Core Java Concurrency

ORIGINAL BY **ALEX MILLER** UPDATE BY **IGOR SOROKIN**

CONTENTS

INTRODUCTION

From its creation, Java has supported key concurrency concepts such as threads and locks. This Refcard will help Java developers working with multi-threaded programs to understand core concurrency concepts and how to apply them.

CONCEPTS

CONCEPT	DESCRIPTION
Atomicity	An atomic operation is one which is executed in an all or nothing fashion, therefore partial state is impossible.
Visibility	The conditions when one thread sees changes made by another thread

Table 1: Concurrency concepts

RACE CONDITION

A race condition occurs when more than one thread is performing a series of actions on shared resources, and several possible outcomes can exist based on the order of the actions from each thread. The below code is not thread-safe and the value could be initialized more than once, as `check-then-act` (check for `null`, then initialize) that lazily initializes the field is not **atomic**.

```
class Lazy <T> {
  private volatile T value;

  T get() {
    if (value == null)
      value = initialize();
    return value;
  }
}
```

DATA RACE

A data race occurs when 2 or more threads try to access the same non-final variable without synchronization. Not using synchronization may lead to making changes which are **not visible** to other threads, so reading the stale data is possible, which in turn may have consequences such as infinite loops, corrupted data structures, or inaccurate computations. This code might result in an infinite loop, because the reader thread may never observe the changes made by the writer threads:

```
class Waiter implements Runnable {
  private boolean shouldFinish;

  void finish() { shouldFinish = true; }

  public void run() {
    long iteration = 0;
    while (!shouldFinish) {
      iteration++;
    }
    System.out.println("Finished after: " +
iteration);
  }
}

class DataRace {

  public static void main(String[] args)
    throws InterruptedException {
    Waiter waiter = new Waiter();
    Thread waiterThread = new Thread(waiter);
    waiterThread.start();

    waiter.finish();
    waiterThread.join();
  }
}
```

JAVA MEMORY MODEL: HAPPENS-BEFORE RELATIONSHIP

The Java memory model is defined in terms of actions like reading and writing fields, and synchronizing on a monitor. Actions can be ordered by a **happens-before relationship**, that can be used to reason about when a thread sees the result of another thread's actions, and what constitutes a properly synchronized program.

Happens-before relationships have the following properties:

- The invocation of `Thread#start` happens before any action in this thread.
- Releasing a monitor happens before any subsequent acquisition of the same monitor.
- A write to a volatile variable happens before any subsequent read of a volatile variable.
- A write to a final variable happens before the reference of the object is published.

- All actions in a thread happen before returning from a `Thread#join` on that thread.

In Image 1, `Action X` happens before `Action Y`, therefore in `Thread 2` all operations to the right of `Action Y` will see all the operations to the left of `Action X` in `Thread 1`.

Image 1: *Happens-before illustration*

STANDARD SYNCHRONIZATION FEATURES

THE `SYNCHRONIZED` KEYWORD

The `synchronized` keyword is used to prevent different threads executing the same code block simultaneously. It guarantees that since you acquire a lock (by entering the synchronized block), data, which is protected by this lock, can be manipulated in exclusive mode, so the operation can be **atomic**. Also, it guarantees that other threads will observe the result of the operation after they acquire the same lock.

```
class AtomicOperation {
  private int counter0;
  private int counter1;

  void increment() {
    synchronized (this) {
      counter0++;
      counter1++;
    }
  }
}
```

The synchronized keyword can be also specified on a method level.

TYPE OF METHOD	REFERENCE WHICH IS USED AS A MONITOR
static	The class object of the class with the method
non-static	The `this` reference

Table 2: *Monitors, which are used when the whole method is synchronized*

```
class Reentrantcy {

  synchronized void doAll() {
    doFirst();
    doSecond();
  }

  synchronized void doFirst() {
    System.out.println("First operation is" +
    "successful.");
  }

  synchronized void doSecond() {
    System.out.println("Second operation is" +
    "successful.");
  }
}
```

The level of contention affects how the monitor is acquired:

STATE	DESCRIPTION
init	Just created, never acquired.
biased	There is no contention and the code protected by the lock is executed only by one thread. The cheapest one to acquire.
thin	Monitor is acquired by several threads with no contention. Relatively cheap CAS is used for taking the lock.
fat	There is contention. The JVM requests an OS mutex and lets the OS scheduler handle thread-parking and wake ups.

Table 3: *Monitor states*

WAIT/NOTIFY

`wait/notify/notifyAll` methods are declared in the `Object` class. `wait` is used to make a thread to advance to the `WAITING` or `TIMED_WAITING` (if the time-out value is passed) status. In order to wake up a thread, any of these actions can be done:

- Another thread invokes `notify`, which wakes up an arbitrary thread waiting on the monitor.
- Another thread invokes `notifyAll`, which wakes up all the threads waiting on the monitor.
- `Thread#interrupt` is invoked. In this case, `InterruptedException` is thrown.

The most common pattern is a condition loop:

```
class ConditionLoop {
  private boolean condition;

  synchronized void waitForCondition()
    throws InterruptedException {
    while (!condition) {
      wait();
    }
  }

  synchronized void satisfyCondition() {
    condition = true;
    notifyAll();
  }
}
```

- Keep in mind that in order to use `wait/notify/notifyAll` on an object, you need to acquire the lock on this object first.
- Always wait inside a loop that checks the condition being waited on – this addresses the timing issue if another thread satisfies the condition before the wait begins. Also, it protects your code from spurious wake-ups that can (and do) occur.
- Always ensure that you satisfy the waiting condition before calling `notify/notifyAll`. Failing to do so will cause a notification but no thread will ever be able to escape its wait loop.

THE VOLATILE KEYWORD

`volatile` solves the problem of **visibility**, and makes changes of the variable's value to be **atomic**, because there is a happens-before relationship: write to a volatile variable happens before any subsequent read from the volatile variable. Therefore, it guarantees that any subsequent reads of the field will see the value, which was set by the most recent write.

```
class VolatileFlag implements Runnable {

  private volatile boolean shouldStop;

  public void run() {
    while (!shouldStop) {
      //do smth
    }
    System.out.println("Stopped.");
  }

  void stop() {
    shouldStop = true;
  }

  public static void main(String[] args) throws
InterruptedException {
    VolatileFlag flag = new VolatileFlag();
    Thread thread = new Thread(flag);
    thread.start();

    flag.stop();
    thread.join();
  }
}
```

ATOMICS

The `java.util.concurrent.atomic` package contains a set of classes that support atomic compound actions on a single value in a lock-free manner similar to `volatile`.

Using AtomicXXX classes, it is possible to implement an atomic `check-then-act` operation:

```
class CheckThenAct {
  private final AtomicReference<String> value =
new AtomicReference<>();

  void initialize() {
    if (value.compareAndSet(null, "value")) {
      System.out.println("Initialized only once.");
    }
  }
}
```

Both `AtomicInteger` and `AtomicLong` have atomic `increment/decrement` operation:

```
class Increment {
  private final AtomicInteger state =
new AtomicInteger();

  void advance() {
    int oldState = state.getAndIncrement();
    System.out.println("Advanced: '" + oldState +
 "' -> '" + (oldState + 1) + "'.");
  }
}
```

If you want to have a counter and do not need to get its value atomically, consider using `LongAdder` instead of `AtomicLong/AtomicInteger`. `LongAdder` maintains the value across several cells and grows their number if it's needed, consequently it performs better under high contention.

THREADLOCAL

One way to contain data within a thread and make locking unnecessary is to use ThreadLocal storage. Conceptually, ThreadLocal acts as if there is a variable with its own version in every Thread. ThreadLocals are commonly used for stashing per-Thread values like the "current transaction" or other resources. Also, they are used to maintain per-thread counters, statistics, or ID generators.

```
class TransactionManager {
  private final
  ThreadLocal<Transaction> currentTransaction
  = ThreadLocal.withInitial(NullTransaction::new);

  Transaction currentTransaction() {
    Transaction current = currentTransaction.get();
    if (current.isNull()) {
      current = new TransactionImpl();
      currentTransaction.set(current);
    }
    return current;
  }
}
```

SAFE PUBLICATION

Publishing an object is making its reference available outside of the current scope (for example: return a reference from a getter). Ensuring that object is published safely (only when it is fully constructed) may require synchronization. The safe publication could be achieved using:

- **Static initializers**. Only one thread can initialize static variables because initialization of the class is done under an exclusive lock.

```
class StaticInitializer {
  // Publishing an immutable object without
  //additional initialization
  public static final Year year = Year.of(2017);
  public static final Set<String> keywords;

  // Using static initializer to construct a
  //complex object
  static {
    // Creating mutable set
    Set<String> keywordsSet = new HashSet<>();
    // Initializing state
    keywordsSet.add("java");
    keywordsSet.add("concurrency");
    // Making set unmodifiable
    keywords = Collections.
unmodifiableSet(keywordsSet);
  }
}
```

- **Volatile field**. The reader thread will always read the most

recent value because a write to a volatile variable **happens before** any subsequent read.

```
class Volatile {
  private volatile String state;

  void setState(String state) {
    this.state = state;
  }

  String getState() {
    return state;
  }
}
```

- **Atomics**. For example, `AtomicInteger` stores the value in a volatile field, so the same rule for volatile variables is applicable here.

```
class Atomics {
  private final AtomicInteger state =
new AtomicInteger();

  void initializeState(int state) {
    this.state.compareAndSet(0, state);
  }

  int getState() {
    return state.get();
  }
}
```

- **Final Fields**

```
class Final {
  private final String state;

  Final(String state) {
    this.state = state;
  }

  String getState() {
    return state;
  }
}
```

Make sure that the this reference is not escaped during construction.

```
class ThisEscapes {
 private final String name;

 ThisEscapes(String name) {
   Cache.putIntoCache(this);
   this.name = name;
 }

 String getName() { return name; }
}

class Cache {
 private static final Map<String, ThisEscapes>
CACHE = new ConcurrentHashMap<>();

 static void putIntoCache(
ThisEscapes thisEscapes) {
   // 'this' reference escaped before the object
//is fully constructed.

 CACHE.putIfAbsent(thisEscapes.getName(),
thisEscapes);
 }
}
```

- Correctly synchronized field.

```
class Synchronization {

  private String state;

  synchronized String getState() {
    if (state == null)
      state = "Initial";
    return state;
  }
}
```

IMMUTABLE OBJECTS

A great property of immutable objects is that they are thread-safe, so no synchronization is necessary. The requirements for an object to be immutable are:

- All fields are final.
- All fields must be either mutable or immutable objects too, but do not escape the scope of the object so the state of the object cannot be altered after construction.
- this reference does not escape during construction.
- The class is final, so it is not possible to override this behavior in subclasses.

Example of an immutable object:

```
// Marked as final - subclassing is forbidden
public final class Artist {
  // Immutable object, field is final
  private final String name;
  // Collection of immutable objects, field is final
  private final List<Track> tracks;

 public Artist(String name, List<Track> tracks) {
    this.name = name;
     // Defensive copy
    List<Track> copy = new ArrayList<>(tracks);
    // Making mutable collection unmodifiable
    this.tracks = Collections.unmodifiableList(copy);
    // 'this' is not passed to anywhere during
    // construction
  }
  // Getters, equals, hashCode, toString
}

// Marked as final - subclassing is forbidden
public final class Track {
  // Immutable object, field is final
  private final String title;

  public Track(String title) {
    this.title = title;
  }
  // Getters, equals, hashCode, toString
}
```

THREADS

The `java.lang.Thread` class is used to represent an application or JVM thread. The code is always being executed in the context of some Thread class (use `Thread#currentThread()` to obtain your own Thread).

STATE	DESCRIPTION
NEW	Not started.
RUNNABLE	Up and running
BLOCKED	Waiting on a monitor — it is trying to acquire the lock and enter the critical section.
WAITING	Waiting for another thread to perform a particular action (`notify/notifyAll`, `LockSupport#unpark`).
TIMED_WAITING	Same as `WAITING`, but with a timeout.
TERMINATED	Stopped.

Table 4: *Thread states*

THREAD METHOD	DESCRIPTION
start	Starts a Thread instance and execute its `run()` method.
join	Blocks until the `Thread` finishes.
interrupt	Interrupts the thread. If the thread is blocked in a method that responds to interrupts, an `InterruptedException` will be thrown in the other thread, otherwise the interrupt status is set.
stop, suspend, resume, destroy	These methods are all deprecated. They perform dangerous operations depending on the state of the thread in question. Instead, use `Thread#interrupt()` or a volatile flag to indicate to a thread what it should do

Table 5: *Thread coordination methods*

HOW TO HANDLE INTERRUPTEDEXCEPTION?

- Clean up all resources and finish the thread execution if it is possible at the current level.
- Declare that the current method throws `InterruptedException`.
- If a method is not declared to throw `InterruptedException`, the interrupted flag should be restored to true by calling `Thread.currentThread().interrupt()` and an exception, which is more appropriate at this level, should be thrown. It is highly important to set the flag back to true in order to give a chance to handle interruptions at a higher level.

UNEXPECTED EXCEPTION HANDLING

Threads can specify an `UncaughtExceptionHandler` that will receive a notification of any uncaught exception that causes a thread to abruptly terminate.

```
Thread thread = new Thread(runnable);
thread.setUncaughtExceptionHandler((failedThread,
exception) -> {
  logger.error("Caught unexpected exception in thread
      '{}'.", failedThread.getName(), exception);
});
thread.start();
```

LIVENESS

DEADLOCK

A deadlock occurs when there is more than one thread, each waiting for a resource held by another, such that a cycle of resources and acquiring threads is formed. The most obvious kind of resource is an object monitor but any resource that causes blocking (such as `wait/notify`) can qualify.

Potential deadlock example:

```
class Account {
  private long amount;

  void plus(long amount) { this.amount += amount; }

  void minus(long amount) {
    if (this.amount < amount)
      throw new IllegalArgumentException();
    else
      this.amount -= amount;
  }

  static void transferWithDeadlock(long amount,
Account first, Account second){
    synchronized (first) {
      synchronized (second) {
        first.minus(amount);
        second.plus(amount);
      }
    }
  }
}
```

The deadlock happens if at the same time:

- One thread is trying to transfer from the first account to the second, and has already acquired the lock on the first account.
- Another thread is trying to transfer from the second account to the first one, and has already acquired the lock on the second account.

Techniques for avoiding deadlock:

- Lock ordering — always acquire the locks in the same order.

```
class Account {
  private long id;
  private long amount;
  // Some methods are omitted
  static void transferWithLockOrdering(long
amount, Account first, Account second){
    boolean lockOnFirstAccountFirst = first.id <
second.id;
    Account firstLock = lockOnFirstAccountFirst  ?
first  : second;
    Account secondLock = lockOnFirstAccountFirst
? second : first;
    synchronized (firstLock) {
      synchronized (secondLock) {
        first.minus(amount);
        second.plus(amount);
      }
    }
  }
}
```

- Lock with timeout — do not block indefinitely upon acquiring the lock, but rather release all locks and try again.

```
class Account {
  private long amount;
  // Some methods are omitted

  static void transferWithTimeout(
      long amount, Account first, Account second,
      int retries, long timeoutMillis
  ) throws InterruptedException {
    for (int attempt = 0; attempt < retries;
    attempt++) {
      if (first.lock.tryLock(timeoutMillis,
      TimeUnit.MILLISECONDS)){
        try {

          if (second.lock.tryLock(timeoutMillis,
          TimeUnit.MILLISECONDS)){
            try {
              first.minus(amount);
              second.plus(amount);
            }finally {
              second.lock.unlock();
            }
          }

        }finally {
          first.lock.unlock();
        }
      }
    }
  }
}
```

The JVM can detect monitor deadlocks and will print deadlock information in thread dumps.

LIVELOCK AND THREAD STARVATION

Livelock occurs when threads spend all of their time negotiating access to a resource or detecting and avoiding deadlock such that no thread actually makes progress. Starvation occurs when threads hold a lock for long periods such that some threads "starve" without making progress.

JAVA.UTIL.CONCURRENT

THREAD POOLS

The core interface for thread pools is `ExecutorService`. `java.util.concurrent` also provides a static factory class `Executors`, which contains factory methods for the creation of a thread pool with the most common configurations.

METHOD	DESCRIPTION
`newSingleThreadExecutor`	Returns an `ExecutorService` with exactly one thread.
`newFixedThreadPool`	Returns an `ExecutorService` with a fixed number of threads.
`newCachedThreadPool`	Returns an `ExecutorService` with a varying size thread pool.
`newSingleThreadScheduledExecutor`	Returns a `ScheduledExecutorService` with a single thread.
`newScheduledThreadPool`	Returns a `ScheduledExecutorService` with a core set of threads.
`newWorkStealingPool`	Returns an work-stealing `ExecutorService`.

Table 6: *Static factory methods*

When sizing thread pools, it is often useful to base the size on the number of logical cores in the machine running the application. In Java, you can get that value by calling `Runtime.getRuntime().availableProcessors()`.

IMPLEMENTATION	DESCRIPTION
`ThreadPoolExecutor`	Default implementation with an optionally resizing pool of threads, a single working queue and configurable policy for rejected tasks (via `RejectedExecutionHandler`), and thread creation (via `ThreadFactory`).
`ScheduledThreadPoolExecutor`	An extension of `ThreadPoolExecutor` that provides the ability to schedule periodical tasks.
`ForkJoinPool`	Work stealing pool: all threads in the pool try to find and run either submitted tasks or tasks created by other active tasks.

Table 7: *Thread pool implementations*

Tasks are submitted with `ExecutorService#submit`, `ExecutorService#invokeAll`, or `ExecutorService#invokeAny`, which have multiple overloads for different types of tasks.

INTERFACE	DESCRIPTION
Runnable	Represent a task without a return value.
Callable	Represents a computation with a return value. It also declares to throw raw `Exception`, so no wrapping for a checked exception is necessary.

Table 8: *Tasks' functional interfaces*

FUTURE

Future is an abstraction for asynchronous computation. It represents the result of the computation, which might be available at some point: either a computed value or an exception. Most of the methods of the `ExecutorService` use `Future` as a return type. It exposes methods to examine the current state of the future or block until the result is available.

```
ExecutorService executorService = Executors.
newSingleThreadExecutor();
Future<String> future = executorService.submit(()
-> "result");

try {
  String result = future.get(1L, TimeUnit.SECONDS);
  System.out.println("Result is '" + result + "'.");
}
catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new RuntimeException(e);
}
catch (ExecutionException e) {
  throw new RuntimeException(e.getCause());
}
catch (TimeoutException e) {
  throw new RuntimeException(e);
}
assert future.isDone();
```

LOCKS

LOCK

The `java.util.concurrent.locks` package has a standard Lock interface. The `ReentrantLock` implementation duplicates the functionality of the synchronized keyword but also provides additional functionality such as obtaining information about the state of the lock, non-blocking `tryLock()`, and interruptible locking. Example of using an explicit `ReentrantLock` instance:

```
class Counter {
  private final Lock lock = new ReentrantLock();
  private int value;

  int increment() {
    lock.lock();
    try {
      return ++value;
    } finally {
      lock.unlock();
    }
  }
}
```

READWRITELOCK

The `java.util.concurrent.locks` package also contains a `ReadWriteLock` interface (and `ReentrantReadWriteLock` implementation) which is defined by a pair of locks for reading and writing, typically allowing multiple concurrent readers but only one writer.

```
class Statistic {
  private final ReadWriteLock lock =
new ReentrantReadWriteLock();
  private int value;

  void increment() {
    lock.writeLock().lock();
    try {
      value++;
    } finally {
      lock.writeLock().unlock();
    }
  }

  int current() {
    lock.readLock().lock();
    try {
      return value;
    } finally {
      lock.readLock().unlock();
    }
  }
}
```

COUNTDOWNLATCH

The `CountDownLatch` is initialized with a count. Threads may call `await()` to wait for the count to reach 0. Other threads (or the same thread) may call `countDown()` to reduce the count. Not reusable once the count has reached 0. Used to trigger an unknown set of threads once some number of actions has occurred.

COMPLETABLEFUTURE

`CompletableFuture` is an abstraction for async computation. Unlike plain `Future`, where the only possibility to get the result is to block, it's encouraged to register callbacks to create a pipeline of tasks to be executed when either the result or an exception is available. Either during creation (via `CompletableFuture#supplyAsync/runAsync`) or during adding callbacks (`*async` family's methods), an executor, where the computation should happen, can be specified (if it is not specified, it is the standard global `ForkJoinPool#commonPool`).

Take into consideration that if the `CompletableFuture` is already completed, the callbacks registered via non `*async` methods are going to be executed in the caller's thread.

If there are several futures you can use `CompletableFuture#allOf` to get a future, which is

completed when all futures are completed, or `CompletableFuture#anyOf`, which is completed as soon as any future is completed.

```
ExecutorService executor0 = Executors.
newWorkStealingPool();
ExecutorService executor1 = Executors.
newWorkStealingPool();

//Completed when both of the futures are completed
CompletableFuture<String> waitingForAll =
CompletableFuture
    .allOf(
        CompletableFuture.supplyAsync(() -> "first"),
        CompletableFuture.supplyAsync(() -> "second",
        executor1)
    )
    .thenApply(ignored -> " is completed.");

CompletableFuture<Void> future = CompletableFuture.
supplyAsync(() -> "Concurrency Refcard", executor0)
    //Using same executor
    .thenApply(result -> "Java " + result)

    //Using different executor
    .thenApplyAsync(result -> "Dzone " + result,
    executor1)

    //Completed when this and other future are
    //completed
    .thenCombine(waitingForAll, (first, second) -> first
    + second)

    //Implicitly using ForkJoinPool#commonPool as the
    //executor
    .thenAcceptAsync(result -> {
      System.out.println("Result is '" + result +
    "'.");
    })

    //Generic handler
    .whenComplete((ignored, exception) -> {
      if (exception != null)
        exception.printStackTrace();
    });

//First blocking call - blocks until it is not finished.
future.join();

future
    //Executes in the current thread (which is main).
    .thenRun(() -> System.out.println("Current thread
    is '" + Thread.currentThread().getName() + "'."))

    //Implicitly using ForkJoinPool#commonPool as the
    //executor
    .thenRunAsync(() -> System.out.println("Current" +
    "thread is '" + Thread.currentThread().getName() +
    "'."));
```

CONCURRENT COLLECTIONS

The easiest way to make a collection thread-safe is to use `Collections#synchronized` family methods. Because this solution performs poorly under high contention, `java.util.concurrent` provides a variety of data structures which are optimized for concurrent usage.

LIST

IMPLEMENTATION	DESCRIPTION
`CopyOnWriteArrayList`	It provides copy-on-write semantics where each modification of the data structure results in a new internal copy of the data (writes are thus very expensive, whereas reads are cheap). Iterators on the data structure always see a snapshot of the data from when the iterator was created.

Table 9: *Lists in java.util.concurrent*

MAPS

IMPLEMENTATION	DESCRIPTION
`ConcurrentHashMap`	It usually acts as a bucketed hash table. Read operations, generally, do not block and reflect the results of the most recently completed write. The write of the first node in an empty bin is performed by just CASing (compare-and-set) it to the bin, whereas other writes require locks (the first node of a bucket is used as a lock).
`ConcurrentSkipListMap`	It provides concurrent access along with sorted map functionality similar to `TreeMap`. Performance bounds are similar to `TreeMap` although multiple threads can generally read and write from the map without contention as long as they are not modifying the same portion of the map.

Table 10: *Maps in java.util.concurrent*

SETS

IMPLEMENTATION	DESCRIPTION
`CopyOnWriteArraySet`	Similar to `CopyOnWriteArrayList`, it uses copy-on-write semantics to implement the Set interface.
`ConcurrentSkipListSet`	Similar to `ConcurrentSkipListMap`, but implements the Set interface.

Table 11: *Sets in java.util.concurrent*

Another approach to create a concurrent set is to wrap a concurrent map:

```
Set<T> concurrentSet = Collections.newSetFromMap(
new ConcurrentHashMap<T, Boolean>());
```

QUEUES

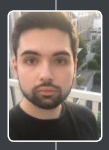
Queues act as pipes between "producers" and "consumers." Items are put in one end of the pipe and emerge from the

other end of the pipe in the same "first-in first-out" (FIFO) order. The `BlockingQueue` interface extends `Queue` to provide additional choices of how to handle the scenario where a queue may be full (when a producer adds an item) or empty (when a consumer reads or removes an item). In these cases, `BlockingQueue` provides methods that either block forever or block for a specified time period, waiting for the condition to change due to the actions of another thread.

IMPLEMENTATION	DESCRIPTION
`ConcurrentLinkedQueue`	An unbounded non-blocking queue backed by a linked list.
`LinkedBlockingQueue`	An optionally bounded blocking queue backed by a linked list.
`PriorityBlockingQueue`	An unbounded blocking queue backed by a min heap. Items are removed from the queue in an order based on the Comparator associated with the queue (instead of FIFO order).
`DelayQueue`	An unbounded blocking queue of elements, each with a delay value. Elements can only be removed when their delay has passed and are removed in the order of the oldest expired item.
`SynchronousQueue`	A 0-length queue where the producer and consumer block until the other arrives. When both threads arrive, the value is transferred directly from producer to consumer. Useful when transferring data between threads.

Table 12: *Queues in java.util.concurrent*

ABOUT THE AUTHOR



IGOR SOROKIN is a Java and Scala developer. He has working experience with big data analytics companies (comScore), highload web projects (Yandex.Music), and big financial institutions (Moscow Exchange).

He has expertise in a wide range of technologies (i.e. Apache Spark, Spring, MongoDB, Akka) and a passion for continuous learning.

Currently, resides in Amsterdam, Netherlands working as a Senior Java Developer at comScore. You can find him on GitHub here, LinkedIn here, and DZone here.

DZone, Inc. 150 Preston Executive Dr. Cary, NC 27513

888.678.0399 - 919.678.0300