threadPool.execute(new Service("AlertService", latch));
                threadPool.execute(new Service("ValidationService", latch));

                try {
                        latch.await(); // main thread is waiting on CountDownLatch to finish
                        System.out.println("All services are up, Application is starting now");
```

```
                    } catch (InterruptedException ie) {  ie.printStackTrace();
                    }finally{  threadPool.shutdown(); }
          }
}
/**
 * Service class which will be executed by Thread using CountDownLatch synchronizer.
 */
class Service implements Runnable {
          private final String name;
          private final CountDownLatch latch;
          public Service(String name, CountDownLatch latch) {
                    this.name = name;
                    this.latch = latch;
          }
          @Override
          public void run() {
                    try {
                              Thread.sleep(1000);
                    } catch (InterruptedException ex) {
                              Logger.getLogger(Service.class.getName()).log(Level.SEVERE, null, ex);
                    }
                    System.out.println(name + " is Up");
                    latch.countDown(); // reduce count of CountDownLatch by 1
          }
}
```

Note: The CountDownLatch is not reusable once count reaches to zero i.e., we cannot reuse CountDownLatch anymore.

CyclicBarrier <<class>>

It allows multiple threads to wait for each other at barrier before proceeding.
All threads should call CyclicBarrier.await() to wait for each other to reach barrier.
In general calling await() is shout out that thread is waiting on barrier.
CyclicBarrier is initialized with number of threads to be waited for each other.

Example :

```
import java.util.concurrent.*;
import java.util.logging.*;
/**
 * Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a new Concurrency
 * Utility added in Java 5 Concurrent package.
 */
public class CyclicBarrierExample {
    // Runnable task for each thread
    private static class Task implements Runnable { //static inner class
            private CyclicBarrier barrier;
            public Task(CyclicBarrier barrier) { this.barrier = barrier; }
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName() + " is waiting on barrier");
                    barrier.await();
                    System.out.println(Thread.currentThread().getName() + " has crossed the barrier");
                } catch (InterruptedException ex) {
                    Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);
            } catch (BrokenBarrierException ex) {
                    Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);
            }}
    }

  public static void main(String args[]) {
    //creating CyclicBarrier with 3 tasks i.e. 3 Threads needs to call await()
    final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){
      @Override
      public void run(){
        //This task will be executed once all thread reaches barrier
        System.out.println("All parties are arrived at barrier");
      }
    });

    ExecutorService threadPool = Executors.newFixedThreadPool(3);
    threadPool.execute(new Task(cb));
    threadPool.execute(new Task(cb));
    threadPool.execute(new Task(cb));

    cb.reset(); // resets barrier to its initial state to reuse

    threadPool.execute(new Task(cb));
    threadPool.execute(new Task(cb));
    threadPool.execute(new Task(cb));

    threadPool.shutdown();
  }
}
```

We can reuse CyclicBarrier by calling **reset**() method which resets barrier to its initial state.
The CountDownLatch is good for one time event like application start-up time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently calculating solution of big problem etc.

The CyclicBarrier is used in following scenarios:

1. To implement multi player game which cannot begin until all players has joined.
2. Concurrently calculating solution of big problem such as Map reduce technique.
3. It is used to implement fork-join, where a big task is broken down into smaller pieces and to complete the task we need output from individual small tasks.

Deadlock Condition and Solution

Deadlock occurred if two or more threads are blocked forever because of waiting for each other to get resources to proceed.

Example #1:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class DeadLockCondition {
	public static Object lock1 = new Object();
	public static Object lock2 = new Object();
	public static void main(String args[]) {
		ExecutorService threadPool = Executors.newFixedThreadPool(2);
		threadPool.execute(new ThreadDemo1());
		threadPool.execute(new ThreadDemo2()); //view JVM in java profiler
		threadPool.shutdown(); //no use
	}

	private static class ThreadDemo1 implements Runnable {
		public void run() {
			synchronized (lock1) {
				System.out.println("Thread 1: Holding lock 1...");
				try {
					Thread.sleep(10);
				} catch (InterruptedException e) {
				}
				System.out.println("Thread 1: Waiting for lock 2...");
				synchronized (lock2) {
					System.out.println("Thread 1: Holding lock 1 & 2...");
				}
			}
		}
	}
	private static class ThreadDemo2 implements Runnable {
		public void run() {
			synchronized (lock2) {
				System.out.println("Thread 2: Holding lock 2...");
				try {
					Thread.sleep(10);
				} catch (InterruptedException e) {
				}
				System.out.println("Thread 2: Waiting for lock 1...");
				synchronized (lock1) {
					System.out.println("Thread 2: Holding lock 1 & 2...");
				}
			}
		}
	}
}
```

The actual reason for deadlock is not multiple threads but the way they are requesting locks, if we provide **an ordered access on locks** then problem will be resolved, here is my fix, which avoids deadlock by avoiding circular wait.

Example #2:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class DeadLockSolution {
	public static Object lock1 = new Object();
	public static Object lock2 = new Object();
	public static void main(String args[]) {
		ExecutorService threadPool = Executors.newFixedThreadPool(2);
		threadPool.execute(new ThreadDemo1());
		threadPool.execute(new ThreadDemo2());
		threadPool.shutdown(); //put debug point here
	}

	private static class ThreadDemo1 implements Runnable {
		public void run() {
			synchronized (lock1) {
				System.out.println("Thread 1: Holding lock 1...");
				try {
					Thread.sleep(10);
				} catch (InterruptedException e) {
				}
				System.out.println("Thread 1: Waiting for lock 2...");
				synchronized (lock2) {
```

```
                                System.out.println("Thread 1: Holding lock 1 & 2...");
                            }
                        }
                    }
                }
                private static class ThreadDemo2 implements Runnable {
                    public void run() {
                        synchronized (lock1) {
                            System.out.println("Thread 2: Holding lock 2...");
                            try {
                                Thread.sleep(10);
                            } catch (InterruptedException e) {
                            }
                            System.out.println("Thread 2: Waiting for lock 1...");
                            synchronized (lock2) {
                                System.out.println("Thread 2: Holding lock 1 & 2...");
                            }
                        }
                    }
                }
            }
```

Race Condition

Race condition is a type of concurrency bug or issue which is presented by the use of multi-threading just like deadlock.

Race condition occurs when two threads operate on same object without proper synchronization and their operation interleaves on each other.

Classical example of Race condition is incrementing a counter, since increment is not an atomic operation and can be further divided into three steps like read, update and write. If two threads tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread.

Atomic operations are not subjected to race condition because those operations cannot be interleaved.

There will be many race condition problems; one among them is **read-modify-write** race condition.

The read-modify-write pattern comes due to improper synchronization of non-atomic operations or combination of two individual atomic operations which is not atomic together.

Example:

```
    if(!map.contains(key)){ //key does't exist
        ….
        map.put(key,value);
    }
```

Consider thread T1 checks for condition and goes inside if block. Now CPU is switched from thread T1 to thread T2 which also checks condition and goes inside if block. Now we have two threads inside if block which result in either T1 overwriting T2 value or vice-versa based on which thread has CPU for execution.

Example #1: Non thread-safe code

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import sun.org.mozilla.javascript.internal.Synchronizer;
/**
 * Non Thread-Safe Class in Java
 */
class Counter {
	private int count=0;
	/*
	 * This method is not thread-safe because ++ is not an atomic operation
	 */
	public int getCount() {
		System.out.println("******");
		try {
			Thread.sleep(10);
		} catch (InterruptedException e) {  e.printStackTrace(); }
		return count++;
	}
}
public class RaceConditionEx1 {
	public static void main(String[] args) {
		final Counter c = new Counter();
		ExecutorService threadPool = Executors.newFixedThreadPool(2);
		threadPool.execute(new Runnable() {
			@Override
			public void run() {
				System.out.println(c.getCount());
			}
		});
		threadPool.execute(new Runnable() {
			@Override
			public void run() {
				System.out.println(c.getCount());
			}
		});
		threadPool.shutdown();
		try {  Thread.sleep(500);
```

```
                    } catch (InterruptedException e) {  e.printStackTrace(); }
                    System.out.println("End:"+c.getCount());
          }
}
o/p:
******
******
0
0
******
End:1
```

There are multiple ways to fix race condition in Java:

1) Use **synchronized keyword** in Java and lock the getCount() method so that only one thread can execute it at a time which removes possibility of coinciding or interleaving.

```
//This method thread-safe now because of locking and synchronization
public class Counter{
          private int count;
          public synchronized int getCount() {
                    System.out.println("******");
                    try {  Thread.sleep(1);
                    } catch (InterruptedException e) {  e.printStackTrace(); }
                    return count++;
          }
}
```

O/P:

```
******
******
0
1
******
End:2
```

2) Use **java.util.concurrent.atomic.AtomicInteger**, which makes this ++ operation as atomic and since atomic operations are thread-safe and saves cost of external synchronization.

```
class Counter {
             private AtomicInteger atomicCount = new AtomicInteger( 0 );
             /**
             * This method is thread-safe because count is incremented atomically
             */
            public int getCount(){
                return atomicCount.incrementAndGet();
            }
}
```

Semaphore

In java, counting Semaphore is used to specify number of passes or permits.

It safely allows us to ensure that only **n** threads can access a certain resource at a given time.

In order to access a shared resource, current thread must acquire a permit.

If permit is already exhausted by other thread then it should wait until a permit is released by different thread.

Note: This concurrency utility can be very useful to implement Producer Consumer design pattern.

The counting Semaphore is initialized with number of permits.

The Semaphore class provides two methods:

1) acquire() : This method is used to get permit if the permit is available otherwise this method blocks until the permit is available.
2) release() : The method is used to release permit.

The following example on **Binary Semaphore** demonstrates how **mutual exclusion** can be achieved using semaphore:

Example:

```
import java.util.concurrent.Semaphore;
/**
 * A Counting semaphore with one permit is known as binary semaphore because it has only two states
 * which is either permit available or permit unavailable. Binary semaphore can be used to implement
 * mutual exclusion or critical section where only one thread is allowed to execute. Thread will wait on
 * acquire() until Thread  inside critical section release permit by calling release() on semaphore.
 */
public class SemaphoreTest {
        Semaphore binary = new Semaphore(1); //one permit here
        public static void main(String args[]) {
                final SemaphoreTest test = new SemaphoreTest();
                ExecutorService threadPool = Executors.newFixedThreadPool(2);

                threadPool.execute(new Runnable() {
                        @Override
                        public void run() { test.mutualExclusion(); }
                   }
                );
                threadPool.execute(new Runnable() {
                        @Override
                        public void run() {  test.mutualExclusion(); }
                 }
                );
```

```
		threadPool.shutdown();
	}
	private void mutualExclusion() {
		try {
			binary.acquire();
			// mutual exclusive region
			System.out.println(Thread.currentThread().getName() + " inside mutual exclusive region");
			Thread.sleep(1000);
		} catch (InterruptedException ie) {
			ie.printStackTrace();
		} finally {
			binary.release();
			System.out.println(Thread.currentThread().getName() + " outside of mutual exclusive region");
		}
	}
}
```

Semaphore is useful in different scenarios where we have to limit the amount of concurrent access to certain parts of our application.

Example #2:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
public class SemaphoreTest2 {
	public static void main(String args[]) {
		Semaphore semaphore = new Semaphore(5); // five permits here
		ExecutorService threadPool = Executors.newFixedThreadPool(10);
		Runnable longRunningTask = new Runnable() {
			@Override
			public void run() {
				boolean permit = false;
				try {
					permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
					if (permit) {
						System.out.println("Semaphore acquired");
						Thread.sleep(3 * 1000);
					} else {
						System.out.println("Could not acquire semaphore");
					}
				} catch (InterruptedException e) {
					throw new IllegalStateException(e);
```

```
                        } finally {
                                if (permit) {
                                        semaphore.release();
                                }
                        } // finally blk
                }
        };

        threadPool.execute(longRunningTask); // get semaphore
        threadPool.execute(longRunningTask); // get semaphore
        threadPool.execute(longRunningTask); // get semaphore
        threadPool.execute(longRunningTask); // get semaphore
        threadPool.execute(longRunningTask); // get semaphore

        threadPool.execute(longRunningTask); // could not acquire semaphore
        threadPool.execute(longRunningTask); // could not acquire semaphore
        threadPool.execute(longRunningTask); // could not acquire semaphore
        threadPool.execute(longRunningTask); // could not acquire semaphore
        threadPool.execute(longRunningTask); // could not acquire semaphore

        threadPool.shutdown();
    }
}
```

The executor can potentially run 10 tasks concurrently but we use a semaphore of size 5, thus limiting concurrent access to 5.

ForkJoin

The problem with the executor framework is that a Callable is free to submit a new sub-task to its executor and wait for its result. The issue is that of parallelism: When a Callable waits for the result of another Callable, it is put in a waiting state, and thus wasting an opportunity to handle another Callable queued for execution.

To solve this issue Java 7 has newly added fork-join framework.

It divides one task into several small tasks as a new fork means child and joins all the forks when all the sub-tasks complete.

The fork-join executor framework is responsible for creating one new task object which is again responsible for creating new sub-task object and waiting for sub-task to be completed. Internally it maintains a thread pool and executor assign pending task to this thread pool to complete when one task is waiting for another task to complete.

The **fork**() method allows a new ForkJoinTask to be launched from an existing one.

The **join**() method allows a ForkJoinTask to wait for the completion of another one.

INTERVIEW QUESTIONS

1. What is the relation between the two interfaces Executor and ExecutorService?
   Ans) The interface Executor only defines one method: execute(Runnable). Implementations of this interface will have to execute the given Runnable instance at some time in the future. The ExecutorService interface is sub interface of the Executor interface and provides additional methods to shut down the underlying implementation, to await the termination of all submitted tasks and it allows submitting instances of Callable.

2. What happens when we `submit()` a new task to an ExecutorService instance whose queue is already full?
   Ans) As the method signature of submit() indicates, the ExecutorService implementation is supposed to throw a RejectedExecutionException.

3. What are differences between Runnable and Callable?
   Ans)

Java.lang.Runnable<<interface>>	**Java.util.concurrent.Callable<<interface>>**
This interface contains **run**() method, which doesn't return value and doesn't throw exception. Public void run()	This interface contains **call**() method, which returns task result and throws exception. Public Object call()throws Exception
The runnable tasks can be submitted to the executor service using either execute() or submit() methods.	The callable tasks are always submitted to the executor service using **submit**() method.

4. What is Producer Consumer problem and its solution?
   Ans) Producer should wait if queue is full and Consumer should wait if queue is empty. We can implement Producer Consumer problem using put() and take() methods in BlockingQueue.

5. What are the differences between synchronized version of HashMap and ConcurrentHashMap?

Synchronized version of HashMap	**ConcurrentHashMap**
The whole HashMap is locked hence only one thread is allowed at a time to read or modify i.e., poor performance.	Only part means **segment** of ConcurrentHashMap is locked hence at a time multiple threads [default 16] are allowed to read or modify. Hence achieves very high level of concurrency i.e., better performance.
It will throw ConcurrentModification exception if we try to modify HashMap while iterating it.	It will not throw ConcurrentModification exception if we try to modify ConcurrentHashMap while iterating it.

6. What is CountDownLatch?
   Ans) This class is used to **disable** (means wait) main processing thread for thread scheduling until other thread(s) complete their processing.

The **awaits**() is used to disable main processing thread and **countDown**() is used to reduce count by one. Once count reaches zero then only main processing thread will start running.

7. What is CyclicBarrier?
Ans) It allows multiple threads to wait for each other at barrier before proceeding.
All threads should call CyclicBarrier.**await**() to wait for each other to reach barrier.
CyclicBarrier is initialized with number of threads to be waited for each other.
We can reuse CyclicBarrier by calling **reset**() method which resets barrier to its initial state.

8. What are the differences between CountDownLatch and CyclicBarrier?

CountDownLatch	**CyclicBarrier**
We cannot reuse CountDownLatch once count reaches to zero.	We can reuse CyclicBarrier by calling **reset**() method which resets barrier to its initial State.
Main processing thread waits at latch until all its dependent threads finish their processing i.e., await() called only once inside main processing thread.	All threads wait for each other at barrier until all treads are ready. Hence all threads must call await() method.

9. What is deadlock and what is the solution?
Ans) Deadlock occurred if two or more threads are waiting means blocking for each other forever.
One of the solutions is if we provide **an ordered access on locks** then this problem will be resolved.

10. What is Race condition?
Ans) Race condition is a type of concurrency bug or issue which occurs when two threads operate on same object without proper synchronization and their operations interleave on each other. One of the examples for race condition is **read-modify-write**.
There are multiple ways to fix race condition problem. Two solutions among them are synchronized keyword and java.util.concurrent.atomic.AtomicInteger.

11. What is Semaphore?
Ans) It is used to achieve mutual exclusion using **acquire**() and **release**() methods.

12. What is ForkJoin?
Ans) It divides one task into several small tasks as a new fork means child and joins all the forks when all the sub-tasks complete. The **fork**() method allows a new ForkJoinTask to be launched from an existing one. The **join**() method allows a ForkJoinTask to wait for the completion of another one.

Summary:

Concurrency	**Methods**
CountDownLatch	Await() & countDown()

CyclicBarrier	Await() & reset()
Producer Consumer Pattern	Put() & take()
Semaphore	Acquire() & release()
ForkJoin	fork() & join()

Please visit http://jvmmonitor.org/index.html for more details on **Java profiling** using JVM Monitor plugin for eclipse.

For other Java Profiling tools, please visit:
https://blog.idrsolutions.com/2014/06/java-performance-tuning-tools/