10-java-concurrency-model

Contents

- Thread model
  - Terms
  - Creation
  - Synchronization
  - Deadlocks
  - Livelocks
  - Atomics
  - Classes
    - CyclicBarrier
    - Collections
    - CopyOnWriteArrayList
  - Executors
  - Callable
  - ExecutorService

Thread model

Concurrency is gaining importance with more widespread use of multi-core processors. The Latin root of the work concurrency means - running together. In programming you can have multiple threads running in parallel in a program executing different tasks at the same time. When used correctly concurrency can improve the performance and responsiveness of the application and hence it is a powerful and useful feature. From the beginning Java has supported concurrency in the form of low-level threads management, locks and synchronization. Since 5.0, it also supports high level concurrency API in its concurrent package. From

version 8, Java has gotten even better support for concurrency with the introduction of parallel streams.

Terms

Here are some of the more critical terms which need to be understood

- Critical section - user object used for allowing the execution of just one active thread from many others `within one process`. The other non selected threads which try to acquire this object are put to sleep
- Mutex - Kernel object used for allowing the execution of just `one active thread` from many others `among different processes`. The other non selected threads which try to acquire this object are put to sleep
- Lock - Kernel object used for allowing the execution of just `one active thread` from many others `within the same process`. The other non selected threads which are trying to acquire the object are put to sleep
- Semaphore - Kernel object used for allowing the execution of a group of active threads, from many others. The other non selected threads trying to acquire this object are put to sleep. It can be used for interprocess/shared actions but is not recommended due to it not being very safe, lacking some of the properties and attributes of the shared mutex (mentioned above)
- Monitors - are special objects which comprise of the primitive object `Mutex` and `Conditional variable` mentioned above, they are a synchronization construct to control access to shared resources. Provides a mutual exclusion, ensuring that only one thread can execute critical section at a time, and condition synchronization, allowing threads to wait for a specific condition to be met

```
In Java every object has an intrinsic monitor associated with it. This monitor can be
used to synchronize access to the objects's methods or blocks of code. It is comprised of
 the intrinsic monitor locks (mutex or a lock)and the wait-notify mechanism (conditional
synchronization)
```

Creation

The `Thread` and Object classes and the `Runnable` interface provide the necessary support for concurrency in Java. The Thread class has methods such as run, start and sleep that are useful for multi-threading. The Object class has methods such as wait and notify that support concurrency and are designed to be used for that purpose actually. Since every class in Java derives from the Object class all the objects have some basic multi-threading capabilities. It is also very easy to acquire a lock on an object/instance using the synchronized keyword.

Method	Type	Description
Thread currentThread()	Static method	Returns reference to the current thread.
String getName()	Instance method	Returns the name of the current thread.
int getPriority()	Instance method	Returns the priority value of the current thread.
void join() void join(long) void join(long, int)	Overloaded instance methods	The current thread invoking join on another thread waits until the other thread completes. You can optionally give the timeout in milliseconds (given in long) or timeout in milliseconds as well as nanoseconds (given in long and int).

Method	Type	Description
void run()	Instance method	Once you start a thread (using the start() method), the run() method will be called when the thread is ready to execute.
void setName(String)	Instance method	Changes the name of the thread to the given name in the argument.
void setPriority(int)	Instance method	Sets the priority of the thread to the given argument value.
void sleep(long) void sleep(long, int)	Overloaded static methods	Makes the current thread sleep for given milliseconds (given in long) or for given milliseconds and nanoseconds (given in long and int).
void start()	Instance method	Starts the thread; JVM calls the run() method of the thread.
String toString()	Instance method	Returns the string representation of the thread; the string has the thread's name, priority, and its group. Creating

To create thread objects, one can extend the Thread class, and override the run method. If the method is not overridden the default method will be run, which does nothing. To override the run method, it needs to be declared as public, it takes no arguments and has a void return type - `public void run()`

```
class MyThread extends Thread {
    public void run() {
        try {
            // sleep the current thread for 1 second, then print out the
               current thread name
            sleep(1000);
            System.out.println("In run(); thread name is: " + Thread.
               currentThread().getName());
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
            // ignore the `InterruptedException` - this is perhaps the one of
                the
            // very few of the exceptions in Java which is acceptable to
               ignore
            // if it happens there is really not much that can be done
               anyways
            // from the user space that is
        }
        System.out.println("In run(); thread name is: " + getName());
    }
    public static void main(String args[]) {
        // create a plain thread object, this is a software thread, it is
           obtained directly from the operating system
        Thread myThread = new MyThread();
        // start the thread execution, this will make sure to invoke the run
           method
        myThread.start();
        // start method is not blocking meaning that the println will be
           invoked immediately, here for the main thread
```

```
        System.out.println("In main(); thread name: " + Thread.currentThread
            ().getName());
    }
}
```

There is also another way to create a thread, instead of extending the Thread class, one can implement an anonymous class from the `Runnable` interface as well. The Thread class itself, implements Runnable interface. The Runnable interface declares a sole method, run(). Hence when one implements Runnable interface the run method has to have an implementation. The Thread class provides a constructor that accepts a Runnable argument, that way one can create a new thread with a Runnable target as well

```
class RunnableImpl implements Runnable {
    public void run() {
        // sleep the current thread for 1 second, then print out the current
            thread name
        sleep(1000);
        System.out.println("In run(); thread name is: " + Thread.
            currentThread().getName());
    }
    public static void main(String args[]) throws Exception {
        // create a plain thread object, this is a software thread, it is
            obtained directly from the operating system
        Thread myThread = new Thread(new RunnableImpl());
        // start the thread execution, this will make sure to invoke the run
            method
        myThread.start();
        // start method is not blocking meaning that the println will be
            invoked immediately, here for the main thread
        System.out.println("In main(); thread name is: " + Thread.
            currentThread().getName());
    }
}
```

The example above creates a Thread object from a Runnable instead of extending the Thread class, this is generally much simpler and easier way to manage threads, it also allows Threads to be re-usable and less exposed to the user land.

Synchronization

Threads share memory, and they can concurrently modify data. Since the modification can be done at the same time without safeguards, this can lead to unintuitive results. When to or more threads are trying to access a variable and one of them wants to modify it, you get a problem known as race condition, data race or race hazard. To solve this problem data modification has to be done in an atomic way. Meaning that only one thread must be allowed to change the data at a given time, this is done through locks and mutexes. In java land those are called synchronized blocks. This can avoid the race condition by locking the data being modified that ensures that only one thread at a time can enter the modification block and only once it exists the modification block then other threads are allowed to modify the data as well - this is called a `critical section`

```
synchronized(objectInstance) {
    // critical section - code segment guarded by the mutex lock
}
```

Synchronized blocks will never be left in only-locked state, even if exception is thrown, before the block finishes, or for some other reason the block is unable to complete fully, the lock will still be released, that ensures that there is no dead-lock, meaning that if the lock was to remain stuck in locked state, other threads will wait indefinitely on it.

There is also a way to synchronize methods too, the keyword is put at the front of the method declaration instead. The entire method in that case is synchronized, In that case when the method declared as synchronized is called a lock is obtained on the object on which the method is called, in this case the instance of the class, it is released when the method exits - it is somewhat equivalent to doing - `synchronized(this){ /* code */ }`

Static methods can also be declared as synchronized, however in that case the target of the synchronization lock is not the class instance it is the class type itself, remember that in Java even the class type is in a way treated as an instance of the class type itself, so it will lock around like that - `synchronized(ClassType.class){ /* code */ }`

Constructors can not be declared synchronized, this is because there is no instance to lock around in the first place, this will produce a compiler error. You can however have synchronized block inside the constructor which locks around some of the data members or input arguments themselves

Why can't you declare constructors synchronized? The JVM ensures that only one thread can invoke a constructor call (for a specific constructor)at a given point in time. So, there is no need to declare a constructor synchronized

It is common to misunderstand that a synchronized block obtains a lock for a block of code. Actually, the lock is obtained for an object and not for a piece of code. The obtained lock is held until all the statements in that block complete execution.

Deadlocks

A deadlock arisees when locking threads results in a situation where they canot prodceed and thus wait indefinitely for others to terminate.Say one thread acquires a lock on reosurce r1 and waits to acquire another on resource r2, At the same time say there is another thread that has already acquired a lock around r2 and is waiting to obtain a lock on r1. Neither of the threads can proceed until the other one releases the lock, which never happens - to they are stuck in a deadlock.

```
// Balls class has a globally accessible data member to hold the number of
   balls thrown
class Balls {
    public static long balls = 0;
}
// Runs class has a globally accessible data member to hold the number of
   runs scored
class Runs {
    public static long runs = 0;
}
class CounterOne implements Runnable {
    // this method increments runs variable first and then increments the
       balls variable
    // since these variables are accessible from other threads,
    // we need to acquire a lock before processing them
    @Override
    public void run() {
        // since we're updating runs variable first, first lock the Runs.
           class
```

```
        synchronized(Runs.class) {
            // lock on Balls.class before updating balls variable
            synchronized(Balls.class) {
                Runs.runs++;
                Balls.balls++;
            }
        }
    }
}
class CounterTwo implements Runnable {
    // this method increments balls variable first and then increments the
       runs variable
    // since these variables are accessible from other threads,
    // we need to acquire a lock before processing them
    @Override
    public void run() {
        // since we're updating balls variable first; so first lock Balls.
           class
        synchronized(Balls.class) {
            // acquire lock on Runs.class before updating runs variable
            synchronized(Runs.class) {
                Balls.balls++;
                Runs.runs++;
            }
        }
    }
}
public class DeadLock {
    public static void main(String args[]) throws InterruptedException {
        CounterOne c1 = new CounterOne();
        CounterTwo c2 = new CounterTwo();
        // create two threads and start them at the same time
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
        t1.start();
        t2.start();
        System.out.println("Waiting for threads to complete execution...");
        t1.join();
        t2.join();
        System.out.println("Done.");
    }
}
```

It is clear in the example above that depending on which thread starts first, and runs its method, it will acquire lock around `Runs (CounterOne)`, however it is possible, that the other acquires lock around `Balls (CounterTwo)`. In that case they will never be able to obtain the inner synchronized block lock since the other thread is already holding it - leading to deadlock

It is not guaranteed that this program will lead to a deadlock every time it is started, it is quite dependent on very specific circumstances but it is possible, and that is the problem. The sequence in which the threads are executed and the order in which the locks are acquired and released, the time it takes to obtain the lock

and so on, all play a role in a deadlock situation like that

Livelocks

Assume that there are two robotic cars that are programmed to automatically drive in the road. There is a situation where two robotic cars reach the two opposite ends of a narrow bridge. The bridge is so narrow that only one car can pass through at at time. The robotic cars are programmed such that they wait for the other car to pass through first. When both the cars attempt to enter the bridge at the same time, the following situation could happen each car starts to enter the bridge notices that the other car is attempting to do the same and reverses. Note that the cars keep moving forward and backward and thus appear as if they are doing lots of work, but there is no progress made by either of the cars. This situation is called a `livelock`

Consider two threads `t1` and `t2` assume that thread `t1` makes a change and thread `t2` undoes that change. When both the threads `t1` and `t2` work, it will appear as though lots of work is getting done, but no progress is made. This situation is called a `livelock in threads`

Consider the situation in which numerous threads have different priorities assigned to them in the range of lowest priority 1 to highest priority 10 which is the range allowed for priority of threads in Java. When a lock is available the thread scheduler will give priority to the threads with priority over low priority. If there are many high prio threads that want to obtain the lock and also hold the lock for long time periods when will the low priority threads get a chance to obtain the lock. In other words in a situation where low priority threads starve for a long time trying to obtain the lock is known as lock starvation. There are many techniques available for detecting or avoiding threading problems like `livelocks` and starvation but they are not within the scope of these discussions.

As mentioned monitors in java are a way to make the locking mechanism more flexible and allow threads to inter cooperate, in the example below

```
class SharedResource {
    private Queue<Integer> queue = new LinkedList<>();
    private int capacity = 5;

    public synchronized void produce(int value) throws InterruptedException {
        while (queue.size() == capacity) {
            wait(); // Wait if the queue is full
        }
        queue.add(value);
        notifyAll(); // Notify consumers
    }

    public synchronized int consume() throws InterruptedException {
        while (queue.isEmpty()) {
            wait(); // Wait if the queue is empty
        }
        int value = queue.poll();
        notifyAll(); // Notify producers
        return value;
    }
}

public static void main(String[] args) {
    SharedResource sharedResource = new SharedResource();
```

```
    // Producer thread
    Thread producer = new Thread(() -> {
        try {
            for (int i = 1; i <= 10; i++) {
                sharedResource.produce(i);
                Thread.sleep(100); // Simulate production delay
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Producer interrupted");
        }
    });

    // Consumer thread
    Thread consumer = new Thread(() -> {
        try {
            for (int i = 1; i <= 10; i++) {
                sharedResource.consume();
                Thread.sleep(150); // Simulate consumption delay
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Consumer interrupted");
        }
    });

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

    // Wait for both threads to complete
    try {
        producer.join();
        consumer.join();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.err.println("Main thread interrupted");
    }

    System.out.println("Producer and Consumer have finished.");
}
```

Atomics

The concurrent package has two sub-packages atomic and locks. Below are discussed the atomic variables in the atomic package. Often one desires to see code that acquires and releases locks for implementing primitive simple operations like `incrementing` a variable, `decrementing` a variable and so on. Acquiring and releasing locks for such primitive operations is not efficient. In such cases Java provides an efficient alternative in the form of atomic variables. Here is a list of some of the classes in this package and their short description

- AtomicBoolean - Atomically updates a boolean value primitive

- AtomicInteger - Atomically updates integer primitive value; inherits from the number Class
- AtmoicIntegerArray - An integer array in which elements ca be updated atomically
- AtomicLong - Atomically update-able long value; inherits from the Number class
- AtomicLongArray - A long array in which elements can be updated atomically
- AtmoicRefernce - An atomically update-able object reference of type V
- AtomicRefernceArray - An atomically update-able array that can hold object references of type E (E refers to the base type of the elements within the array)

`Only AtmoicInteger & AtomicLong extend from the Number class but not AtomicBoolean. All other classes in the atomic sub-package inherit directly from the Object class`

Of the classes mentioned above, the AtomicInteger and AtomicLong are the most important. The table below lists some of the most important methods from these classes as well.

Method	Description
AtomicInteger()	Creates an instance of AtomicInteger with initial value 0.
AtomicInteger(int initVal)	Creates an instance of AtomicInteger with initial value initVal.
int get()	Returns the integer value held in this object.
void set(int newVal)	Resets the integer value held in this object to newVal.
int getAndSet(int newValue)	Returns the current int value held in this object and sets the value held in this object to newVal.
boolean compareAndSet(int expect, int update)	Compares the int value of this object to the expect value, and if they are equal, sets the int value of this object to the update value.
int getAndIncrement()	Returns the current value of the integer value in this object and increments the integer value in this object. Similar to the behavior of i++ where i is an int.
int getAndDecrement()	Returns the current value of the integer value in this object and decrements the integer value in this object. Similar to the behavior of i– where i is an int.
int getAndAdd(int delta)	Returns the integer value held in this object and adds given delta value to the integer value.
int incrementAndGet()	Increments the current value of the integer value in this object and returns that value. Similar to the behavior of ++i where i is an int.
int decrementAndGet()	Decrements the current integer value in this object and returns that value. Similar to behavior of –i where i is an int.
int addAndGet(int delta)	Adds the delta value to the current value of the integer in this object and returns that value.
int intValue() long longValue() float floatValue() double doubleValue()	Casts the current int value of the object and returns it as int, long, float, or double values.

Classes

There are many classes and interfaces in the concurrent package, that provide high-level `APIs` for concurrent programming. When one uses the synchronized keyword, it employs `mutexes` to synchronize between threads for safe shared access. Threads also often needed to coordinate their executions to complete a bigger higher-level task. It is possible to build higher level abstractions for thread synchronization. These high-level abstractions for synchronizing activities of two or more threads are known as `synchronizers`. `Synchronizers`, internally make use of the existing low level API for thread coordination.

- `Semaphore` controls access to shared resource. A semaphore maintains a counter to specify number of resources that the semaphore controls.
- `CountjdownLatch` allows one or more threads to wait for countdown to complete.
- `Exchanger` class is meant for exchanging data between two threads. This class is useful when two threads need to synchronize between each other and continuously exchange data.
- `CyclicBarrier` - helps provide a synchronization point where threads may need to wait at a predefined execution point until all other threads reach that point.
- `Phaser` - is a useful feature when few independent threads have to work in phases to compete a task

CyclicBarrier

There are many situations in concurrent programming where threads may need to wait at a predefined execution point until all other thread reach that point `CyclicBarrier` helps provide such a synchronization point. In other words it allows a set of threads to wait for each other at common barrier point before continuing execution. It is useful when one needs multiple threads to perform their tasks independently but then synchronize at a certain point before proceeding to the next phase.

- Fixed - the number of threads that must reach the barrier point when creating the `CyclicBarrier`
- Action - one can specify an optional `barrier action` which is a task executed by one of the threads when all reach the barrier
- Reusable - the barrier can be reused after the threads are released, hence the name `cyclic`
- Blocking - threads calling await are `blocked until required` number of threads have called it.

So how does it work, well threads call the await method, when they reach the barrier, in other words during some time of the execution, the barrier implies that the thread has reached the point at which it can perform no more work, and needs to wait for other threads to finish their work. If the required number of threads (specified in the constructor of the `CyclicBarrier` have called await(), the barrier is broken and all waiting threads are released. If there is a barrier action it is executed by one of the threads just before the barrier is broken

Method	Description
CyclicBarrier(int numThreads)	Creates a CyclicBarrier object with the number of threads waiting on it specified. Throws IllegalArgumentException if numThreads is negative or zero.
CyclicBarrier(int parties, Runnable barrierAction)	Same as the previous constructor; this constructor additionally takes the thread to call when the barrier is reached.
int await()	Blocks until the specified number of threads have called await() on this barrier. The method returns the arrival index of this thread. This method can throw an InterruptedException if the thread is interrupted while waiting for other threads or a BrokenBarrierException if the barrier was broken for some reason (for example, another thread was timed-out or interrupted).The overloaded method takes a time-out period as an additional option; this overloaded version throws a TimeoutException if all other threads aren't reached within the time-out period.
int await(long timeout, TimeUnit unit)	
boolean isBroken()	Returns true if the barrier is broken. A barrier is broken if at least one thread in that barrier was interrupted or timed-out, or if a barrier action failed throwing an exception.

void reset()	Resets the barrier to the initial state. If there are any threads waiting on that barrier, they will throw the BrokenBarrier exception.

```
// The run() method in this thread should be called only when
// four players are ready to start the game
class MixedDoubleTennisGame extends Thread {
    public void run() {
        System.out.println("All four players ready, game starts \n Love all
           ...");
    }
}
// This thread simulates arrival of a player.
// Once a player arrives, he/she should wait for other players to arrive
class Player extends Thread {
    CyclicBarrier waitPoint;
    public Player(CyclicBarrier barrier, String name) {
        this.setName(name);
        waitPoint = barrier;
        this.start();
    }
    public void run() {
        System.out.println("Player " + getName() + " is ready ");
        try {
            waitPoint.await(); // await for all four players to arrive
        } catch(BrokenBarrierException | InterruptedException exception) {
            System.out.println("An exception occurred while waiting... "
                + exception);
        }
    }
}
// Creates a CyclicBarrier object by passing the number of threads and the
   thread to run
// when all the threads reach the barrier
class CyclicBarrierTest {
    public static void main(String []args) {
        // a mixed-double tennis game requires four players;
        // so wait for four players
        // (i.e., four threads) to join to start the game
        System.out.println("Reserving tennis court \n"
            + "As soon as four players arrive, game will start");
        CyclicBarrier barrier = new CyclicBarrier(4, new
           MixedDoubleTennisGame());
        new Player(barrier, "G I Joe");
        new Player(barrier, "Dora");
        new Player(barrier, "Tintin");
        new Player(barrier, "Barbie");
    }
}
```

In the example above, the premise is quite simple, a tennis game is required to have 4 players, until 4 players

are present the game can not start, the start of the game itself is represented with the class MixedDoubleTennisGame, which is representing the action that has to be executed when the 4 threads call the await method on the CyclicBarrier object. Once all 4 threads do call the await method, then the run method of the MixedDoubleTennisGame will be run, and the game will start

```
Reserving tennis court
As soon as four players arrive, game will start
Player Dora is ready
Player G I Joe is ready
Player Tintin is ready
Player Barbie is ready
All four players ready, game starts
Love all...
```

An example output from the program above, can look something like that, notice that until the 4 players have registered, i.e. the await methods are not triggered, the game will not start. This crude example, shows how a set of threads can be interlinked, or rather their work or actions, in such a way that only when they finish their work, another thread action can be initiated.

Collections

The concurrent package provides a number of classes that are thread-safe equivalents of the ones provided in the collections framework classes in the java util package. For example the `ConcurrentHashMap` is a concurrent equivalent of the `HashMap`, The `main` difference between these two containers is that one needs to explicitly synchronize insertions and deletions with `HashMap`, whereas such synchronization is built into the `ConcurrentHashMap`. If one knows how to use the interface of the `HashMap`, then the `ConcurrentHashMap` is no different.

Class	Description
BlockingQueue	This interface extends the Queue interface. In BlockingQueue, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
ArrayBlockingQueue	This class provides a fixed-sized array based implementation of the BlockingQueue interface.
LinkedBlockingQueue	This class provides a linked-list-based implementation of the BlockingQueue interface.
DelayQueue	This class implements BlockingQueue and consists of elements that are of type Delayed. An element can be retrieved from this queue only after its delay period.
PriorityBlockingQueue	Equivalent to java.util.PriorityQueue, but implements the BlockingQueue interface.
SynchronousQueue	This class implements BlockingQueue. In this container, each insert() by a thread waits (blocks) for a corresponding remove() by another thread and vice versa.
LinkedBlockingDeque	This class implements BlockingDeque where insert and remove operations could block; uses a linked-list for implementation.
ConcurrentHashMap	Analogous to Hashtable, but with safe concurrent access and updates.
ConcurrentSkipListMap	Analogous to TreeMap, but provides safe concurrent access and updates.
ConcurrentSkipListSet	Analogous to TreeSet, but provides safe concurrent access and updates.
CopyOnWriteArrayList	Similar to ArrayList, but provides safe concurrent access. When the container is modified, it creates a fresh copy of the underlying array.
CopyOnWriteArraySet	A Set implementation, but provides safe concurrent access and is implemented using CopyOnWriteArrayList. When the container is modified, it creates a fresh copy of the underlying array.

CopyOnWriteArrayList

Both `ArrayList` and `CopyOnWriteArrayList` implement the List interface. These are three main differences between both of those classes, when used in concurrent context

- `ArrayList` is not thread safe but `CopyOnWriteArrayList` is. That means it is unsafe to use `ArrayList` in contexts where multiple threads are executing on the same instance of the `ArrayList`, especially when a modification is being made
- Methods in `ArrayList` such as remove add and set methods can throw a `ConcurrentModificationException` if another thread modifies the `ArrayList` when on thread is accessing it. However it is safe to perform these operations from multiple threads on a `CopyOnWriteArrayList`; and hence methods such as remove add and set do not throw this exception. All the active iterators will still have access to the unmodified version of the container and hence they remain unaffected; if one tries to create an iterator after the modification one will get the iterator for the modified container
- Iterator can be obtained by calling the iterator() method on a List object; if remove() is called when the underlying container is modified an exception can be thrown. However one cannot call remove method on an iterator of a `CopyOnWriteArrayList`, it always throws `UnsupportedOperationException`

```
public class ModifyingList {
    public static void main(String []args) {
        List<String> aList = new ArrayList<>();
        aList.add("one");
        aList.add("two");
        aList.add("three");
        Iterator listIter = aList.iterator();
        while(listIter.hasNext()) {
            System.out.println(listIter.next());
            aList.add("four");
        }
    }
}
```

This example above shows how a `CopyOnWriteArrayList` can be also useful in non-concurrent context, this is because the example above, will throw `ConcurrentModificationException`, since an element is being added to the array, while the array is being iterated over, in this scenario one might want to replace the `ArrayList` with a `CopyOnWriteArrayList`.

The way the `CopyOnWriteArrayList` works, is by creating a copy of the container data when a new element is being added, meaning that if an iterator is obtained before a call to add (or any method that modifies the array) the iterator will still point to the original unmodified array, thus there is no exception being thrown, but the iterator will iterated over the original array, meaning that new elements being added will not be printed out (taking the example above)

Executors

Threads can be directly managed by the applications by creating Thread objects, however that is cumbersome to manage. If one wishes to abstract away this low level detail of a multi-threading programming, the executor services are a good choice. That interface declares only one method, which is `void execute(Runnable)`. The derived interfaces and classes such as `ExecutorService`, `ThreadPoolExecutor`, and `ForkJoinPool`, support useful functionality, which extends the base interface. The basic premise of the Executor chain/hierarchy of classes is that they are meant to provide re-usable containers for Thread objects, which aid the creation, destruction, reuse and running of threads.

Callable

Callable is an interface that declares one method `V call()`. It represents a task that needs to be completed by a thread. Once the task completes it returns a value. For some reason if the call method cannot execute or fails it throws an Exception. To execute a task using the Callable object, a thread pool first must be created. A thread pool is a collection of threads that can execute tasks.

ExecutorService

The `ExecutorService` interface extends the Executor interface and provides services such as termination of threads and production of Future objects. Some tasks may take considerable execution time to complete. So when one submits a task so the executor service, a Future object is returned back. Future represents object that contain a value that is returned by a thread in the future. In other words, the Future is the result of the action that would be executed by the thread, sometime in the future. The Future object has methods like - `isDone()`, that check if the task is complete and then a `get()` method that can be used to obtain the result of the task. Note that the `get()` method, is blocking, if it is called before the task is done, it will block the current thread until the task is done.

```
// Factorial implements Callable so that it can be passed to a
   ExecutorService
// and get executed as a task.
class Factorial implements Callable<Long> {
    long n;
    public Factorial(long n) {
        this.n = n;
    }
    public Long call() throws Exception {
        if(n <= 0) {
            throw new Exception("for finding factorial, N should be > 0");
        }
        long fact = 1;
        for(long longVal = 1; longVal <= n; longVal++) {
            fact *= longVal;
        }
        return fact;
    }
}
// Illustrates how Callable, Executors, ExecutorService, and Future are
   related;
// also shows how they work together to execute a task
class CallableTest {
    public static void main(String []args) throws Exception {
        // the value for which we want to find the factorial
        long N = 20;
        // get a callable task to be submitted to the executor service
        Callable<Long> task = new Factorial(N);
        // create an ExecutorService with a fixed thread pool having one
           thread
        ExecutorService es = Executors.newSingleThreadExecutor();
        // submit the task to the executor service and store the Future
           object
        Future<Long> future = es.submit(task);
```

```
        // wait for the get() method that blocks until the computation is
           complete.
        System.out.printf("factorial of %d is %d", N, future.get());
        // done. shutdown the executor service since we don't need it anymore
        es.shutdown();
    }
}
```

In this program the `Factorial` class implements Callable, Since the task is to compute the factorial of a number N, the task needs to return a result. In the example it is also important to note that for sake of simplicity the single threaded executor service is called with `newSingleThreadExecutor`, method ins the Executors class. Note that there are other methods such as `newFixedThreadPool` to create a thread pool with multiple threads depending on the level of parallelism one needs. The Executors class is mainly supposed to act as a factory for different Executor service implementations, which provide different types of executors, which in turn provide different capabilities for managing the native Thread objects

The Fork/Join framework in the concurrent package helps simplify writing `parallelized` code. The framework is an implementation of the `ExecutorService` interface and provides an easy-to-use concurrent platform in order to exploit multiple processors. This framework is very useful for modeling divide and conquer problems. This approach is suitable for tasks that can be divided recursively and computed on a smaller scale; the computed results are then combined. Dividing the task into smaller tasks is called forking and merging the results from the smaller tasks is called joining

The Fork/Join framework uses the work-stealing algorithm: when a worker thread completes all its work, and is free, it takes or steals, work from other threads that are still busy with doing some other work. Each thread in the fork/join framework has a queue with tasks, other threads can steal tasks from that queue.

The `ForkJoinPool` is the most important class in the fork/join framework, it is a thread pool for running fork/join tasks and it executes an instance of `ForkJoinTask`. It executes tasks and manages their `lifecycle`.

Briefly the Fork/Join algorithm is designed as follows, below the pseudo code for this algorithm is laid out

```
forkJoin {
    fork the task;
    join the task;
    compose the results;
}
```

```
doRecursive {
    if (task is small enough to be handled by a single thread) {
        compute the small task
        if there is a result to return, do so
    } else {
        divide or fork the task into two parts
        call compute on first task and obtain result
        join on second task, and obtain result
        combine results from both tasks
    }
}
```

In the `ForkJoin` framework there are a few other classes besides the `ForkJoinPool` which are used to describe the algorithm presented above

- `RecursiveTask<V>` - that is a task that can run in a ForkJoinPool the compute method returns a value of type V. It inherits from ForkJoinTask

- `RecursiveAction` - is a task that can run in a ForkJoinPool It is similar to RecursiveTask but does not return a value

Let us ascertain how to use the fork join framework in problem solving, here are the steps to use the framework

- First check whether the problem is suitable for the fork join approach or not. Remember that it is not suitable for all kinds of tasks, the problem at hand must follow some core characteristics

```
*   The problem can be designed as a recursive task where the task can be
   subdivided into smaller units and the
    results can be combined together.
*   The subdivided tasks are independent and can be computed separately
   without hte need for communication etween the
    tasks when computation is in process. (Of course after the computation is
        over, the results have to be joined together
```

- If the problem to be solved can be modeled recursively then define a task class that extends either RecursiveTask or RecursiveAction if a task returns a result extend from RecursiveTask otherwise extend from RecursiveAction.
- Override the compute method in the newly defined task class. The compute method actually performs the task if the task is small enough to be executed; or splits the task into subtasks and invoke them. The subtasks can be invoked either by invokeAll or fork method (use fork when the subtask returns a value). Use the join method to get the computed results.
- Merge the results, if computed from the subtasks.
- Instantiate `ForkJoinPool` create an instance of the task class and start the execution of the task using the invoke method on the ForkJoinPool instance.
- That is is - computation of the algorithm is done

The example below illustrates how one can compute the sum of 1..N numbers using fork join framework the range of numbers are divided into half until the range can be handled by a single thread. Once the range summation completes the result gets summed up together.

```
class SumOfNUsingForkJoin {
    // one million - we want to compute sum
    // from 1 .. one million
    private static long N = 1000_000;

    // number of threads to create for distributing the effort
    private static final int NUM_THREADS = 10;

    // This is the recursive implementation of the algorithm; inherit from
       RecursiveTask
    // instead of RecursiveAction since we're returning values.
    static class RecursiveSumOfN extends RecursiveTask<Long> {
        // from and to are range of values to sum-up
        long from, to;

        public RecursiveSumOfN(long from, long to) {
            this.from = from;
            this.to = to;
        }
```

```
        // the method performs fork and join to compute the sum if the range
        // of values can be summed by a threadremember that we want to divide
        // the summation task equally among NUM_THREADS) then, sum the range
        // of numbers from..to using a simple for loop;
        // otherwise, fork the range and join the results
        public Long compute() {
            if( (to - from) <= N/NUM_THREADS) {
                // the range is something that can be handled
                // by a thread, so do summation
                long localSum = 0;
                // add in range 'from' .. 'to' inclusive of the value 'to'
                for(long i = from; i <= to; i++) {
                    localSum += i;
                }
                return localSum;
            }
            else {
                // no, the range is too big for a thread to handle,
                // so fork the computation
                // we find the mid-point value in the range from..to
                long mid = (from + to)/2;

                // determine the computation for first half
                // with the range from..mid
                RecursiveSumOfN firstHalf = new RecursiveSumOfN(from, mid);
                // now, fork off that task
                firstHalf.fork();

                // determine the computation for second half
                // with the range mid+1..to
                RecursiveSumOfN secondHalf = new RecursiveSumOfN(mid + 1, to)
                    ;

                // now, wait for the first half of computing sum to
                // complete, once done, add it to the remaining part
                long resultSecond = secondHalf.compute();
                return firstHalf.join() + resultSecond;
            }
        }
    }
    public static void main(String []args) {
        // Create a fork-join pool that consists of NUM_THREADS
        ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);

        // submit the computation task to the fork-join pool
        long computedSum = pool.invoke(new RecursiveSumOfN(0, N));

        // this is the formula sum for the range 1..N
        long formulaSum = (N * (N + 1)) / 2;

        // Compare the computed sum and the formula sum
```

```
        System.out.printf("Sum for range 1..%d; computed sum = %d, " + "
           formula sum = %d %n", N, computedSum, formulaSum);
    }
}
```

Analyzing how this program works. In this program one wishes to compute the sum of the values in the range of 1..1,000,000. For the sake of simplicity, the program uses ten threads to execute the tasks. The class RecursiveSumOfN extends `RecursiveTask<Long>`. In that class long is used since the sum of numbers in each sub-range is a long value. In addition the RecursiveTask is chosen instead of RecursiveAction because each subtask returns value. If the subtask does not return a value, the RecursiveAction can be used instead.

In the compute method, the decision to either compute the sum or split the task is made. The condition on which that is done is `(to - from)<= N/NUM_THREADS`. This is the threshold value in this computation. In other words, if the range of values is within the threshold that can be handled by a task, then the computation is performed, otherwise the task is split recursively. Either the sum is computed using a simple for loop from the ranges, or the middle of the range is ound, afterwards that is handled in the recursive part of the algorithm.

recursive