JAVA MULTITHREADING & CONCURRENCY

- Rajesh Ananda Kumar
  - (rajesh_1290k@yahoo.com)

1. Introduction

- What is a Process?
- What is a Thread?
- Benefits of Threads
- Risks of Threads
- Real-time usage of threads
- Where are Threads used in Java?

WHAT IS A PROCESS?

- Process is a isolated, independently executing programs to which OS allocate resources such as memory, file handlers, security credentials, etc.
- Processes communicate with one another through mechanisms like sockets, signal handlers, shared memory, semaphores and files.

WHAT IS A THREAD?

- Threads are called lightweight processes
- Threads are spawned from Processes. So all the threads created by one particular process can share the process's resources (memory, file handlers, etc).
- Threads allow multiple program flows to coexist within the process.
- Even though threads share process-wide resources (memory, file handlers), but each thread has its own Program Counter, Stack and local variables.

What are the benefits of Threads?

BENEFITS OF THREADS

- Exploiting multiple processors
- Handling asynchronous events
- More responsive user interfaces

What are the risks of Threads?

RISKS OF THREADS

Threads in Java are like double-ended swords. Following are few risks;

- Safety Hazards
- Liveness Hazards
- Performance Hazards

RISKS OF THREADS – SAFETY HAZARDS

- Unsafe codes may result in race conditions.

```
public class SequenceGenerator {
  private int currentSequence = 0;

  public int getNextSequence() {
    return currentSequence++;
  }
}
```

- Proper synchronization should be done.

RISKS OF THREADS – LIVENESS HAZARDS

- A liveness failure occurs when an activity gets into a state such that it permanently unable to make forward progress. (Eg. Infinite loop)
- A liveness hazard scenario;

Thread A waits for a lock to be released by Thread B and vice versa. In this scenario, these programs will wait for ever.

RISKS OF THREADS – PERFORMANCE HAZARDS

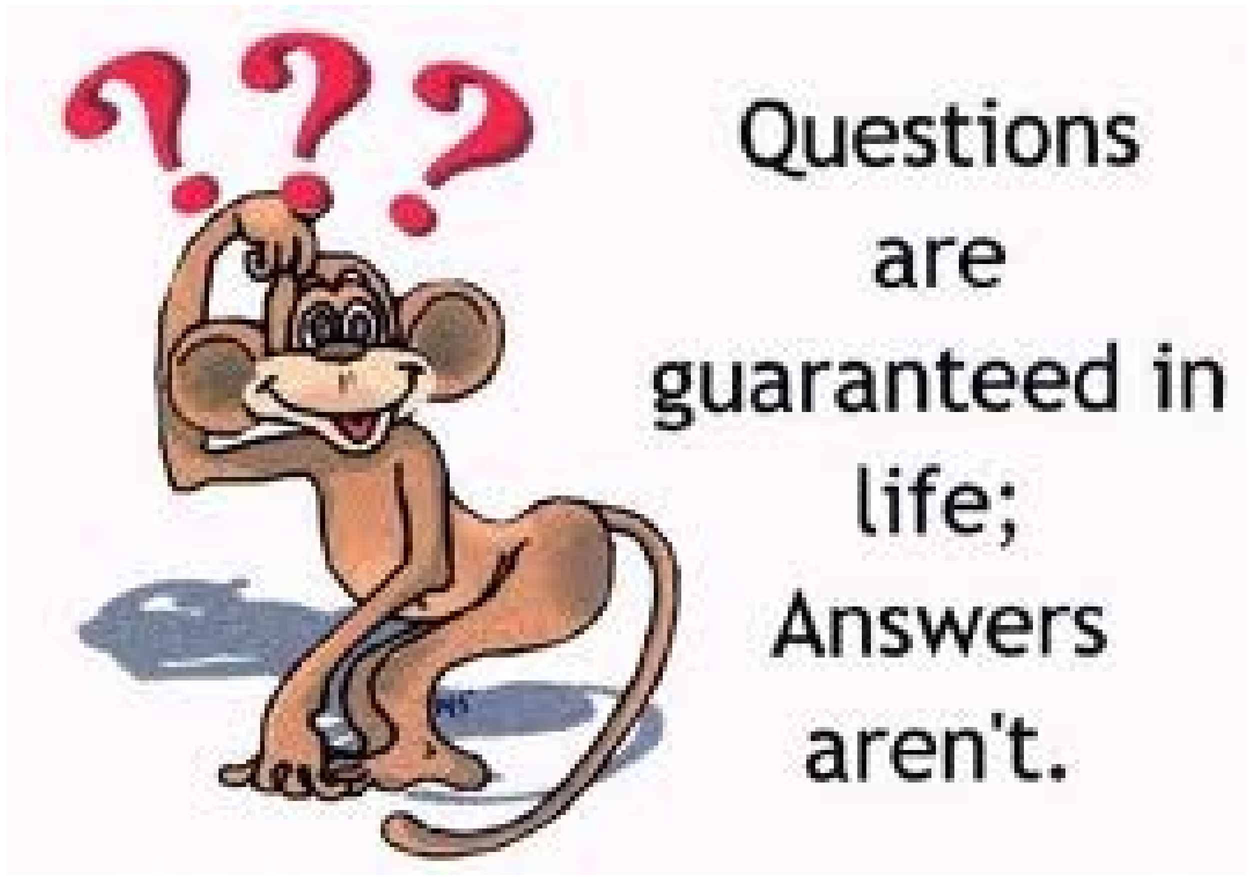
- Context Switches
- When threads share data, they must use synchronization which prevents compiler optimizations, flush or invalidate memory caches and create synchronization traffic on shared memory bus.

Where are Threads used in Java? Or how frequently are we using threads in our day-today work?

WHERE ARE THREADS USED IN JAVA?

- Threads are everywhere. Every Java application uses thread. Even a simple CUI based application that runs in a single class uses threads.
- When JVM starts, it creates threads for JVM housekeeping tasks (garbage collection, finalization) and a main thread for running “main” method.
- Examples; Timers, Servlets & JSPs, RMI, Swing & AWT

Q & A



2. Java Threading Basics

- Defining a Thread
- Instantiate a Thread
- Start a Threads
- Thread Life-cycle
- Thread Priorities
- Important methods from java.lang.Thread & java.lang.Object class;
  - sleep()
  - yield()
  - join()
  - wait()
  - notify() & notifyAll()
- Synchronization
  - Locks
  - When & how to Synchronize?
- Deadlocks
- Thread Interactions

DEFINING A THREAD

- Extending java.lang.Thread class

```
public class SampleThread extends Thread {
   public void run() {
      system.out.println("Running....");
   }
}
```

- Implementing java.lang.Runnable interface

```
public class SampleRunnable implements Runnable {
   public void run() {
      system.out.println("Running....");
   }
}
```

INSTANTIATE A THREAD

- We can use any of the above mentioned ways to define a thread. But can instantiate a thread only by using java.lang.Thread class.
- If we create a thread by extending Thread class;

```
SampleThread thread1 = new SampleThread();
```

- If we create a thread by implementing Runnable interface;

```
SampleRunnable runnable1 = new SampleRunnable();
Thread thread2 = new Thread(runnable1);
```

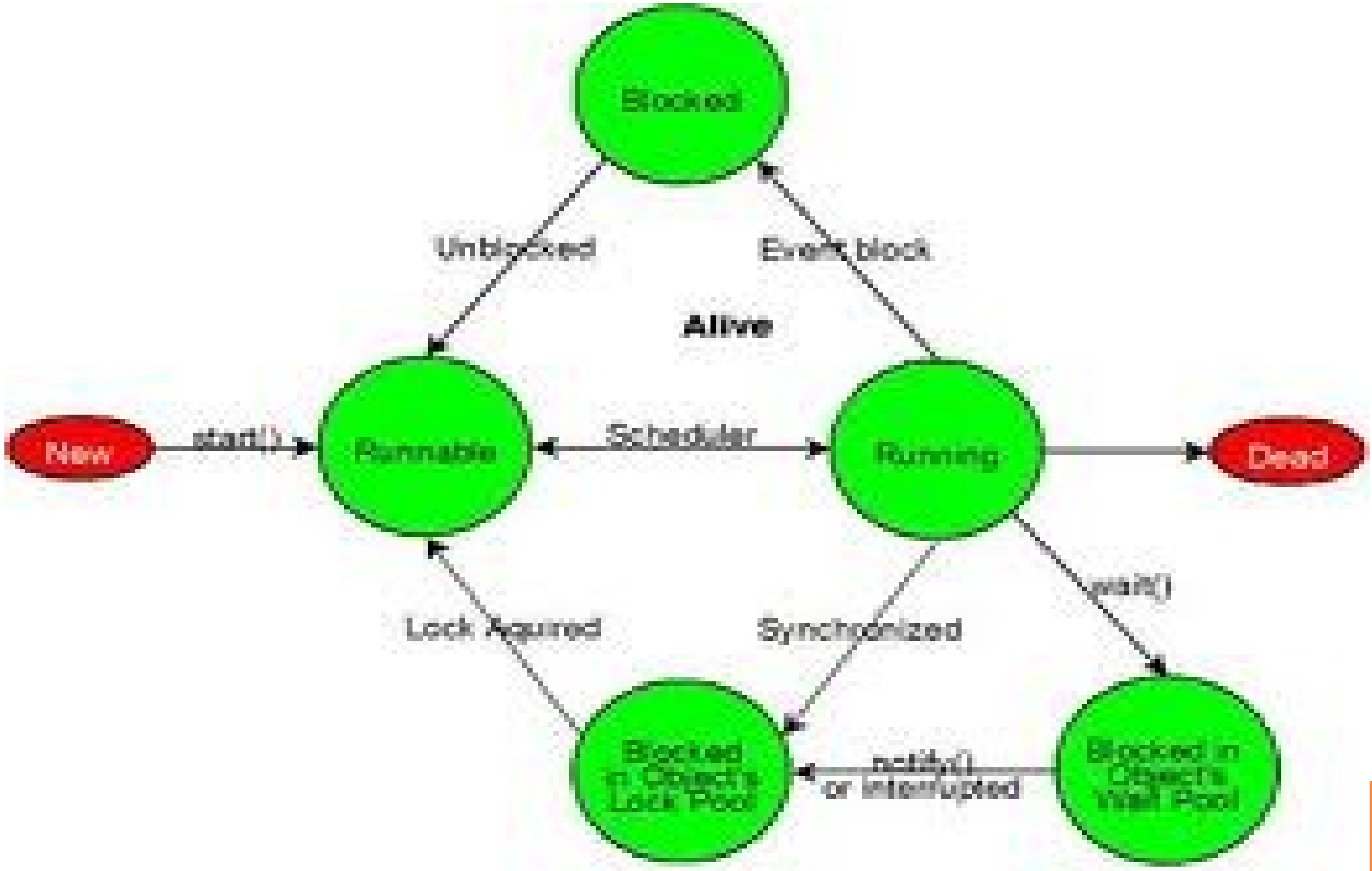
START A THREAD

- Use the java.lang.Thread class's start() method to start a thread.

```
public class TestThread {
  public static void main(String... args) {
    SampleThread t1 = new SampleThread();
    t1.start();
  }
}
```

- Once if we start a thread, it can never be started again.

THREAD LIFE-CYCLE



THREAD PRIORITIES

- In java Threads always run with some priorities, usually a number between 1 to 10.
- If a thread enters the runnable state, and it has higher priority than the current running thread, the lower-priority thread will be bumped back to runnable and highest-priority thread will be chosen to run.
- All priority threads being equal, a JVM implementation if the scheduler is free to do just about anything it likes.
- It is recommended not to depend on that behavior for correctness

THREAD PRIORITIES (CONT.)

- Use the following syntax to set priorities;

```
SampleThread thread1 = new SampleThread();
thread1.setPriority(8);
thread1.start();
```

- Default priority is 5.
- We can use the pre-defined constants like;

Thread.MIN_PRIORITY, Thread.NORM_PRIORITY and Thread.MAX_PRIORITY

IMPORTANT METHODS FROM JAVA.LANG.THREAD & JAVA.LANG.OBJECT - SLEEP()

- Since sleep is a static method in java.lang.Thread class, the call affects only the current thread in execution.
- Used for pausing the thread execution for a specific milliseconds.
- Throws InterruptedException when another thread is trying to interrupt a sleeping thread.
- Syntax;

```
try {
  Thread.sleep(60 * 1000); // sleep for 1 second
} catch(InterruptedException iex) { }
```

IMPORTANT METHODS FROM JAVA.LANG.THREAD & JAVA.LANG.OBJECT - YIELD()

- Since yield is a static method in java.lang.Thread class, the call affects only the current thread in execution.
- This method <u>is supposed</u> to make the current thread head back to runnable to allow other threads of the same priority to get their turn.
- There is no guarantee to do what it claims.

IMPORTANT METHODS FROM JAVA.LANG.THREAD & JAVA.LANG.OBJECT - JOIN()

- This is a non-static method.
- Syntax;

```
public class Test {
  public static void main(String... args) {
    SampleThread thread1 = new SampleThread();
    thread1.start();
    thread1.join();
    // Some more code here
  }
}
```

- The above code takes the main thread and attaches it to the end of 'thread1'. Main thread will be paused. 'thread1' will be finished and main thread will continue its execution.

IMPORTANT METHODS FROM JAVA.LANG.THREAD & JAVA.LANG.OBJECT – WAIT(), NOTIFY() & NOTIFYALL()

- Use non-static wait method to prevent or pause a thread from execution. Three overloaded versions of wait are available. Preferably use the method which takes a long as argument. This will make the thread wake-up after a particular time.
- 'notify' and 'notifyAll' are methods used to inform the waiting threads to stop waiting and move them from blocked to runnable state.
- All these 3 methods should be called only with in a synchronization context. A thread can't invoke these methods on an object unless it owns that object's lock.

IMPORTANT METHODS FROM JAVA.LANG.THREAD & JAVA.LANG.OBJECT – WAIT(), NOTIFY() & NOTIFYALL() (CONT.)

- When the wait method is called, the locks acquired by the objects are temporarily released.
- An InterruptedException is thrown when a waiting thread is interrupted by another thread.

SYNCHRONIZATION

- Synchronization can be done only for methods and blocks of code. Not for classes.
- Use synchronize keyword in places where we refer/use the mutable instance variables, as these mutable instance variables can be accessed by multiple threads simultaneously.
- Another use of synchronization is memory visibility.
- Synchronization ensures atomicity and visibility

SYNCHRONIZATION - LOCKS

- Every object in java has exactly one built-in lock by default. This is called intrinsic lock.
- When we synchronize a block of code with 'this' or when we synchronize a method, actually we synchronize it using intrinsic locks.
- We can use our own locks in place of intrinsic locks. We can declare an object as instance variable in a class and we can use block synchronization where the block is synchronized using this object.

SYNCHRONIZATION – WHEN & HOW?

- Synchronize all parts of code that access the mutable instance variables declared inside a class.
- Follow the same synchronization policy when modifying the code.
- Make granular synchronizations.
- Two ways to use synchronization.
- 1) Add synchronized keyword to method
- 2) Wrap a block of code with-in synchronized keyword

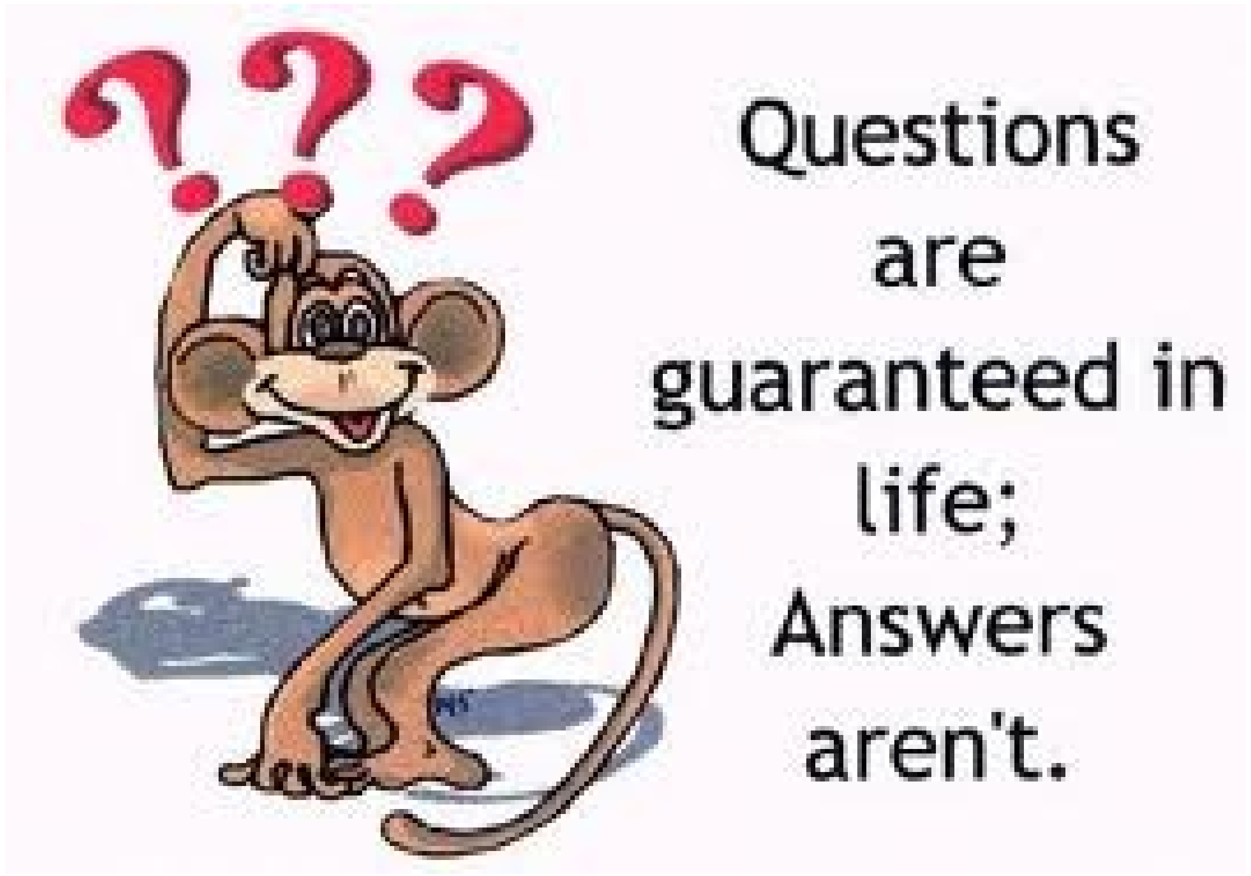
DEADLOCK

```
public class DeadlockDemo {
  private Object lockA = new Object();
  private Object lockB = new Object();

  public void read() {
    synchronized(lockA) {
      synchronized(lockB) {
        // Code for read operation
      }
    }
  }

  public void write() {
    synchronized(lockB) {
      synchronized(lockA) {
        // Code for write operation
      }
    }
  }
}
```

THREAD INTERACTIONS

```
public class Operator extends Thread {
  public void run() {
    while(true) {
      // Get data
      synchronized(this) {
        // Calculate input required for Machine
        notify();
      }
    }
  }
}

public class Machine extends Thread {
  Operator operator; // Assume we have a constructor
  public void run() {
    while(true) {
      synchronized(operator) {
        try { operator.wait(); }
        catch (InterruptedException ex) { }
        // Get the generated input and process it
      }
    }
  }
}
```

Q & A



3. Advanced Java thread concepts

- Immutable objects
- Thread-safety
  - What is Thread-safety?
  - Tops for creating thread-safe classes?
- Volatile variables
- Synchronized collections
- Synchronized collections - Problems

IMMUTABLE OBJECTS

- An object is immutable if;
  - Its state cannot be modified after construction
  - All its fields are final
  - It is properly constructed
- Example is String.
- Create a custom immutable class.

THREAD-SAFETY – WHAT IS THE MEANING?

- A class is tread-safe when it continues to behave correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

THREAD-SAFETY – TIPS

- Stateless objects are thread-safe.
- State-full objects with ALL immutable instance variable are thread-safe.
- Access to immutable instance variables don't have to be synchronized.
- Synchronize atomic operations (on mutable instance variables). If not, there will be race condition. For example;

THREAD-SAFETY – TIPS (CONT.)

```
public class CounterServlet implements Servlet {
  private long count = 0;

  public void service(ServletRequest request,
ServletResponse response) {
    // Some code here
    ++count;
    // Some code here
  }
}
```

- To fix the above scenario use AutomicLong instead of 'long'.
- For mutable state variables that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held. In this case, we say that the variable is guarded by the same lock.

VOLATILE VARIABLES

- Volatile variables are not cached in registers or in caches where they are hidden from other processors, so read of a volatile variable always returns the most recent write by any thread.
- Accessing volatile variables perform no locking and so cannot cause the executing thread to block. Volatile variables are called light-weight synchronization.
- Locking can guarantee both visibility and atomicity; but volatile variables can only guarantee visibility.

VOLATILE VARIABLE (CONT.)

Use volatile variables only when;

- Writes to the variable do not depend on its current value.
- Variable does not participate in invariants with other state variables.
- Locking not required for any other reason while the variable is being accessed.

SYNCHRONIZED COLLECTIONS

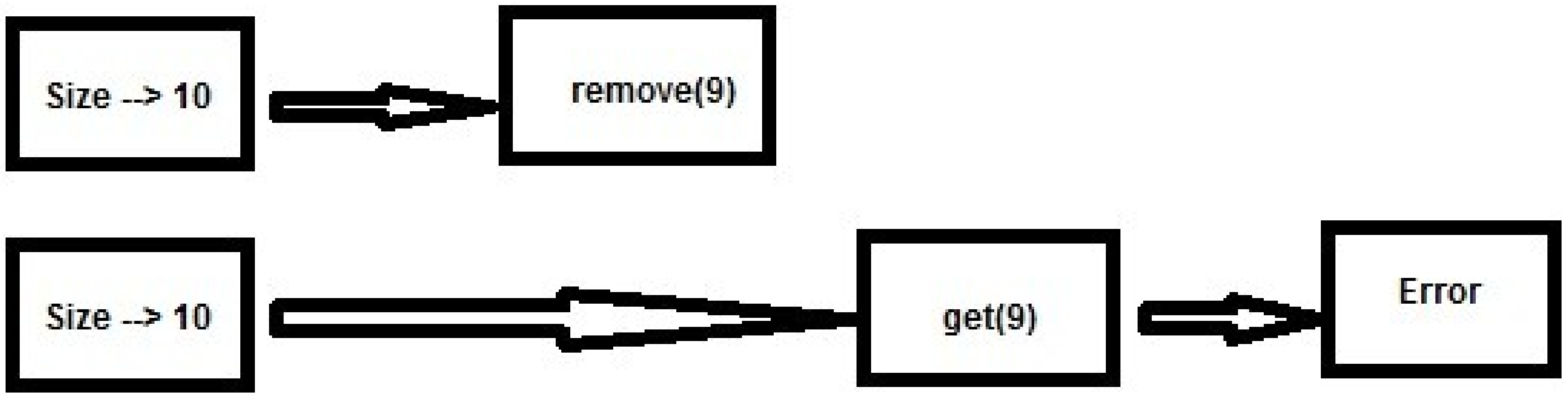
- Vector and Hashtable are synchronized classed from JDK 1.2
- Collections.synchronizedXxx helps to create synchronized wrapper classes. These classes achieve thread-safety by encapsulating their state and synchronizing all public methods so that only one thread can access the collection at a time.

SYNCHRONIZED COLLECTIONS - PROBLEMS

- Compound actions are still technically thread-safe even without client-side locking, but they may not behave as you might expect when other threads can concurrently modify the collection.

```
public Object getLast(Vector list) {
  int lastIndex = list.size() - 1;
  return list.get(lastIndex);
}

public void deleteLast(Vector list) {
  int lastIndex = list.size() - 1;
  list.remove(lastIndex);
}
```

SYNCHRONIZED COLLECTION – PROBLEMS (CONT.)



SYNCHRONIZED COLLECTION – PROBLEMS (CONT.)

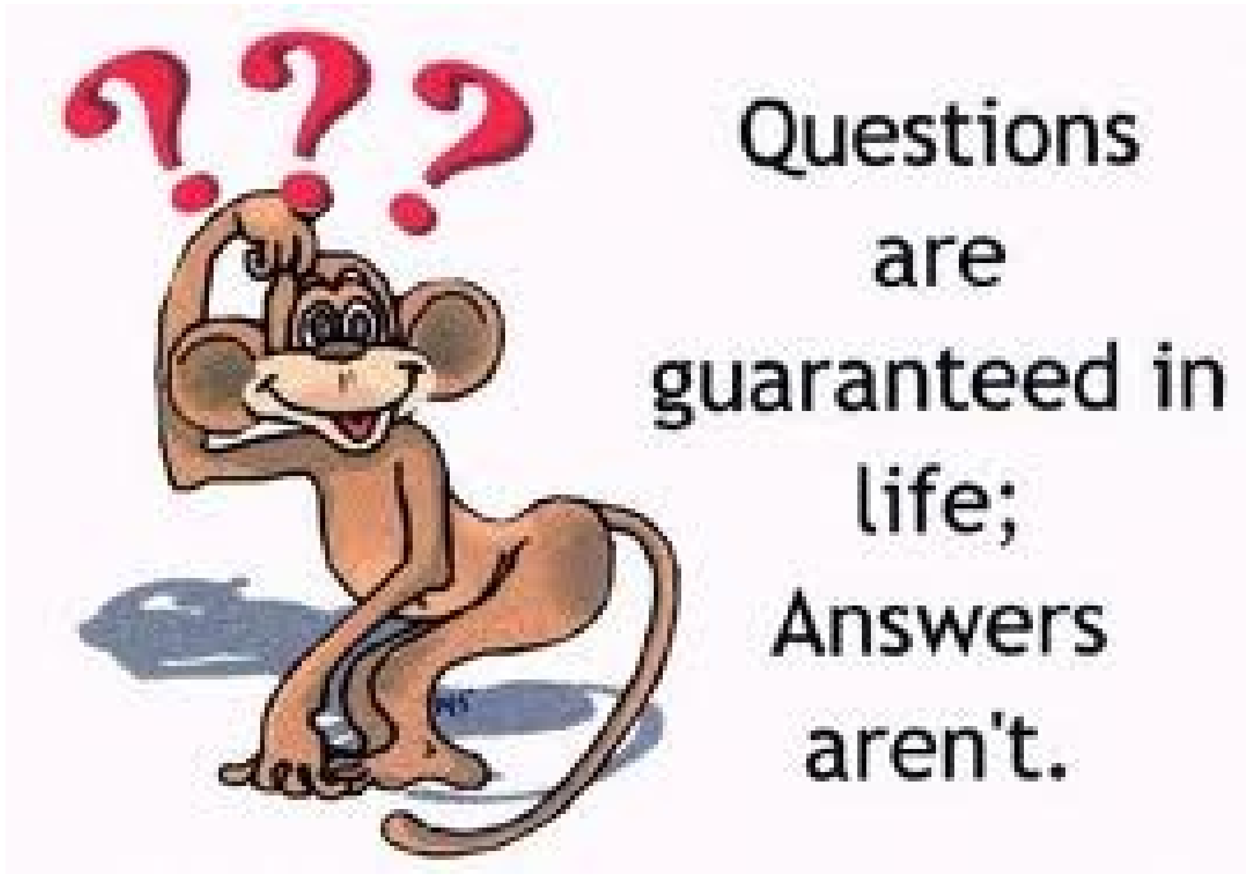
- To fix this issue the code inside the methods has to be synchronized using the input argument 'list'.

```
public Object getLast(Vector list) {
  synchronized(list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
  }
}

public void deleteLast(Vector list) {
  synchronized(list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
  }
}
```

SYNCHRONIZED COLLECTIONS – PROBLEMS (CONT.)

- The iterators returned by the synchronized collections are fail-fast iterators.
- While iteration, if another thread modifies the list, then the iteration would be stopped by throwing the exception ConcurrentModificationException.
- Ways to avoid; clone the collection and send it for iteration or wrap the collection with UnmodifiableXxx class or use concurrent collections.

Q & A



4. Java Concurrency

Use of java.util.concurrent.lock.Lock
Use of java.util.concurrent.atomic package
Concurrent Collections
Synchronizers
- Latches
- FutureTask
- Semaphores & Barriers

Executor Framework
Thread Pools
Executor Lifecycle
Callable & Future
Task Cancellation
Cancellation via Future
ExecutorService shutdown

USE OF JAVA.UTIL.CONCURRENT.LOCK.LOCK

- An alternative synchronization mechanism introduced in JDK 1.5.
- This was developed to solve the problems that we had with traditional synchronization.
- A commonly used Lock implementation is ReenterantLock.
- In general a reentrant lock mean, there is an acquisition count associated with the lock, and if a thread that holds the lock acquires it again, the acquisition count is incremented and the lock then needs to be released twice to truly release the lock.

USE OF JAVA.UTIL.CONCURRENT.LOCK.LOCK (CONT.)

```
Lock lock = new ReentrantLock();

lock.lock();
try {
  // update object state
}
finally {
  lock.unlock();
}
```

- Many benchmarks have proven that, Reenterant locks perform much better than synchronization.

USE OF JAVA.UTIL.CONCURRENT.LOCK.LOCK (CONT.)

- This might tempt us to stop using synchronization. But experts say that the new concurrency Lock is only for advanced users for 2 reasons;
- 1) It is easy to forget to use a finally block to release the lock, to the great detriment of your program.
- 2) When the JVM manages lock acquisition and release using synchronization, the JVM is able to include locking information when generating thread dumps. The Lock classes are just ordinary classes, and the JVM does not (yet) have any knowledge of which Lock objects are owned by specific threads.

USE OF JAVA.UTIL.CONCURRENT.ATOMIC PACKAGE

- A small toolkit of classes that support lock-free thread-safe programming on single variables.
- Few classes under this package are AtomicInteger, AtomicLong, AtomicReference, etc.
- Classes like AtomicInteger can be used in situations like atomically incremented counter.

USE OF JAVA.UTIL.CONCURRENT.ATOMIC PACKAGE (CONT.)

- To understand the atomic package correctly, we should understand 2 concepts;
- CAS: Compare And Swap. This idea is used in processors also. A CAS operation includes three operands -- a memory location (V), the expected old value (A), and a new value (B). The processor will atomically update the location to the new value if the value that is there matches the expected old value, otherwise it will do nothing. In either case, it returns the value that was at that location prior to the CAS instruction. CAS effectively says "I think location V should have the value A; if it does, put B in it, otherwise, don't change it but tell me what value is there now."

USE OF JAVA.UTIL.CONCURRENCY.ATOMIC PACKAGE (CONT.)

Lock-Free: Concurrent algorithms based on CAS are called lock-free, because threads do not ever have to wait for a lock (sometimes called a mutex or critical section, depending on the terminology of your threading platform). Either the CAS operation succeeds or it doesn't, but in either case, it completes in a predictable amount of time. If the CAS fails, the caller can retry the CAS operation or take other action as it sees fit.

USE OF JAVA.UTIL.CONCURRENT.ATOMIC PACKAGE (CONT.)

```
package java.util.concurrent.atomic;

public class AtomicInteger {
  private int value;

  public final int get() {
    Return value;
  }

  public final int incrementAndGet() {
    for (;;) {
      int current = get();
      int next = current + 1;
      if (compareAndSet(current, next))
        return next;
    }
  }
}
```

CONCURRENT COLLECTIONS

- These collections are designed for concurrent access from multiple threads.
- Interface ConcurrentMap is introduced for adding support for compound actions like put-if-absent, replace and conditional remove.
- Interface BlockingQueue is useful for producer-consumer design scenarios.
- ConcurrentHashMap is replacement for synchronized hash-based map.
- CopyOnWriteArrayList is replacement for synchronized list.

SYNCHRONIZERS

- A Synchronizer is any object that co-ordinates the control flow of threads based on its state.
- Blocking queues can act as synchronizers.
- Other synchronizers are Latches, Semaphores & Barriers.

SYNCHRONIZER - LATCHES

- A latch is a synchronizer that can delay the process of threads until it reaches its terminal state.
- Once the latch reaches the terminal state, it cannot change the state again. So it remains open forever.
- Implementation call is CountDownLatch. Uses await() and countDown() methods for controlling the thread flow.

SYNCHRONIZERS – LATCHES (CONT.)

```
Class TaskTimeCalculator {
  public long timeTaks(int noOfThreads, final Runnable task) {
    CountDownLatch startGate = new CountDownLatch(1);
    CountDownLatch endGate = new CountDownLatch(noOfThreads);

    for(int i = 0; i < noOfThreads; i++) {
      Thread t = new Thread() {
        public void run() {
          startGate.await();
          try { task.run(); }
          finally { endGate.countDown(); }
        }
      };
      t.start();
    } // End of for

    long start = System.nanoTime();
    startGate.countDown();
    endGate.await();
    long end = System.nanoTime();
    return end - start;
  }
}
```

SYNCHRONIZERS - FUTURETASK

- FutureTask acts like a latch.
- It implements Future.
- Computation represented by FutureTask is implemented with a Callable interface.
- Behavior of Future.get() depends on the state of the task. If task is not completed, get() method waits or blocks till the task is completed. Once completed, it returns the result or throws an ExecutionException.

SYNCHRONIZERS – FUTURETASK (CONT.)

```
class PreLoader {
  private final FutureTask<ProductInfo> future =
    new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
      public ProductInfo call() throws DataLoadException {
        return loadProductInfo();
      }
    });

  private final Thread thread = new Thread(future);

  public void start() { thread.start(); }

  public ProductInfo get()
    throws DataLoadException, InterruptedException {
    try {
      return future.get();
    } catch(ExecutionException eex) {
      Throwable cause = e.getCause();
      // other exception handling code here
    }
  }
}
```

SYNCHRONIZERS – SEMAPHORES & BARRIERS

- Counting Semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time. This can be used for implementing resource pooling. Methods used; acquire() & release().
- While Latches are for waiting fir events, Barriers are for waiting for other threads. All the threads must come together at a barrier point at the same time in order to proceed.

EXECUTOR FRAMEWORK

- Tasks are logical units of work, and threads are a mechanism by which tasks can run asynchronously.
- Java.util.concurrent provides a flexible thread pool implementation as part of the Executor framework.
- The primary abstraction for task execution in the java class libraries is not Thread, but Executor.

EXECUTOR FRAMEWORK (CONT.)

```
public interface Executor {
  void execute(Runnable command);
}
```

- This simple interface forms the basic for flexible & powerful framework for asynchronous task execution that support wide variety of task execution policy.
- Executors help us decoupling task submission from task execution.

THREAD POOLS

- Thread pool manages a homogenous pool of worker threads.
- Different thread pool implementation are;

1) FixedThreadPool

2) CachedThreadPool

3) SingleThreadExecutor

4) ScheduledThreadPool

- These pools are created using Executors class.

EXECUTOR LIFECYCLE

- Since Executors provide a service to applications, they should be able to shut down properly.

```
public interface ExecutorService
       extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(
     long timeout, TimeUnit unit)
     throws InterruptedException;

  // Other methods
}
```

EXECUTOR LIFECYCLE - EXAMPLE

```
class LifeCycleWebServer {
  private final ExecutorSerive exec =
                        Executors.newFixedThreadPool(100);

  public void start() throws IOException {
    ServerSocket socket = new ServerSocket(80);

    while(!exec.isShutdown()) {
      final Socket conn = socket.accept();
      exec.execute(new Runnable() {
        public void run() { handleRequest(conn); }
      });
    }
  }

  public void stop() { exec.shutdown(); }

  void handleRequest(Socket connection) {
    Request req = readRequest(connection);
    if(isShutdownRequest(req)
      stop();
    else
      dispatchRequest(req);
  }
}
```

CALLABLE & FUTURE

- In Runnable interface, run cannot return a value or throw checked exception. Callable interface solves this problem.
- Callable expects a main entry point called 'call' method and can return a value. Also it can throw an Exception.
- Future represents the lifecycle of a task and provides methods to test wherher a task is completed or been canceled, retrieve its result and cancel the task.

CALLABLE & FUTURE (CONT.)

```
public interface Callable<V> {
  V call() throws Exception;
}

public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws …;
  V get(long timeout, TimeUnti unit) throws …;
}
```

TASK CANCELLATION

- An activity is cancellable if external code can move it to completion before its normal completion.
- There are different ways to achieve this.

TASK CANCELLATION – MANUAL

```
public class PrimeGenerator implements Runnable {
  private final List<BigInteger> primes =
       new ArrayList<BigInteger>();

  private volatile boolean cancelled = false;

  public void run() {
    BigInteger p = BigInteger.ONE;
    while(!cancelled) {
      p = p.nextProbablePrime();
      synchronized(this) {
        primes.add(p);
      }
    }
  }

  public void cancel() { cancelled = true; }

  public synchronized List<BigInteger> get() {
    return new ArrayList<BigInteger>(primes);
  }
}
```

TASK CANCELLATION – THREAD INTERRUPTION

- The problem with previous approch is, it cancels all the executing threads. We cannot cancel a specific instance.
- We can use the following methods in Thread class instead.

```
public class Thread {
  public void interrupt() { .. }
  public boolean isInterrupted() { .. }
  public static boolean interrupted() { .. }
}
```

TASK CANCELLATION – THREAD INTERRUPTION (CONT.)

```
public class PrimeGenerator implements Runnable {
  private final List<BigInteger> primes =
       new ArrayList<BigInteger>();

  private volatile boolean cancelled = false;

  public void run() {
    BigInteger p = BigInteger.ONE;
    While(!Thread.currentThread().isInterrupted()) {
      p = p.nextProbablePrime();
      synchronized(this) {
        primes.add(p);
      }
    }
  }

  public void cancel() { interrupt(); }

  public synchronized List<BigInteger> get() {
    return new ArrayList<BigInteger>(primes);
  }
}
```

CANCELLATION VIA FUTURE

```
public void timedRun(Runnable r,
           long timeout,
           TimeUnit unit) throws InterruptedException {

  Future<?> task = taskExec.submit(r);

  try {
    task.get(timeout, unit);
  } catch(TimeoutException tex) {
    // task will be cancelled below
  } catch(ExecutionException eex) {
    // Process eex and re-throw if needed
  } finally {
    if(!task.isCancelled() && !task.isDone()) {
      task.cancel(true);
    }
  }
}
```

EXECUTORSERVICE SHUTDOWN

- If Executors are not shutdown properly, it won't allow the JVM process to close.
- Two ways to shutdown;

1) Call shutdown()

2) Call shutdownNow()

That’s all flocks! ☺

Thank you!! Please send me your valuable feedbacks.