Concurrency

Outline

- Objectives
- Synchronization
- Atomic Classes
- Concurrent Collections
- CyclicBarrier

Objectives

- Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.
- Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList.

Synchronization

Using the **synchronized** keyword on either a block or a method. That lock is acquired when a thread enters an unoccupied synchronized block or method.

Synchronize

Object

```
Object o = new Object();
synchronized (o) { // Get the lock of Object o
    // Code guarded by the lock
}

// Get the lock of the object this code belongs to
synchronized (this) {
   // Code guarded by the lock
}
```

Synchronize

Method

```
public synchronize void method() {
    // Code guarded by the lock
}
```

```
public void method() {
    synchronized (this) {
        // Code guarded by the lock
    }
}
```

Synchronize

Static method

```
class MyClass {
    synchronized static void method() {
        /** .. */
    }
}
```

```
class MyClass {
    static void method() {
        synchronized (MyClass.class) {
            /** ... */
        }
    }
}
```

Atomic Classes

Primitive and Single Object

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference<V>

Array

- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray<E>

AtomicBoolean

- get
- set
- getAndSet
- compareAndSet

AtomicInteger, AtomicLong

- get
- getAndAdd
- getAndDecrement
- getAndIncrement
- getAndSet
- set
- incrementAndGet
- decrementAndGet
- addAndGet
- compareAndSet

- accumulateAndGet
- updateAndGet
- getAndUpdate
- getAndAccumulate
- byteValue
- shortValue
- intValue
- longValue
- doubleValue
- floatValue

AtomicReference<V>

- get
- getAndSet
- set
- compareAndSet

- getAndUpdate
- updateAndGet
- accumulateAndGet
- getAndAccumulate

AtomicIntegerArray

Exactly like primitive version except first argument of every method start with index.

Concurrent Collections

The ***java.util.concurrent*** package provides some thread-safe classes equivalent to the collection classes of ***java.util***.

Concurrent Collections

- **BlockingQueue** (for queues)
- **BlockingDeque** (for deques)
- **ConcurrentMap** (for maps)
- **ConcurrentNavigableMap** (for navigable maps like TreeMap)
- **CopyOnWriteArrayList** (for lists)

BlockingQueue

	Blocks	Times Out
Insert	void put(E e)	boolean offer (E e, long timeout, TimeUnit unit)
Remove	E take()	E poll (long timeout, TimeUnit unit)

BlockingQueue

The main implementations of this interface are:

- ArrayBlockingQueue
- DelayQueue
- LinkedBlockingQueue
- LinkedTransferQueue
- PriorityBlockingQueue
- SynchronousQueue

BlockingDeque

deque stands for double-end queue

Head	Blocks	Times Out
Insert	void putFirst(E e)	boolean offerFirst (E e, long timeout, TimeUnit unit)
Remove	E takeFirst()	E pollFirst (long timeout, TimeUnit unit)

Tail	Blocks	Times Out
Insert	void putLast(E e)	boolean offerLast (E e, long timeout, TimeUnit unit)
Remove	E takeLast()	E pollLast (long timeout, TimeUnit unit)

ConcurrentMap

This interface represents a thread-safe map and it's implemented by the ConcurrentHashMap class.

- V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
- V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
- V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
- V getOrDefault(Object key, V defaultValue)
- V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
- V putIfAbsent(K key, V value)

All of above methods perform atomically

ConcurrentNavigableMap

It represents a thread-safe NavigableMap (like TreeSet) and is implemented by the ConcurrentSkipListMap class, sorted according to the natural ordering of its keys, or by a Comparator provided in its constructor.

CopyOnWriteArrayList

It represents a thread-safe List, similar to an ArrayList, but when it's modified (with methods like add() or set()), a new copy of the underlying array is created (hence the name).

call remove() on the Iterator is not supported (UnsupportedOperationException)

CyclicBarrier

It provides a synchronization point (a barrier point) where a thread may need to wait until all other threads also reach that point.

CyclicBarrier

Constructors

CyclicBarrier(int threads)

CyclicBarrier(int parties, Runnable barrierAction)

```
int await()
  throws InterruptedException,
         BrokenBarrierException

int await(long timeout, TimeUnit unit)
  throws InterruptedException,
         BrokenBarrierException,
         TimeoutException
```

Key Points

- Synchronization works with locks. Every object comes with a built-in lock and since there is only lock per object, only one thread can hold that lock at any time. You acquire a lock by using the synchronized keyword in either a block or a method.
- Static code can also be synchronized but instead of using this to acquire the lock of an instance of the class, it is acquired on the class object that every class has associated.

Key Points

- The java.concurrent.atomic package contains classes (like AtomicInteger) to perform atomic operations, which are performed in a single step without the intervention of more than thread.
- BlockingQueue represents a thread-safe queue and BlockingDeque represents a thread-safe deque. Both wait (with an optional timeout) for an element to be inserted if the queue/deque is empty or for an element to be removed if the queue/deque is full.

Key Points

- ConcurrentMap represents a thread-safe map and it's implemented by the ConcurrentHashMap class.
- ConcurrentNavigableMap represents a thread-safe NavigableMap (like TreeSet) and is implemented by the ConcurrentSkipListMap class.
- CopyOnWriteArrayList represents a thread-safe List, similar to an ArrayList, but when it's modified (with methods like add() or set()), a new copy of the underlying array is created (hence the name).

Key Points

- CyclicBarrier that provides a synchronization point (a barrier point) where a thread may need to wait until all other threads reach that point.