Concurrency in Java

Allan Huang@Patentcloud.com

Processes vs. Threads

- A Process runs independently and isolated of other processes. It cannot directly access shared data in other processes.
- A Thread is a so called Lightweight Process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache – Thread Local Storage.
- A Java application runs by default in One Process.
- Inter-process communication (IPC) is a mechanism that allows the exchange of data between processes, e.g. Java applications.

Thread Safety

- If it behaves 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.
- Use proper locking mechanism when modifying shared data.
- Locking establishes the orderings needed to satisfy the Java Memory Model (JSR-133) and guarantee the visibility of changes to other threads.

Synchronized, final and volatile modifiers

- *synchronized* modifier
  - A block of code that is marked as synchronized in Java tells the JVM: "only let one thread in here at a time".
- *final* modifier
  - Values of final fields, including objects inside collections referred to by a final reference, can be safely read without synchronization.
  - Store a reference to an object in a final field only makes the reference immutable, not the actual object.
- *volatile* modifier
  - It's used to mark a field and indicate that changes to that field must be seen by all subsequent reads by other threads, regardless of synchronization.
  - It's not suitable for cases where we want to Read-Update-Write as an atomic operation.
  - *synchronization* modifier supports mutual exclusion and visibility. In contrast, the *volatile* modifier only supports visibility.

Thread Synchronization

```
public class DemoClass
{
    public synchronized void demoMethod(){}
}

or

public class DemoClass
{
    public void demoMethod(){
        synchronized (this)
        {
            //other thread safe code
        }
    }
}

or

public class DemoClass
{
    private final Object lock = new Object();
    public void demoMethod(){
        synchronized (lock)
        {
            //other thread safe code
        }
    }
}
```

Object Level Locking

```
public class DemoClass
{
    public synchronized static void demoMethod(){}
}

or

public class DemoClass
{
    public void demoMethod(){
        synchronized (DemoClass.class)
        {
            //other thread safe code
        }
    }
}

or

public class DemoClass
{
    private final static Object lock = new Object();
    public void demoMethod(){
        synchronized (lock)
        {
            //other thread safe code
        }
    }
}
```

Class Level Locking

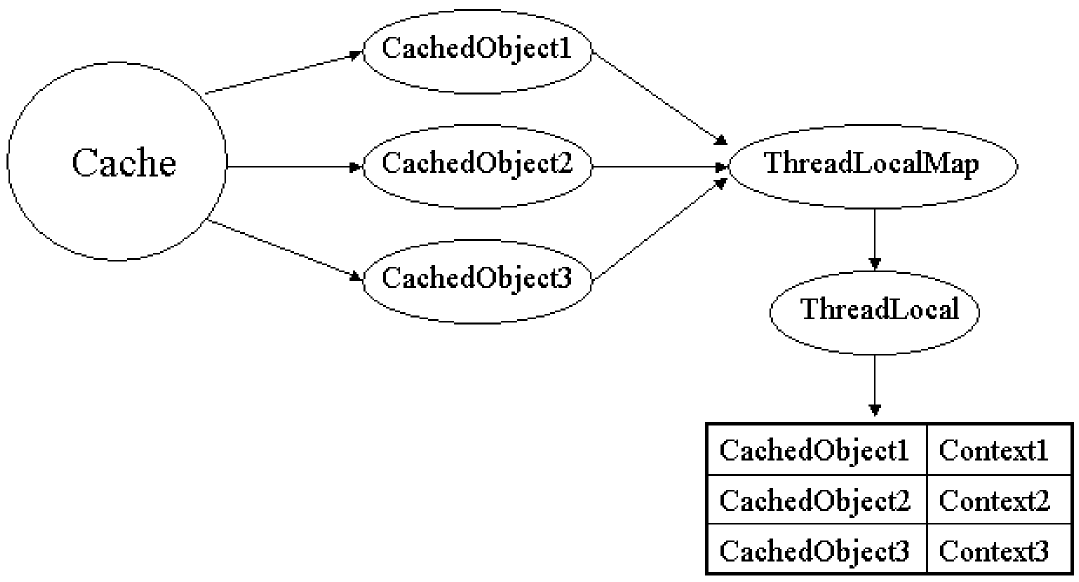
Atomic Action

- Compare-and-swap (CAS)
  - A hardware instruction is much more Lightweight than Java's monitor-based synchronization mechanism and is used to implement some highly scalable concurrent classes.
- Atomic classes
  - Support atomic compound actions on a single value in a lock-free manner similar to volatile.

```
public class Counter {

    private AtomicInteger value = new AtomicInteger();

    public int next() {
        return this.value.incrementAndGet();
    }

}
```

Thread Local

- Java Thread Local Storage – TLS
- It's typically private static fields in classes that wish to associate state with a thread, e.g. a current user or transaction.
- The best way to avoid memory leak is to call remove() method or set(null) on the ThreadLocal instance once the job is done.

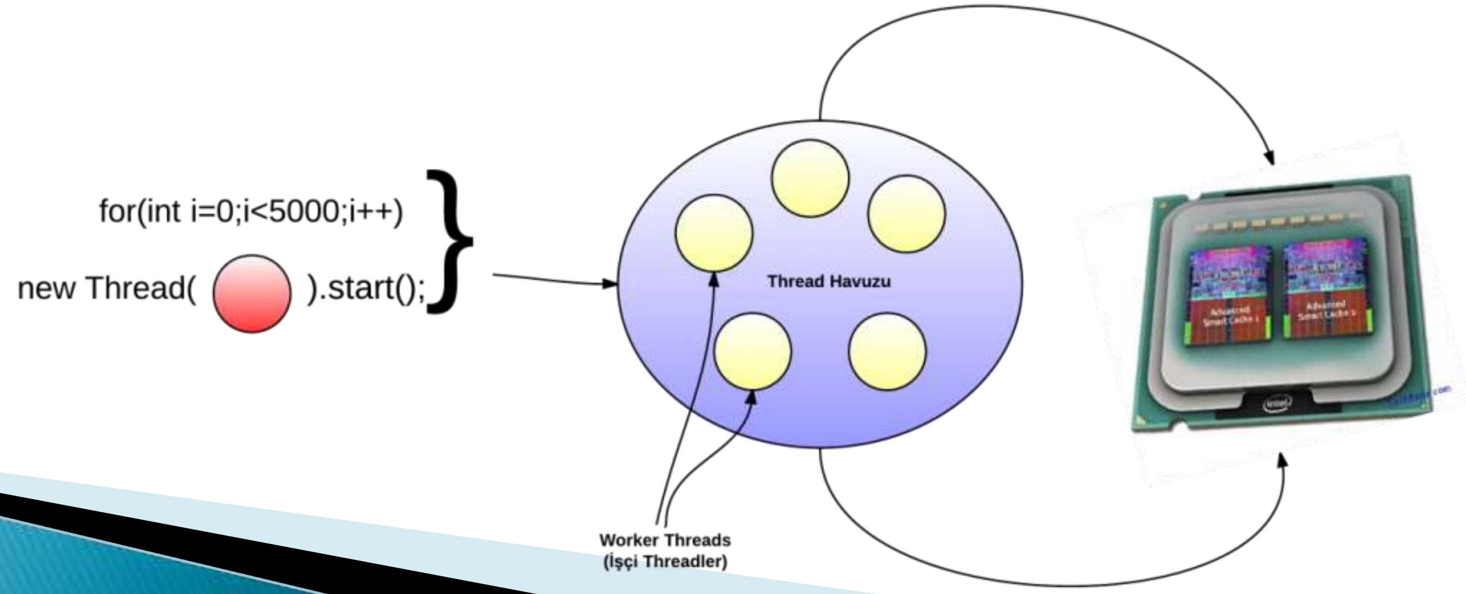


Thread Local Example

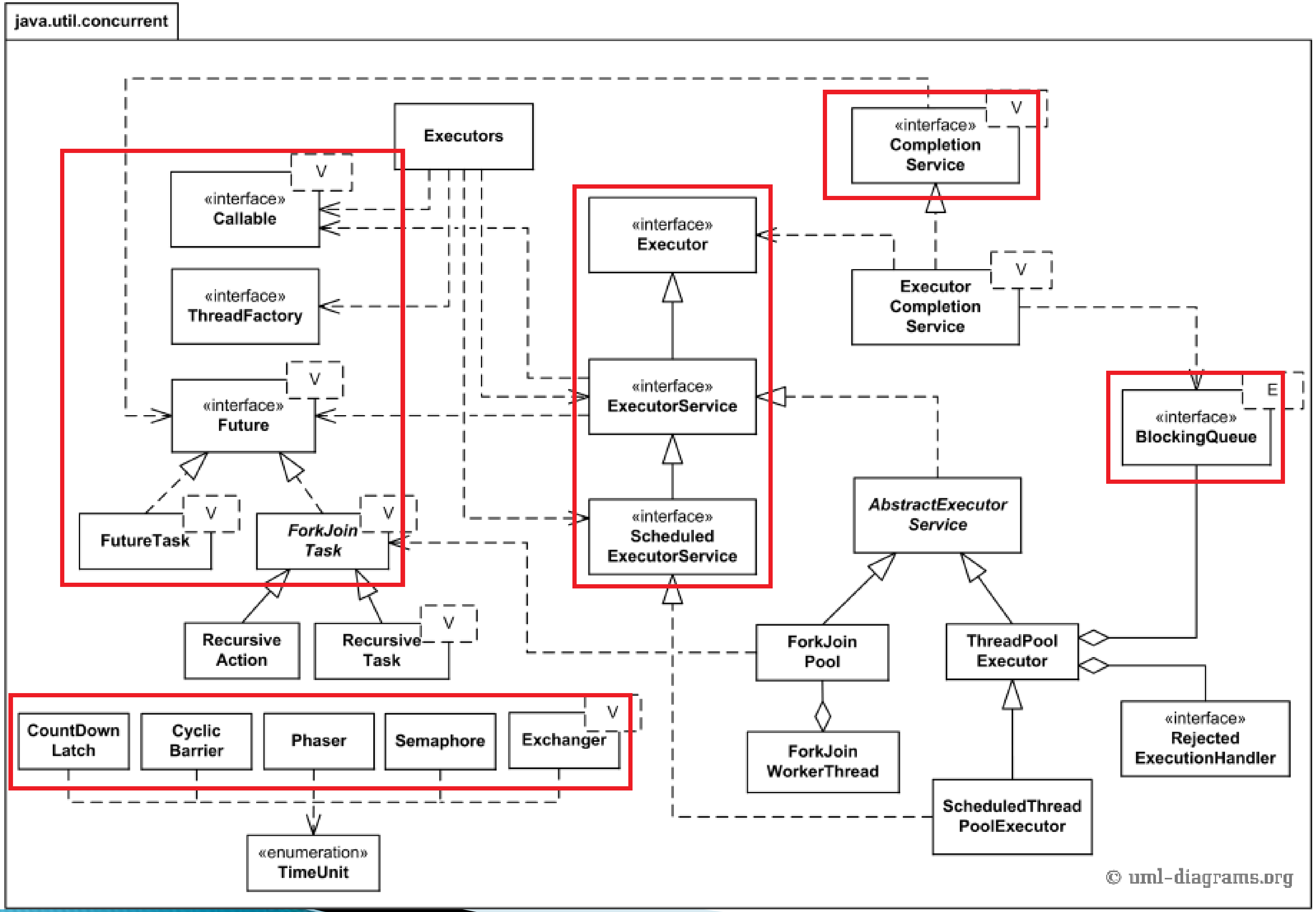
```java
public class ConnectionManager {

  ① private static ThreadLocal<Connection> currentConnection = new ThreadLocal<Connection>();

    public final Connection getConnection() throws SQLException {
      ② Connection conn = currentConnection.get();

        if (conn == null || conn.isClosed()) {
            String connectionString = "jdbc:sqlserver://localhost:1433;databaseName=test;"
                    + "lockTimeout=3000;loginTimeout=5;responseBuffering=adaptive;selectMethod=cursor";
            conn = DriverManager.getConnection(connectionString);

          ③ currentConnection.set(conn);
        }

        return conn;
    }

    public static void closeQuietly(Connection conn) {
        try {
            if (conn != null && !conn.isClosed()) {
                conn.close();
            }
        } catch (SQLException e) {
            // quiet
        } finally {
          ④ currentConnection.set(null);
            currentConnection.remove();
        }
    }
}
```

Unbounded Thread Creation

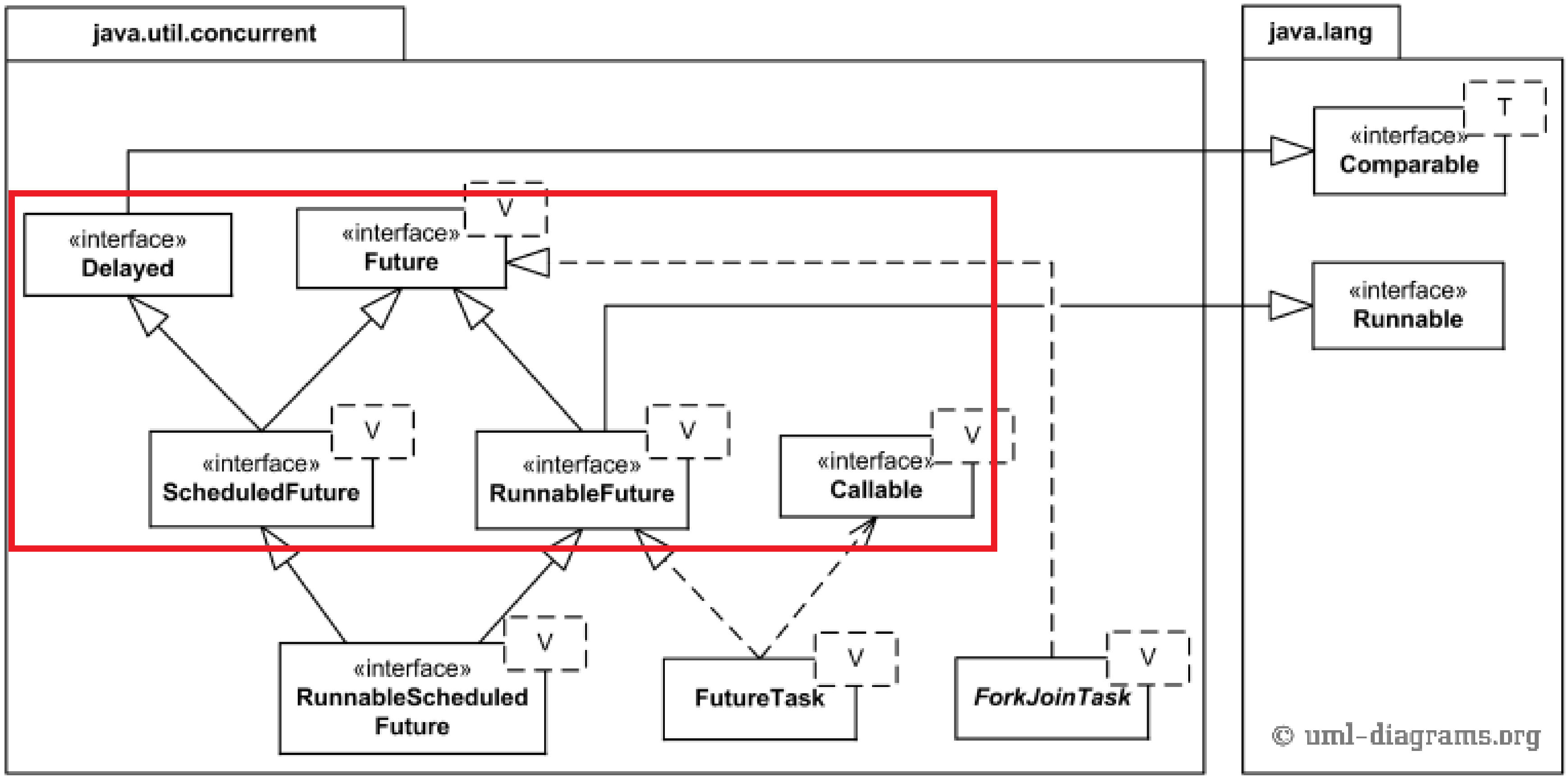
- Thread-per-task approach
  - When a large number of threads may be created, each created thread requires memory, and too many threads may exhaust the available memory, forcing the application to terminate.
- Executor Framework
  - Use a flexible Thread Pool implementation, in which a fixed or certain number of threads would service incoming tasks.



Executors and Thread Pool Managers

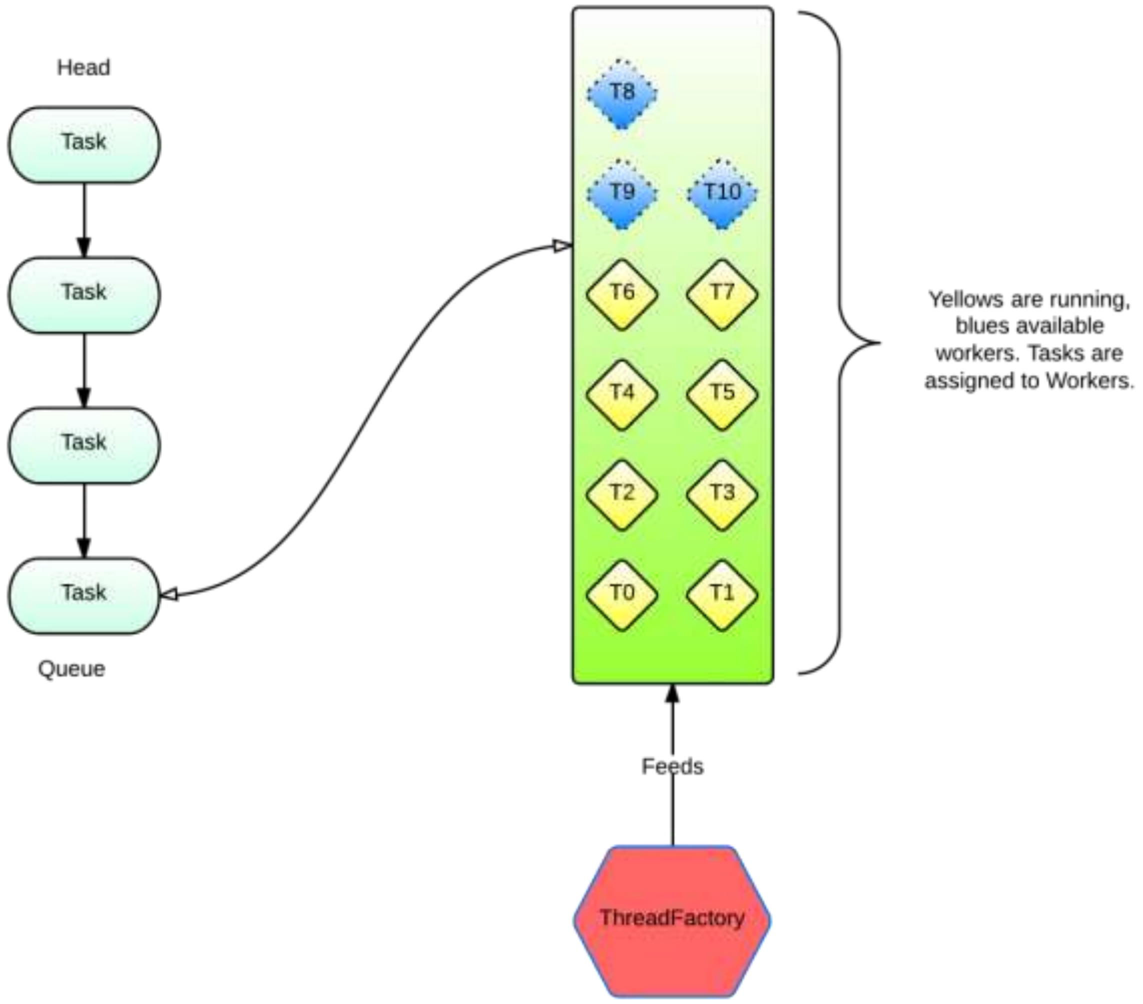


Asynchronous Results / Futures



Pooled-based Executor Service

- Executors factory methods
  - *newSingleThreadExecutor*
  - *newFixedThreadPool*
  - *newCachedThreadPool*
  - *newSingleThreadScheduledExecutor*
  - *newScheduledThreadPool*
- Runnable, Callable and Future
  - A Callable is like the familiar Runnable but can return a result and throw an exception.
  - A Future is a marker representing a result that will be available at some point in the future.



Executor Service Example

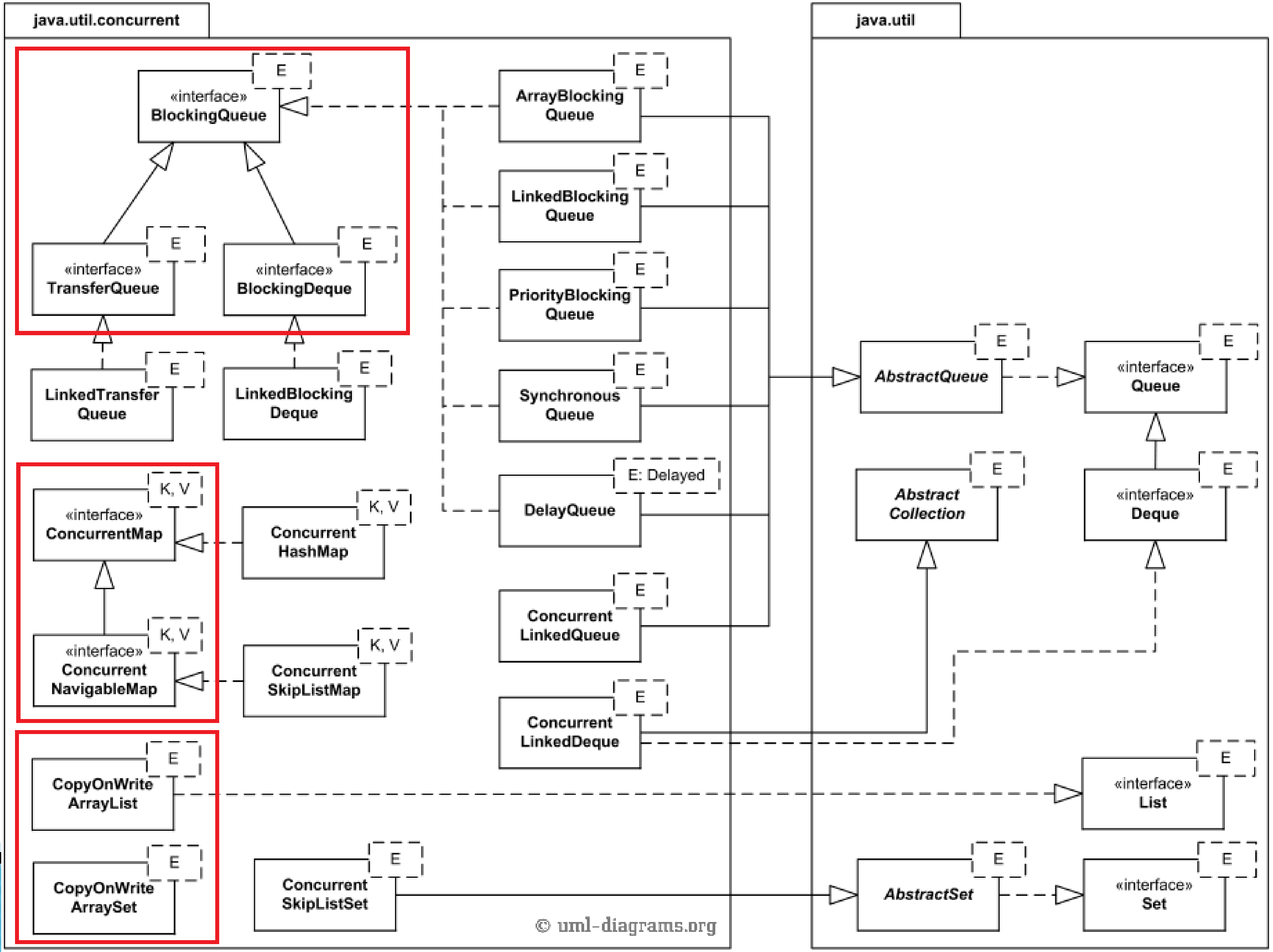
```java
public class Server {

    static final int processors = Runtime.getRuntime().availableProcessors() * 2;

    static Executor pool = Executors.newFixedThreadPool(processors);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(9000);

        while (true) {
            final Socket s = socket.accept();

            Runnable r = new Runnable() {
                @Override
                public void run() {
                    doWork(s);
                }
            };
            pool.execute(r);
        }
    }

    static void doWork(Socket s) {
        // to do somethings
    }
}
```

```java
public static void main(String[] args) throws Exception {
    Runnable runnabledelayedTask = new Runnable() {
        @Override
        public void run() {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is Running Delayed Task");
        }
    };

    Callable<String> callabledelayedTask = new Callable<String>() {
        @Override
        public String call() throws Exception {
            return "GoodBye! See you at another invocation...";
        }
    };

    ScheduledExecutorService scheduledPool = Executors.newScheduledThreadPool(4);
    scheduledPool.scheduleWithFixedDelay(runnabledelayedTask, 1, 1, TimeUnit.SECONDS);

    ScheduledFuture<String> future = scheduledPool.schedule(callabledelayedTask,
            4, TimeUnit.SECONDS);
    String value = future.get();

    System.out.println("Callable returned" + value);

    scheduledPool.shutdown();
}
```

Fixed Thread Pool

Scheduled Thread Pool

Concurrent Collection Family



Concurrent Collection

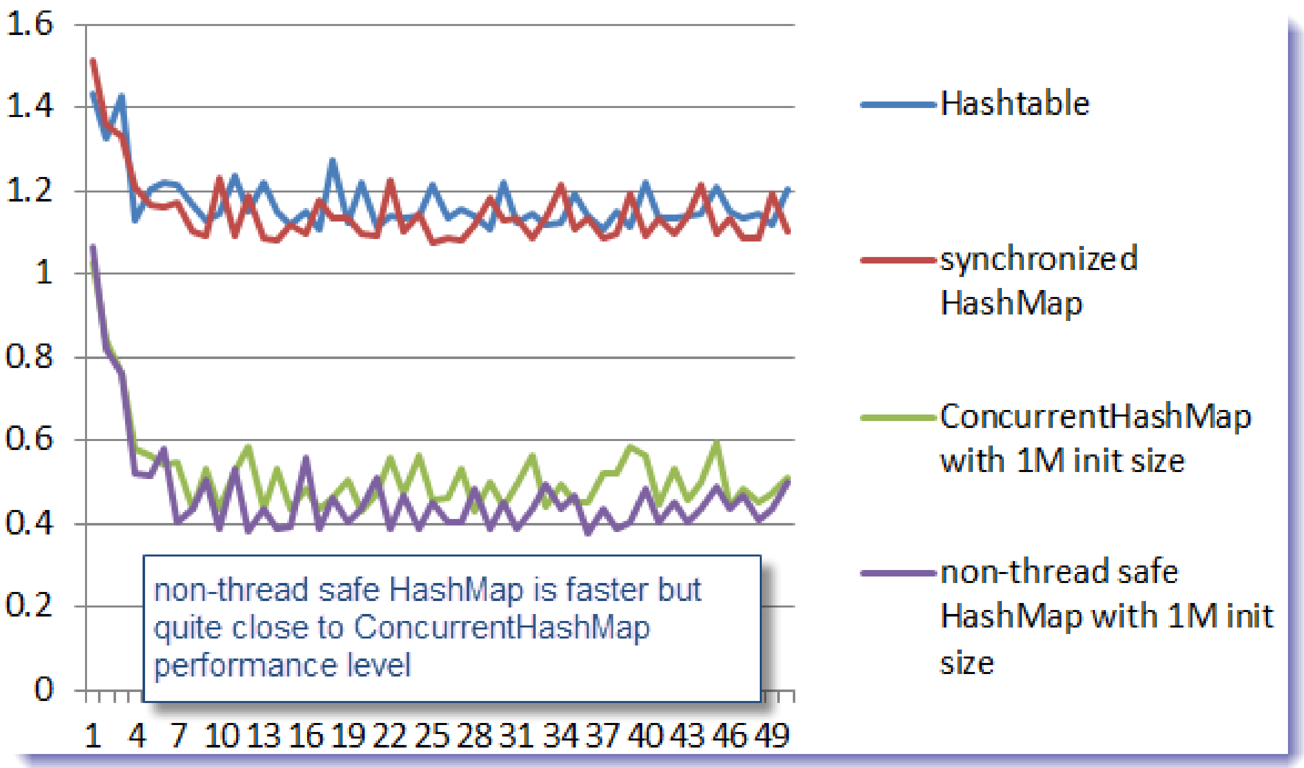
- CopyOnWriteArrayList
- CopyOnWriteArraySet
  - ConcurrentSkipListSet
- Iterator
  - Uses a snapshot of the underlying list (or set) and does not reflect any changes to the list or set after the snapshot was created.
  - Never throw a ConcurrentModificationException.
  - Doesn't support remove(), set(o) and add(o) methods, and throws UnsupportedOperationException.
- Use cases
  - Share the data structure among several threads and have few writes and many reads.

CopyOnWriteArrayList

```java
public static void main(String[] args) {
  ① List<String> empList = new ArrayList<>();
    empList.add("John Doe");
    empList.add("Jane Doe");
    empList.add("Rita Smith");

    Iterator<String> empIter = empList.iterator();
    while (empIter.hasNext()) {
        try {
            System.out.println(empIter.next());
            if (!empList.contains("Tom Smith")) {
              ② empList.add("Tom Smith");
            }
      ③ } catch (ConcurrentModificationException e) {
            System.err.println("attempt to modify list during iteration");
            break;
        }
    }

    List<String> empList2 = new CopyOnWriteArrayList<>();
    empList2.add("John Doe");
    empList2.add("Jane Doe");
    empList2.add("Rita Smith");

    empIter = empList2.iterator();
    while (empIter.hasNext()) {
        System.out.println(empIter.next());
        if (!empList2.contains("Tom Smith")) {
          ④ empList2.add("Tom Smith");
        }
    }
}
```

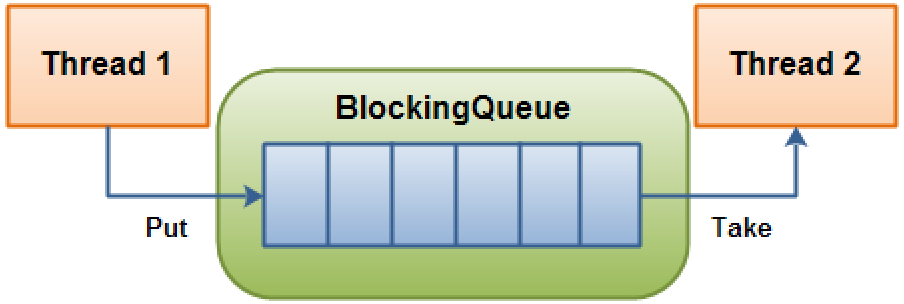
Concurrent Map

- One of most common Java EE performance problems is infinite looping triggered from the non-thread safe HashMap get() and put() operations.
- Performance comparison
  - Non-thread safe HashMap
  - ConcurrentHashMap
    - ConcurrentNavigableMap
    - ConcurrentSkipListMap
  - SynchronizedHashMap
  - HashTable



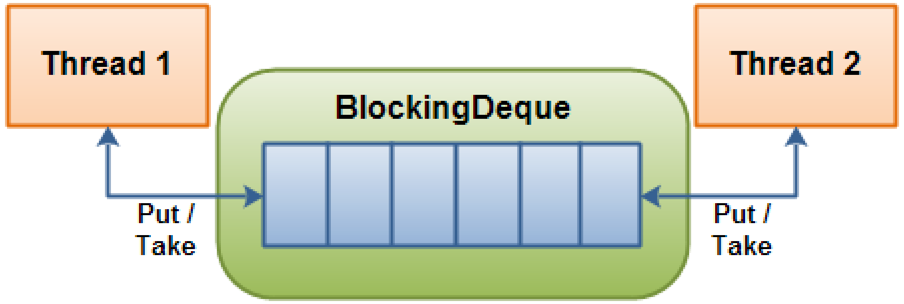
Queue

- BlockingQueue
  - Unidirectional
  - ArrayBlockingQueue
  - ConcurrentLinkedQueue
  - DelayQueue
  - LinkedBlockingQueue
  - PriorityBlockingQueue
  - PriorityQueue
  - SynchronousQueue



Deque

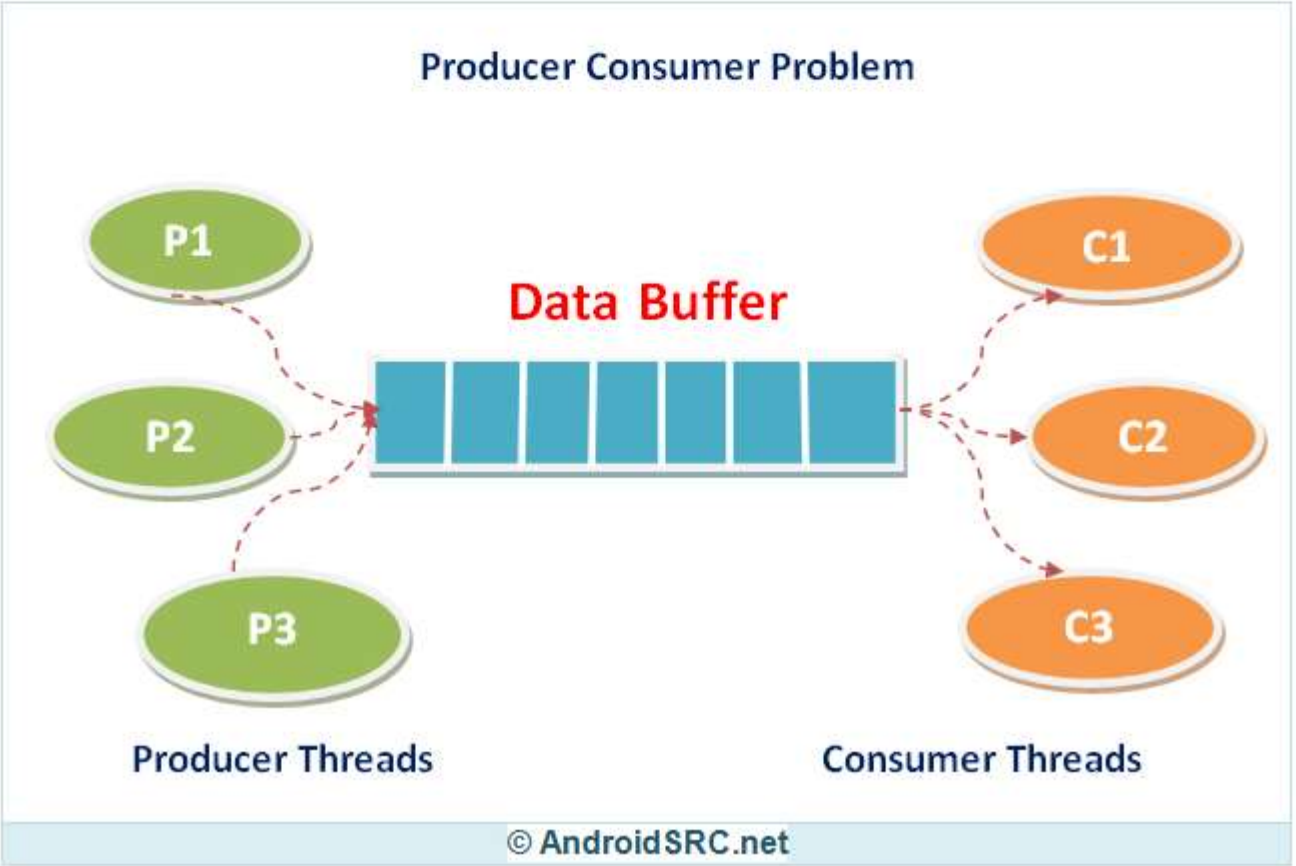
- Deque
  - Bidirectional
  - ArrayDeque
  - ConcurrentLinkedDeque
  - LinkedBlockingDeque



Java Collection Comparison

	Single threaded	Concurrent
Lists	• ArrayList - generic array-based • LinkedList - do not use • Vector - deprecated	• CopyOnWriteArrayList - seldom updated, often traversed
Queues / deques	• ArrayDeque - generic array-based • Stack - deprecated • PriorityQueue - sorted retrieval operations	• ArrayBlockingQueue - bounded blocking queue • ConcurrentLinkedDeque / ConcurrentLinkedQueue - unbounded linked queue (CAS) • DelayQueue - queue with delays on each element • LinkedBlockingDeque / LinkedBlockingQueue - optionally bounded linked queue (locks) • LinkedTransferQueue - may transfer elements w/o storing • PriorityBlockingQueue - concurrent PriorityQueue • SynchronousQueue - Exchanger with Queue interface
Maps	• HashMap - generic map • EnumMap - enum keys • Hashtable - deprecated • IdentityHashMap - keys compared with == • LinkedHashMap - keeps insertion order • TreeMap - sorted keys • WeakHashMap - useful for caches	• ConcurrentHashMap - generic concurrent map • ConcurrentSkipListMap - sorted concurrent map
Sets	• HashSet - generic set • EnumSet - set of enums • BitSet - set of bits/dense integers • LinkedHashSet - keeps insertion order • TreeSet - sorted set	• ConcurrentSkipListSet - sorted concurrent set • CopyOnWriteArraySet - seldom updated, often traversed

Producer–Consumer Problem



Producer-Consumer Example

```java
private static ExecutorService producerPool =
  ① Executors.newFixedThreadPool(2, new NamedThreadFactory("Producer"));

private static class Producer implements Runnable {
  ② private final BlockingQueue<Integer> queue;

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
                Integer randomInt = this.produce();
              ③ this.queue.put(randomInt);

                TimeUnit.SECONDS.sleep(1);
            }
        } catch (InterruptedException e) {
            // nothing to do
        }
    }

    public Integer produce() throws InterruptedException {
        String threadName = Thread.currentThread().getName();

        int randomInt = ThreadLocalRandom.current().nextInt(1, 1000 + 1);
        System.out.printf("[%s] Produce an integer: %s\n", threadName, randomInt);

        return randomInt;
    };
}
```

Producer

```java
private static ExecutorService consumerPool =
  ① Executors.newFixedThreadPool(2, new NamedThreadFactory("Consumer"));

private static class Consumer implements Runnable {
  ② private final BlockingQueue<Integer> queue;

    public Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
              ③ Integer randomInt = this.queue.take();
                this.consume(randomInt);
            }
        } catch (InterruptedException e) {
            // nothing to do
        }
    }

    void consume(Integer value) {
        String threadName = Thread.currentThread().getName();
        System.out.printf("[%s] Consume an integer: %s\n", threadName, value);
    }
}
```

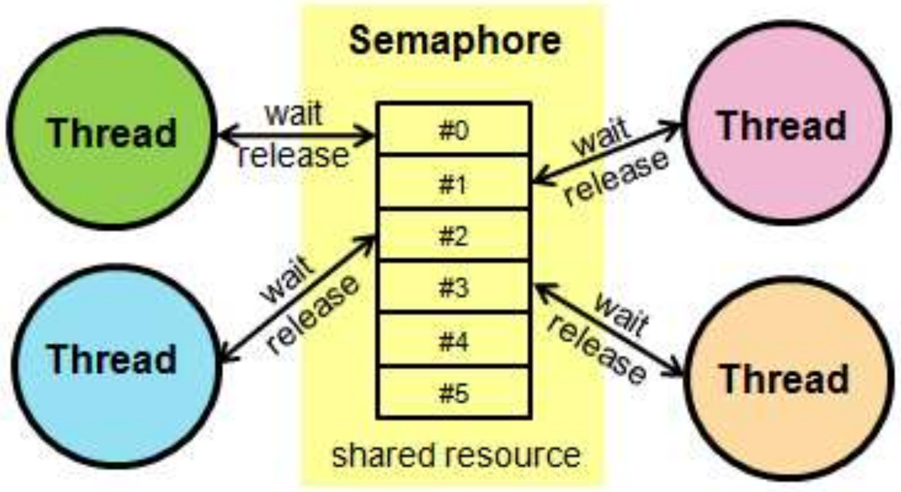
Consumer

Producer-Consumer Example cont.

```
private static BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(100);
1
public static void main(String[] args) throws Exception {
    Producer producer1 = new Producer(queue);
2   Producer producer2 = new Producer(queue);
    Consumer consumer1 = new Consumer(queue);
    Consumer consumer2 = new Consumer(queue);

    producerPool.execute(producer1);
    producerPool.execute(producer2);
    consumerPool.execute(consumer1);
    consumerPool.execute(consumer2);

    TimeUnit.MINUTES.sleep(1);

3   producerPool.shutdown();
    consumerPool.shutdown();

    producerPool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    consumerPool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
}
```

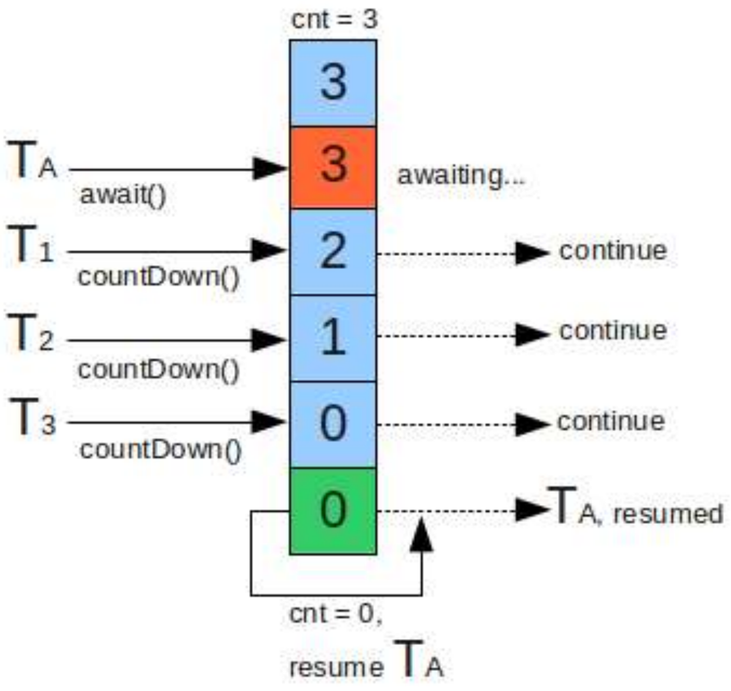
Semaphore

- A Semaphore is capable of restricting thread access to a common resource, or sending signals between threads to avoid missed signals.
- It's often implemented as a protected variable whose value is incremented by acquire(): -1 and decremented by release(): +1.
- Objective
  - Guarding Critical Sections
  - Sending Signals Between Threads
- Use cases
  - Limiting concurrent access to disk
  - Thread creation limiting
  - JDBC connection pooling / limiting
  - Network connection throttling
  - Throttling CPU or memory intensive tasks



CountDownLatch

- A CountDownLatch causes one or more threads to wait for a given set of operations to complete.
- The CountDownLatch is initialized with a count. Threads may call await() to wait for the count to reach 0. Other threads may call countDown(): -1 to reduce count.
- Not reusable once the count has reached 0.
- Use cases
  - Achieving Maximum Parallelism
  - Wait for several threads to complete



Semaphore Example

```java
ExecutorService threadPool = Executors.newFixedThreadPool(3);
final Semaphore semaphore = new Semaphore(2);

int TASK_SIZE = 20;
final CountDownLatch countDownLatch = new CountDownLatch(TASK_SIZE);

List<Callable<String>> taskList = new ArrayList<>();

for (int i = 0; i < TASK_SIZE; i++) {
    Callable<String> task = new Callable<String>() {
        @Override
        public String call() throws Exception {
            String result = null;

            try {
                semaphore.acquire();

                long time = System.currentTimeMillis();

                String threadName = Thread.currentThread().getName();
                UUID uuid = UUID.randomUUID();
                result = String.format("[%s] Produce a UUID: %s at %d",
                        threadName, uuid, time);

                TimeUnit.SECONDS.sleep(1);

            } catch (InterruptedException e) {
                // nothing to do
            } finally {
                semaphore.release();
                countDownLatch.countDown();
            }

            return result;
        }
    };

    taskList.add(task);
}
```

Semaphore

```java
List<Future<String>> futureList = null;
try {
    futureList = threadPool.invokeAll(taskList);

} catch (InterruptedException e) {
    // nothing to do
}

try {
    countDownLatch.await();

} catch (InterruptedException e) {
    // nothing to do
}

for (int i = 0; i < futureList.size(); i++) {
    Future<String> future = futureList.get(i);

    if (future.isDone()) {
        try {
            String result = future.get();
            System.out.printf("%02d. %s\n", i + 1, result);

        } catch (InterruptedException | ExecutionException e) {
            // nothing to do
        }
    }
}

try {
    threadPool.shutdown();
    threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);

} catch (InterruptedException e) {
    // nothing to do
}
```

Semaphore Cont.

CountDownLatch Example

CountDownLatch

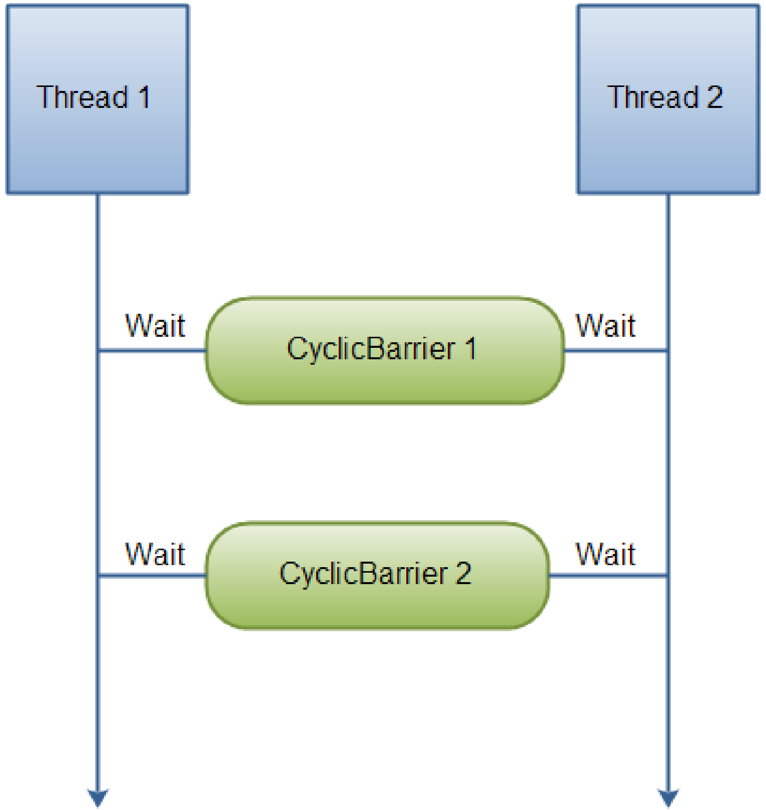
```java
final int WORKER_COUNT = 3;
final CountDownLatch startSignal = new CountDownLatch(1);
final CountDownLatch doneSignal = new CountDownLatch(WORKER_COUNT);
1
for (int i = 0; i < WORKER_COUNT; ++i) {
    // create and start threads
    Thread worker = new Thread(new Runnable() {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();

            try {
              2 startSignal.await();

                System.out.printf("thread %s is working%n", name);

                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    // nothing to do
                }
            } catch (InterruptedException e) {
                // nothing to do
            } finally {
                System.out.printf("thread %s finishing%n", name);
              3 doneSignal.countDown();
            }
        }
    });

    worker.start();
}
```

CountDownLatch cont.

```java
System.out.println("about to let threads proceed");
// let all threads proceed
startSignal.countDown();
1
System.out.println("doing work");

System.out.println("waiting for threads to finish");
try {
    // wait for all threads to finish
  2 doneSignal.await();

} catch (InterruptedException e) {
    // nothing to do
}

System.out.println("main thread terminating");
```

CyclicBarrier

- A CyclicBarrier lets a set of threads wait for each other to reach a common barrier point.
- It can be reused indefinitely after the waiting threads are released.
- Participants call await() and block until the count is reached, at which point an optional barrier task is executed by the last arriving thread, and all threads are released.
- Use cases
  - Multiplayer games that cannot start until the last player has joined.



CyclicBarrier Example

```java
final CyclicBarrier barrier = new CyclicBarrier(3, new Runnable() {
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.printf("Thread %s executing barrier action.%n", name);
    }
});

Runnable task = new Runnable() {
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.printf("%s about to join game...%n", name);

        try {
            barrier.await();

        } catch (BrokenBarrierException e) {
            // nothing to do
            return;
        } catch (InterruptedException e) {
            // nothing to do
            return;
        }

        System.out.printf("%s has joined game%n", name);
    }
};
```

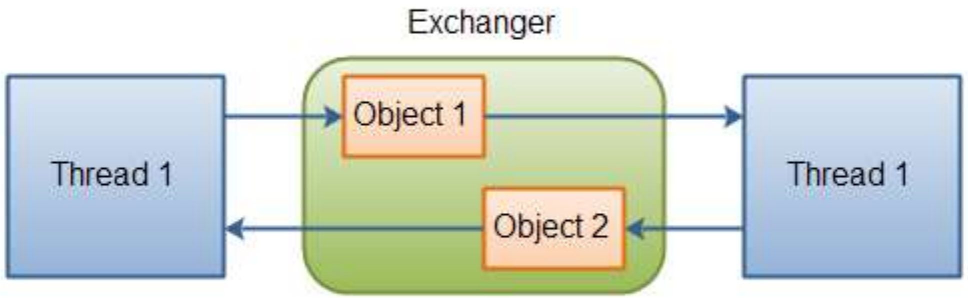
```java
ExecutorService[] executors = new ExecutorService[] {
        Executors.newSingleThreadExecutor(),
        Executors.newSingleThreadExecutor(),
        Executors.newSingleThreadExecutor() };

for (ExecutorService executor : executors) {
    executor.execute(task);
    executor.shutdown();
}
```

CyclicBarrier

CyclicBarrier cont.

Exchanger

- An Exchanger (rendezvous) lets a pair of threads exchange data items. An exchanger is similar to a cyclic barrier whose count is set to 2 but also supports exchange of data when both threads reach the exchange point.
- An Exchanger waits for threads to meet at the exchange() method and swap values atomically.
- Use cases
  - Genetic Algorithm, Pipeline Design



Exchanger Example

```java
private static Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();

public static void main(String[] args) {
    new Thread(new Remover(new DataBuffer("ITEM-"))).start();
    new Thread(new Adder(new DataBuffer())).start();
}

private static class Remover implements Runnable {

    private DataBuffer buffer;

    public Remover(DataBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            while (true) {
                this.takeFromBuffer(this.buffer);

                if (this.buffer.isEmpty()) {
                    this.buffer = exchanger.exchange(this.buffer);
                    System.out.println("Remover's buffer after exchanging: "
                            + this.buffer);

                    TimeUnit.SECONDS.sleep(1);
                }
            }
        } catch (InterruptedException e) {
            // nothing to do
        }
    }

    private void takeFromBuffer(DataBuffer buffer) {
        System.out.printf("Taking %s%n", buffer.remove());
    }
}
```

Exchanger

```java
private static class Adder implements Runnable {

    private int count = 0;

    private DataBuffer buffer;

    public Adder(DataBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            while (true) {
                this.addToBuffer(this.buffer);

                if (this.buffer.isFull()) {
                    this.buffer = exchanger.exchange(this.buffer);
                    System.out.println("Adder's buffer after exchanging: "
                            + this.buffer);

                    TimeUnit.SECONDS.sleep(1);
                }
            }
        } catch (InterruptedException e) {
            // nothing to do
        }
    }

    private void addToBuffer(DataBuffer buffer) {
        String item = "NEWITEM-" + (++this.count);
        System.out.printf("Adding %s%n", item);
        buffer.add(item);
    }
}
```

Exchanger cont.

Phaser

- A Phaser (introduced in Java 7) is similar to a CyclicBarrier in that it lets a group of threads wait on a barrier and then proceed after the last thread arrives.
- It’s similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage.
- Unlike a cyclic barrier, which coordinates a fixed number of threads, a Phaser can coordinate a variable number of threads, which can register or deregister at any time.

Phaser Example

```java
private static Phaser phaser;
❶
public static void main(String[] args) {
❷  phaser = new Phaser(1);

    new Thread(new Racer(1)).start();
    new Thread(new Racer(3)).start();
    new Thread(new Racer(5)).start();

    String threadName = Thread.currentThread().getName();
    System.out.printf("%s arrives at %,3d, parties: %d%n", threadName,
            System.currentTimeMillis(), phaser.getRegisteredParties());
❸  phaser.arriveAndDeregister();
    System.out.printf("%s deregisters at %,3d, parties: %d%n", threadName,
            System.currentTimeMillis(), phaser.getRegisteredParties());
}
```

```java
private static class Racer implements Runnable {
    private int sleep;

    public Racer(int sleep) {
        this.sleep = sleep;
    }

    public void run() {
   ❶  phaser.register();

        String threadName = Thread.currentThread().getName();
        System.out.printf("%s begins at %,3d%n", threadName,
                System.currentTimeMillis());

        try {
            TimeUnit.SECONDS.sleep(this.sleep);
        } catch (InterruptedException e) {
            // nothing to do
        }

        System.out.printf("%s in phase[%d] waits at %,3d%n", threadName,
                phaser.getPhase(), System.currentTimeMillis());

   ❷  phaser.arriveAndAwaitAdvance();

        System.out.printf("%s in phase[%d] advances at %,3d%n", threadName,
                phaser.getPhase(), System.currentTimeMillis());

        try {
            TimeUnit.SECONDS.sleep(this.sleep);
        } catch (InterruptedException e) {
            // nothing to do
        }

        System.out.printf("%s in phase[%d] ends at %,3d%n", threadName,
                phaser.getPhase(), System.currentTimeMillis());
    }
}
```

Phaser

Phaser cont.

CountDownLatch vs. CyclicBarrier vs. Phaser

- CountDownLatch
  - Created with a fixed number of threads.
  - Cannot be reset.
  - Allows threads to wait or continue with its execution.
- CyclicBarrier
  - Created with a fixed number of threads.
  - Can be reset.
  - The threads have to wait till all the threads arrive.
- Phaser
  - Can register/add or deregister/remove threads dynamically.
  - Can be reset.
  - Allows threads to wait or continue with its execution.
  - Supports multiple Phases.

Lock and ReadWriteLock

- Lock
  - Implemented by ReentrantLock.
  - It provides all the features of synchronized keyword with additional ways to create different Conditions for locking, providing timeout for thread to wait for lock.
- ReadWriteLock
  - Implemented by ReentrantReadWriteLock.
  - It contains a pair of associated locks, Read Lock for read-only operations and Write Lock for writing. The Read Lock may be held simultaneously by multiple reader threads as long as there are no writer threads. The Write Lock is exclusive.

Lock vs. Synchronization

- Lock
  - Ability to lock interruptibly.
  - Ability to timeout while waiting for lock.
  - Power to create fair lock.
  - API to get list of waiting thread for lock.
  - Flexibility to try for lock without blocking.
- Synchronization
  - Not required a try-finally block to release lock.
  - Easy to read code.

ReentrantLock Example

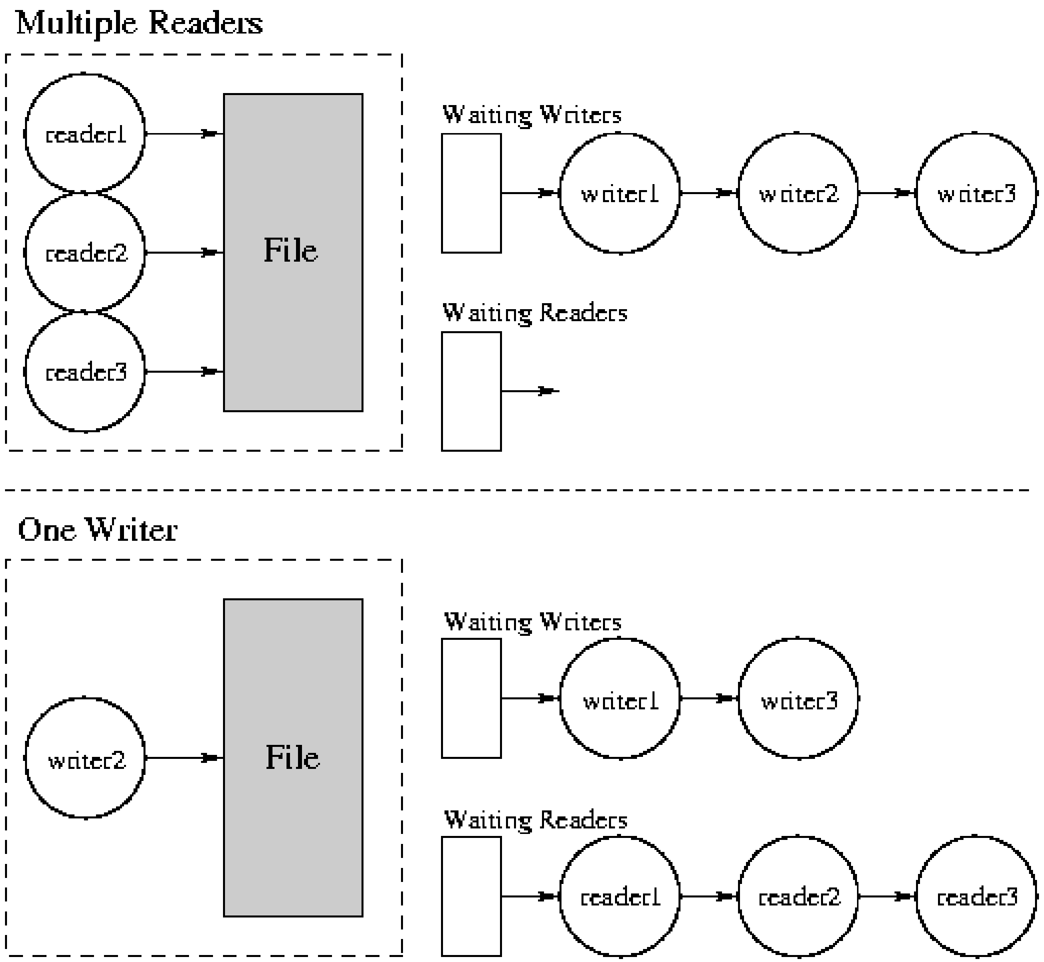
```java
private final ReentrantLock lock = new ReentrantLock();
①
private long idCounter = 0;

public long nextIdByLock() {
② this.lock.lock();

    try {
        if (Long.MAX_VALUE == this.idCounter) {
            this.idCounter = 0;
        }
        ++this.idCounter;

        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + " gets next id: " + this.idCounter
                + " at " + System.currentTimeMillis());

        return this.idCounter;

    } finally {
   ③ this.lock.unlock();
    }
}

public synchronized long nextIdBySync() {
④ if (Long.MAX_VALUE == this.idCounter) {
        this.idCounter = 0;
    }
    ++this.idCounter;

    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + " gets next id: " + this.idCounter
            + " at " + System.currentTimeMillis());

    return this.idCounter;
}
```

ReentrantLock

```java
public static void main(String args[]) {
    final ReentrantLockDemo counter = new ReentrantLockDemo();

    Runnable task = new Runnable() {
        @Override
        public void run() {
            while (true) {
              ① counter.nextIdByLock();
            }
        }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

    t1.start();
    t2.start();
}
```

ReentrantLock cont.

Reader-Writer Problem



Reader-Writer Example

```java
private static final ReentrantReadWriteLock lock =
1       new ReentrantReadWriteLock(true);

private static String message = "$$";

public static void main(String[] args) throws Exception {
    Thread t1 = new Thread(new Reader());
2   Thread t2 = new Thread(new Reader());
    Thread t3 = new Thread(new Writer("a", 250));
    Thread t4 = new Thread(new Writer("b", 500));

    t1.start();
    t2.start();
    t3.start();
    t4.start();

    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

private static class Reader implements Runnable {
    public void run() {
        String threadName = Thread.currentThread().getName();

        for (int i = 0; i <= 10; i++) {
          3 if (lock.isWriteLocked()) {
                System.out.println("I'll take the lock from Writer");
            }
          4 lock.readLock().lock();
            System.out.printf("Reader %s ---> Message is %s%n", threadName,
                    message);
          5 lock.readLock().unlock();
        }
    }
}
```

Reader

```java
private static class Writer implements Runnable {
    private int sleep;

    private String word;

    public Writer(String word, int sleep) {
        this.word = word;
        this.sleep = sleep;
    }

    public void run() {
        for (int i = 0; i <= 10; i++) {
            try {
              1 lock.writeLock().lock();
                try {
                    TimeUnit.MILLISECONDS.sleep(this.sleep);
                } catch (InterruptedException e) {
                    // nothing to do
                }
                message = message.concat(this.word);
            } finally {
              2 lock.writeLock().unlock();
            }
        }
    }
}
```

Writer

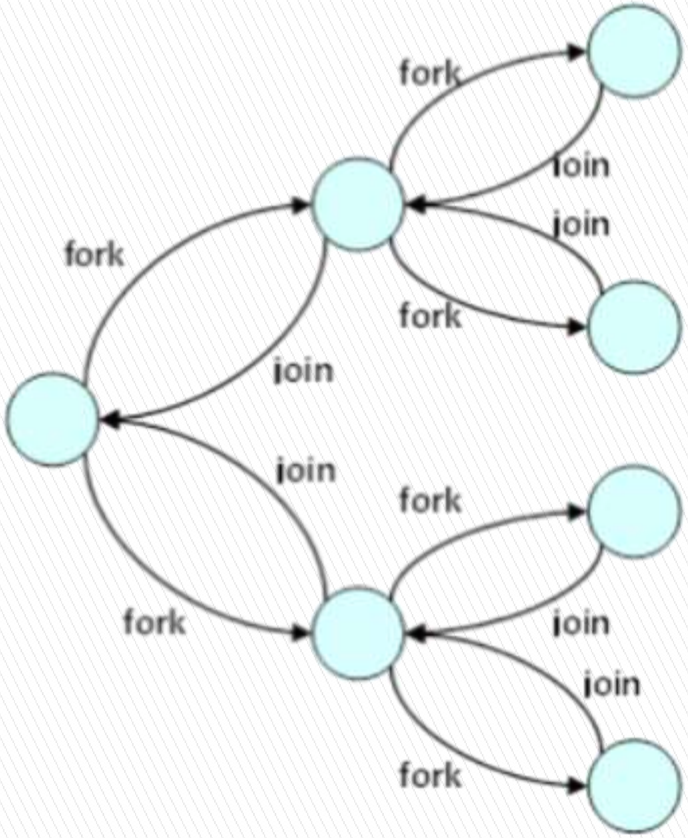
Fork/Join Framework

- Fork/Join Framework (introduced in Java 7)
  - A style of parallel programming in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel.

    – *Professor Doug Lea*
- The Fork/Join Framework is a special executor service for running a special kind of task. It is designed to work well with for divide-and-conquer, or recursive task-processing.
  1. Separate (fork) each large task into smaller tasks.
  2. Process each task in a separate thread (separating those into even smaller tasks if necessary).
  3. Join the results.

Fork/Join Principle

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve part
        join all subtasks spawned in previous loop
        combine results from subtasks
```



Fork/Join Pseudo Code

Fork/Join Process

Fork/Join Pool and Action

- ForkJoinPool
  - An ExecutorService implementationthat runs ForkJoinTasks.
- ForkJoinWorkerThread
  - A thread managed by a ForkJoinPool, which executes ForkJoinTasks.
- ForkJoinTask
  - Describes thread-like entities that have a much lighter weight than normal threads. Many tasks and subtasks can be hosted by very few actual threads.
  - RecursiveAction
    - A recursive resultless ForkJoinTask.
  - RecursiveTask
    - A recursive result-bearing ForkJoinTask.

Fork/Join Example

```java
private static class SortTask extends RecursiveAction {

    private final long[] array;

    private final int lo, hi;

    public SortTask(long[] array, int lo, int hi) {
        this.array = array;
        this.lo = lo;
        this.hi = hi;
    }

    public SortTask(long[] array) {
        this(array, 0, array.length);
    }

    private final static int THRESHOLD = 1000;

    @Override
    protected void compute() {
        if (this.hi - this.lo < THRESHOLD) {
            this.sortSequentially(this.lo, this.hi);
        } else {
            int mid = (this.lo + this.hi) >>> 1;

            invokeAll(new SortTask(this.array, this.lo, mid), new SortTask(
                    this.array, mid, this.hi));

            this.merge(this.lo, mid, this.hi);
        }
    }

    private void sortSequentially(int lo, int hi) {
        Arrays.sort(this.array, lo, hi);
    }

    private void merge(int lo, int mid, int hi) {
        long[] buf = Arrays.copyOfRange(this.array, lo, mid);
        for (int i = 0, j = lo, k = mid; i < buf.length; j++) {
            this.array[j] = (k == hi || buf[i] < this.array[k]) ? buf[i++]
                    : this.array[k++];
        }
    }
}
```

Fork/Join

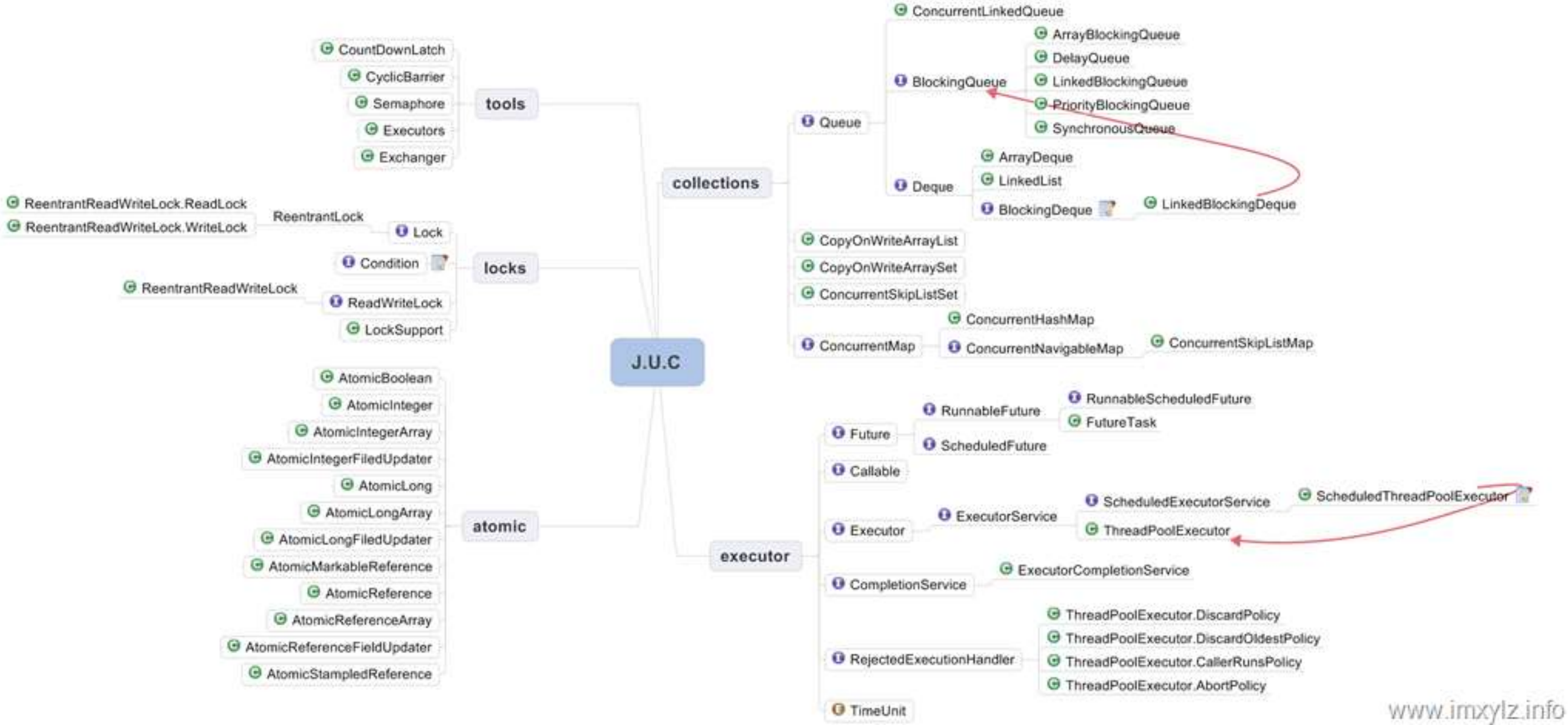
```java
public static void main(String[] args) {
    long[] array = new long[3000000];
    for (int i = 0; i < array.length; i++) {
        array[i] = (long) (Math.random() * 10000000);
    }

    long[] array2 = new long[array.length];
    System.arraycopy(array, 0, array2, 0, array.length);

    long startTime = System.currentTimeMillis();
    Arrays.sort(array, 0, array.length - 1);

    System.out.printf("sequential sort completed in %d millis%n",
            System.currentTimeMillis() - startTime);
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }

    System.out.println();

    int cores = Runtime.getRuntime().availableProcessors();
    ForkJoinPool pool = new ForkJoinPool();
    startTime = System.currentTimeMillis();
    pool.invoke(new SortTask(array2));

    System.out.printf("parallel sort completed in %d millis, cores: %d%n",
            System.currentTimeMillis() - startTime, cores);
    for (int i = 0; i < array2.length; i++) {
        System.out.println(array2[i]);
    }
}
```

Fork/Join cont.

Fork/Join vs. Executor Service

- Fork/Join allows you to easily execute divide-and-conquer jobs, which have to be implemented manually if you want to execute it in ExecutorService.
- In practice ExecutorService is usually used to process many independent requests concurrently, and fork-join when you want to accelerate one coherent job.

java.util.concurrent Mind Map



Reference (1)

- Added Value of Task Parallelism in Batch Sweeps
- Java 7 util.concurrent API UML Class Diagram Examples
- Java performance tuning tips or everything you want to know about Java performance in 15 minutes
- Thread synchronization, object level locking and class level locking
- Java 7: HashMap vs ConcurrentHashMap
- HashMap Vs. ConcurrentHashMap Vs. SynchronizedMap – How a HashMap can be Synchronized in Java

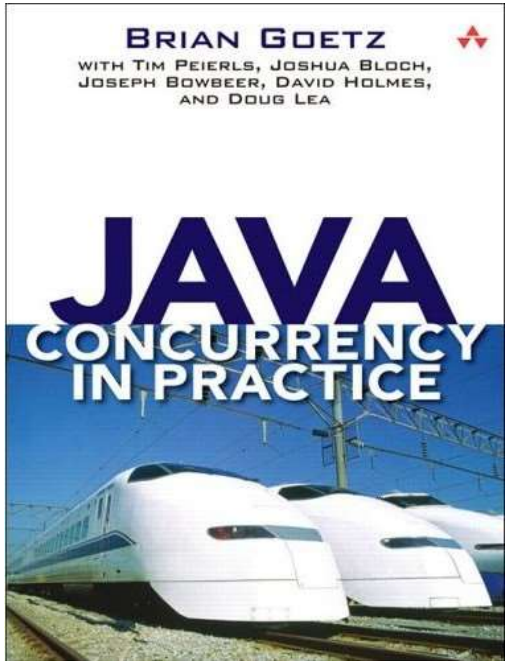
Reference (2)

- Java concurrency: Understanding CopyOnWriteArrayList and CopyOnWriteArraySet
- java.util.concurrent.Phaser Example
- Java Tip: When to use ForkJoinPool vs ExecutorService
- Java 101: Java concurrency without the pain, Part 1
- Java 101: Java concurrency without the pain, Part 2
- Book excerpt: Executing tasks in threads
- Modern threading for not-quite-beginners
- Modern threading: A Java concurrency primer

Reference (3)

- Java concurrency (multi-threading) - Tutorial
- Understanding the Core Concurrency Concepts
- Java Concurrency / Multithreading Tutorial
- java.util.concurrent - Java Concurrency Utilities
- Java BlockingQueue Example implementing Producer Consumer Problem
- Java Concurrency with ReadWriteLock
- Java Lock Example and Concurrency Lock vs synchronized
- Java ReentrantReadWriteLock Example
- ReentrantLock Example in Java, Difference between synchronized vs ReentrantLock

Q & A


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE


Java Multithread Design Pattern
Java多執行緒與平行處理
KingStone


增訂第二版
Design Patterns
於Java語言上的實習應用
KingStone