Volatile

- volatile keyword offers a lock-free mechanism for synchronizing access to an instance field.
- compiler and the virtual machine take into account that the field may be concurrently updated by another thread
- mark a Java variable as "being stored in main memory“ – R/W from main memory, not CPU cache
- does not ensure atomicity -> if two threads write to same variable volatile is not enough, only useful if one thread reads and the other writes

```
private boolean done;
public synchronized boolean isDone() {
return done; }
public synchronized void setDone() { done =
true; }

private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```

Atomic actions

- No side effects until action is complete
- Atomic action is carried out as a single unit of execution without any interference by another thread.
- Atomic actions:
  - Reads and writes for reference variables and primitive variables except long and double
  - Reads and writes for all volatile variables (including long and double)
- Compared to synchronization, competing threads are not suspended, however some of them might not perform the operation

Atomic variables

- java.util.concurrent.atomic
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicLong
  - AtomicLongArray
- AtomicInteger
  - addAndGet(int delta)
  - compareAndSet(int expect, int update)
  - decrementAndGet()
  - getAndDecrement()
  - lazySet(int newValue)
  - set(int newValue)

Synchronized counter

```
class SynchronizedCounter {
    private int c = 0;
    public synchronized void
increment() {
        c++;
    }
    public synchronized void
decrement() {
        c--;
    }
    public synchronized int
value() {
        return c;
    }
}
```

```
class AtomicCounter {
    private AtomicInteger c
= new AtomicInteger(0);

    public void increment(){
        c.incrementAndGet();
}
    public void decrement(){
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Exercise

- Orders are produced concurrently, each order unique id

```
public class Order {
    static int idCount = 0;
    int id;
    Order(){
      id = idCount++;
    }
```

vs

```
public class Order {
    static AtomicInteger atomicInteger=new
AtomicInteger(0);
    int id;
    Order(){
     id=atomicInteger.getAndIncrement();
    }
```

Deadlock

- Two threads blocked waiting for each other

```
public class Deadlock {
  public static void main(String[] args){
    final Object resource1 = "resource1";
    final Object resource2 = "resource2";
    Thread t1 = new Thread() {
      public void run() {
        //Lock resource 1
        synchronized(resource1){
          System.out.println("Thread 1: locked resource 1");
          try{
            Thread.sleep(500);
          } catch (InterruptedException e) {}
          synchronized(resource2){
            System.out.println("Thread 1: locked resource 2");         }}      }    };
    Thread t2 = new Thread(){
      public void run(){
        synchronized(resource2){
          System.out.println("Thread 2: locked resource 2");
          try{
                                    Thread.sleep(500);
                  } catch (InterruptedException e){}
          synchronized(resource1){
            System.out.println("Thread 2: locked resource 1");  }}}};
    t1.start();
    t2.start();}}
```

Other liveness issues

- Starvation
  - A thread cannot gain access to a shared resource because other threads monopolize the access to it
- Livelock
  - A thread acting in response to another thread`s action which in turn depends on the action of another thread

Levels of thread safety

- Immutable: no external synchronization is necessary
- Unconditionally thread-safe: Instances are mutable, but the class has internal synchronization
- Conditionally thread-safe: Some methods require external synchronization
- Not thread-safe: Each method invocation must be externally synchronized
- Thread-hostile: Not thread safe even if all methods are externally synchronized (few. E.g. System.setOut())

Immutable classes design

- No setters
- private and final fields
- final methods
- no sharing of references to mutable fields -> use defensive copies

```
public final class Complex {
private final double re;
private final double im;
public Complex(double re, double im) {
this.re = re;
this.im = im;
}
public double realPart() { return re; }
public double imaginaryPart() { return im; }
public Complex add(Complex c) {
return new Complex(re + c.re, im + c.im);}
```

High Level Concurrency

- Manual synchronization (wait, notify) is error prone
- New high-level concurrency features were introduced:
  - Lock objects : more features than implicit object/class locks
  - Executors : Decouple managing threads from the tasks
  - Concurrent collections: internally synchronized
  - Atomic variables : support atomic methods

Lock objects

- Similar with implicit locks, a thread owns a lock
- Locks can be acquired and released in different scopes and in any order

void **lock**()	Acquires the lock.
void **lockInterruptibly**()	Acquires the lock unless the current thread is **interrupted**.
Condition newCondition()	Returns a new **Condition** instance that is bound to this Lock instance.
boolean **tryLock**()	**Acquires the lock only if it is free at the time of invocation.**
boolean **tryLock**(long time, **TimeUnit** unit)	Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.
void **unlock**()	Releases the lock.

- Must be released manually
- Conditions allow having multiple wait sets per object

ReentrantLock

- Implements Lock interface
- Owned by the last thread which locked but not unlocked it
- ReentrantLock(boolean fair) – when true grants access to longest waiting thread
- getOwner() : owner Thread or null
- getQueueLength() : how many threads are waiting for the lock
- isHeldByCurrentThread()
- isLocked()

Condition

- Replaces the use of wait/notify : can have multiple wait-sets per object
- Suspends a thread until notified by another thread that same condition state is true.
- Bound to a particular lock
- waiting for a condition provides is that it *atomically* releases the associated lock and suspends the current thread
- a thread that is woken up must reacquire the lock

Producer and consumer with locks

```
public class SharedRes {
  private Queue<Integer> queue= new LinkedList<>();
  private int limit;
  final Lock lock=new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  public void produce(Integer i)  {
    lock.lock();
    while(queue.size() == limit) {
      try {
        notFull.await();
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
    queue.add(i);
    notEmpty.signal();
    lock.unlock();
  }
```

```
  public Integer remove(){
    lock.lock();
    while(queue.isEmpty()) {
      try {
        notEmpty.await();
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
    Integer i = queue.remove();
    notFull.signal();
    lock.unlock();
    return i;
  }
}
```

Executor interface

- separating the task (runnable) from the worker that performs it(thread). Three interfaces
- An Executor object executes submitted Runnable tasks
  - Executor executor = *ExecutorImplementation*;
    executor.execute(new RunnableTask());
  - (new Thread(r)).start(); replaced with
  - executor.execute(r);

- ExecutorService interface
  - void shutdown() : no new tasks are accepted
  - <T> Future<T> submit(Callable<T> task) (Callable tasks can return a value)
  - Future<?> submit(Runnable task)
- ScheduledExecutorService
  - schedule: executes a Runnable or Callable after a fixed delay

Thread pools

- Usually, executor implementations use thread pools consisting of worker threads

- Creating executors:
  - java.util.concurrent.Executors utility class
  - newFixedThreadPool(int nThreads)
    - Has a specified number of threads running
  - newSingleThreadExecutor()
    - Single thread
    - tasks are guaranteed to be executed in the order in which they are added to the executor service
  - newCachedThreadPool()
    - Expandable thread pool

Producer consumer with Executors

```
public static void main(String[] args) {
ExecutorService executor=Executors.newFixedThreadPool(20);
BoundedBuffer bb=new BoundedBuffer();
        for(int i=0;i<10;i++){
            executor.execute(new Runnable(){
             public void run() {
                for(int j=0;j<100;j++){
                    bb.put(j);
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException ex) {
                       ex.printStackTrace();
                    }}}});}
```

Producer consumer with Executors

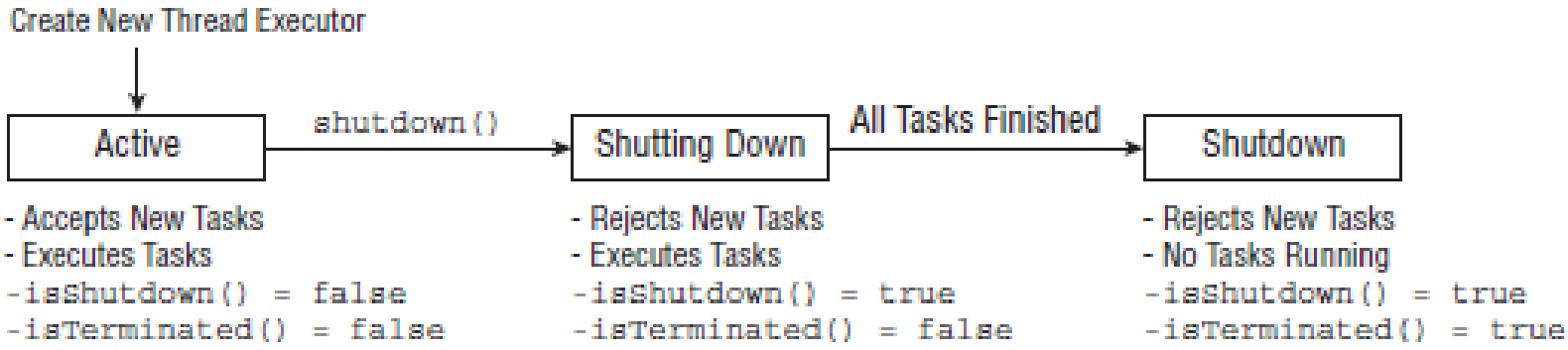
```
for(int i=0;i<10;i++){
            executor.execute(new Runnable(){
                @Override
                public void run() {
                for(int j=0;j<100;j++){
                    System.out.println(bb.take());
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException ex) {
                       ex.printStackTrace();
                    }
                }
                }
            });
        }
         executor.shutdown();
    }}
```

Shutting down

- without shutting down an executor runs forever

```
ExecutorService executor= Executors.newSingleThreadExecutor();
executor.submit(()-> System.out.println("hello"));
executor.shutdown();
```

- shutdown – no new tasks are accepted, previous tasks are still executed



- shutdownNow() – attempt to stop running task, return a list of tasks that were submitted but not started

Future objects

- Result of an asynchronous computation

boolean	**cancel**(boolean mayInterruptIfRunning)Attempts to cancel execution of this task.
V	**get**()Waits if necessary for the computation to complete, and then retrieves its result.
V	**get**(long timeout, **TimeUnit** unit)Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
boolean	**isCancelled**()Returns true if this task was cancelled before it completed normally.
boolean	**isDone**()Returns true if this task completed.

```
Future<Integer> res=executor.submit(()-> 3+5);
System.out.println(res.get());
```

- Wait at most a given time then retrieve

Limited wait

```
Future<Integer> res=executor.submit(()-> {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return 3+5;
    });
    executor.shutdown();
    System.out.println(res.get(50, TimeUnit.MILLISECONDS));
```

Exception in thread "main" java.util.concurrent.TimeoutException

Submit task collections

- `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks`
- Executes tasks, wait until all tasks are finished, return results as Collection of Future objects, same order as task collection
- `<T> T invokeAny(Collection<? extends Callable<T>> tasks`
- Executes tasks, wait until one task is finished, cancel any unfinished tasks

Submit task collections

```
List<Callable<Integer>> tasks=new LinkedList<>();
        for(int i =0;i<10;i++)
            tasks.add(()->
                new Random().nextInt(100));
        ExecutorService executor= Executors.newFixedThreadPool(5);
        List<Future<Integer>> results= executor.invokeAll(tasks);
        for (Future<Integer> x : results) {
            System.out.println(x.get());
        }
System.out.println("==========");
        System.out.println(executor.invokeAny(tasks));
        executor.shutdown();
```

Waiting for tasks to finish

- get on a Future object
- invokeAll/Any
- boolean awaitTermination(long timeout, TimeUnit unit)
  - Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
  - Returns true if this executor terminated and false if the timeout elapsed before termination

AwaitTermination

```
ExecutorService executor= Executors.newSingleThreadExecutor();
        for(int i=0;i<10000;i++)
            executor.submit(()->{
                long sum=0;
                for(int j=0;j<30000;j++)
                    sum+=j;
            });
        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.MILLISECONDS);
        if(executor.isTerminated())
            System.out.println("Tasks finished");
        else
            System.out.println("Tasks still running");
```

Concurrent Collections

- BlockingQueue : blocks or times out when adding to a full queue or retrieving from an empty one. Thread safe methods.

```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) { queue.put(produce()); }} catch (InterruptedException ex) {
... handle ...}   }
class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) { consume(queue.take()); } catch (InterruptedException ex) {
... handle ...}   }
```

BlockingQueue methods

- offer(E e) : inserts if possible, true if successful false if not
- poll(timeout) : retrieves and removes the head if not empty, else wait until timeout
- put (E e) : inserts, waiting if necessary
- remainingCapacity() : number of elements the queue can accept without blocking
- take() : retrieves and removes the head if not empty, else wait until an element becomes available
- Implementing classes: **ArrayBlockingQueue,** DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue

Other Concurrent Collections

Class	Interface
ConcurrentHashMap	ConcurrentMap
ConcurrentLinkDeque	Deque
ConcurrentLinkedQueue	Queue
ConcurrentSkipListMap	ConcurrentMap, SortedMap
ConcurrentSkipListSet	SortedSet
CopyOnWriteArrayList	List
CopyOnWriteArraySet	Set
LinkedBlockingDeque	BlockingDeque
LinkedBlockingQueue	BlockingQueue

- Skip collections – sorted
- CopyOnWrite – copy to a new structure at modifications, useful for concurrent iterations (iterators prior to modification iterate on original elements)

Synchronized collections

- Synchronized versions of **existing** non-concurrent collections. e.g. if you created the collection not knowing it will be used in a concurrent environment

Methods
synchronizedCollection(Collection<T> c)
synchronizedList(List<T> l)
synchronizedMap(Map<K,V> m)
synchronizedNavigableMap(NavigableMap <K,V> m)
synchronizedNavigableSet(NavigableSet<T> s)
synchronizedSet(Set<T> s)
synchronizedSortedMap(SortedMap<K,V> m)
synchronizedSortedSet(SortedSet<T> s)

Synchronized collections

- synchronized get and set but not iterators (should be used in a synchronized block)

```
List<Integer> list = Collections.synchronizedList(new ArrayList<>(List.of(3,5,1,4,6,7));
synchronized(list) {
for(int e: list)
System.out.print(e+" ");
}
```

CyclicBarrier

- coordination between related threads
- CyclicBarrier(n) – n number of threads to wait for. Once n threads call await on the barrier it is released

```
public class Cyclic {
    static final int NO_THREADS = 5;

    public static void main(String[] args) {
        ExecutorService service =
Executors.newFixedThreadPool(NO_THREADS);
        ComplexTask task = new ComplexTask();
        for (int i = 0; i < NO_THREADS; i++)
            service.submit(() -> task.performTask());
        service.shutdown();
    }}
```

```
class ComplexTask {
    CyclicBarrier b1 = new CyclicBarrier(3);
    CyclicBarrier b2 = new CyclicBarrier(3);

    public void performTask() {

        try {
            System.out.println("Task 1");
            b1.await();
            System.out.println("Task 2");
            b2.await();
            System.out.println("Task 3");
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```