## About Java Concurrency

From its creation, Java has supported key concurrency concepts such as threads and locks. This guide helps Java developers working with multi-threaded programs to understand the core concurrency concepts and how to apply them. Topics covered in this guide include built-in Java language features like Thread, synchronized, and volatile, as well as new constructs added in JavaSE 5 such as Locks, Atomic, concurrent collections, thread coordination abstraction, and Executors. Using these building blocks, developers can build highly concurrent and thread-safe Java applications.

## Concepts

This section describes key Java Concurrency concepts that are used throughout this DZone Refcard.

### Table 1: Java Concurrency Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Memory Model</td>
<td>The Java Memory Model (JMM) was defined in Java SE 5 (JSR 133) and specifies the guarantees a JVM implementation must provide to a Java programmer when writing concurrent code. The JMM is defined in terms of actions like reading and writing fields, and synchronizing on a monitor. These actions form an ordering (called the “happens-before” ordering) that can be used to reason about when a thread sees the result of another thread's actions, what constitutes a properly synchronized program, and how to make fields immutable, and more.</td>
</tr>
<tr>
<td>Monitor</td>
<td>In Java, every object contains a “monitor” that can be used to provide mutual exclusion access to critical sections of code. The critical section is specified by marking a method or code block as synchronized. Only one thread at a time is allowed to execute any critical section of code for a particular monitor. When a thread reaches this section of code, it will wait indefinitely for the monitor to be released if another thread holds it. In addition to mutual exclusion, the monitor allows cooperation through the wait and notify operations.</td>
</tr>
<tr>
<td>Atomic field assignment</td>
<td>Assigning a value to a field is an atomic action for all types except doubles and longs. Doubles and longs are allowed to be updated as two separate operations by a JVM implementation so another thread might theoretically see a partial update. To protect updates of shared doubles and longs, mark the field as a volatile or modify it in a synchronized block.</td>
</tr>
<tr>
<td>Race condition</td>
<td>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 are performed.</td>
</tr>
<tr>
<td>Data race</td>
<td>A data race specifically refers to accessing a shared non-final non-volatile field from more than one thread without proper synchronization. The Java Memory Model makes no guarantees about the behavior of unsynchronized access to shared fields. Data races are likely to cause unpredictable behavior that varies between architectures and machines.</td>
</tr>
<tr>
<td>Safe publication</td>
<td>It is unsafe to publish a reference to an object before construction of the object is complete. One way that the thread reference can escape is by registering a listener with a callback during construction. Another common scenario is starting a Thread from the constructor. In both cases, the partially constructed object is visible to other threads.</td>
</tr>
<tr>
<td>Final fields</td>
<td>Final fields must be set to an explicit value by the end of object construction or the compiler will emit an error. Once set, final field values cannot be changed. Marking an object reference field as final does not prevent objects referenced from that field from changing later. For example, a final ArrayList field cannot be changed to a different ArrayList, but objects may be added or removed on the list instance.</td>
</tr>
</tbody>
</table>

## Protecting Shared Data

Writing thread-safe Java programs requires a developer to use proper locking when modifying shared data. Locking establishes the orderings needed to satisfy the Java Memory Model and guarantee the visibility of changes to other threads.

**Hot Tip**

Data changed outside synchronization has NO specified semantics under the Java Memory Model! The JVM is free to reorder instructions and limit visibility in ways that are likely to be surprising to a developer.

**Synchronized**

Every object instance has a monitor that can be locked by one thread at a time. The synchronized keyword can be specified on a method or in block form to lock the monitor. Modifying a field while synchronized on an object guarantees that subsequent reads from any other thread synchronized on the same object will see the updated value. It is important to note that writes outside synchronization or synchronized on a different object than the read are not necessarily ever visible to other threads.

---

### Subscribe Now for FREE!

Refcardz.com

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Get More Refcardz (They’re free!)
The synchronized keyword can be specified on a method or in block form on a particular object instance. If specified on a non-static method, the this reference is used as the instance. In a synchronized static method, the Class defining the method is used as the instance.

**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:

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int value = 0;

    public int increment() {
        lock.lock();
        try {
            ++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. Example of using an explicit ReentrantReadWriteLock to allow multiple concurrent readers:

```java
public class Counter {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private int value = 0;

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

**volatile**
The `volatile` modifier can be 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. Thus, `volatile` provides visibility just like synchronization but scoped only to each read or write of the field. Before Java SE 5, the implementation of `volatile` was inconsistent between JVM implementations and architectures and could not be relied upon. The Java Memory Model now explicitly defines `volatile`'s behavior.

An example of using `volatile` as a signaling flag:

```java
public class Processor implements Runnable {
    private volatile boolean stop;

    public void run() {
        while(! stop) {
            // ... do processing
        }
    }
}
```

**Marking an array as volatile does not make entries in the array volatile! In this case volatile applies only to the array reference itself. Instead, use a class like AtomicIntegerArray to create an array with volatile-like entries.**

**Atomic classes**
One shortcoming of `volatile` is that while it provides visibility guarantees, you cannot both check and update a volatile field in a single atomic call. 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.

```java
public class Counter {
    private AtomicInteger value = new AtomicInteger();

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

The `incrementAndGet` method is just one example of a compound action available on the Atomic classes.

Atomic classes are provided for booleans, integers, longs, and object references as well as arrays of integers, longs, and object references.

**ThreadLocal**
One way to contain data within a thread and make locking unnecessary is to use `ThreadLocal` storage. Conceptually a `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.

```java
public class TransactionManager {
    private static final ThreadLocal<Transaction> currentTransaction = new ThreadLocal<Transaction>() {
        @Override
        protected Transaction initialValue() {
            return new NullTransaction();
        }
    };

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

A key technique for properly protecting shared data is to encapsulate the synchronization mechanism with the class holding the data. This technique makes it impossible to improperly access the data as all usage must conform to the synchronization protocol. The `java.util.concurrent` package holds many data structures designed for concurrent use. Generally, the use of these data structures yields far better performance than using a synchronized wrapper around an unsynchronized collection.

**CONCURRENT COLLECTIONS**
Concurrent lists and sets
The java.util.concurrent package contains three concurrent List and Set implementations described in Table 2.

### Table 2: Concurrent Lists and Sets

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyOnWriteArraySet</td>
<td>CopyOnWriteArraySet 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). Iterators on the data structure always see a snapshot of the data from when the iterator was created.</td>
</tr>
<tr>
<td>CopyOnWriteArrayList</td>
<td>Similar to CopyOnWriteArraySet, CopyOnWriteArrayList uses copy-on-write semantics to implement the List interface.</td>
</tr>
<tr>
<td>ConcurrentSkipListSet</td>
<td>ConcurrentSkipListSet (added in Java SE 6) provides concurrent access along with sorted set functionality similar to TreeSet. Due to the skip list based implementation, multiple threads can generally read and write within the set without contention as long as they aren’t modifying the same portions of the set.</td>
</tr>
</tbody>
</table>

Concurrent maps
The java.util.concurrent package contains an extension to the Map interface called ConcurrentMap, which provides some extra methods described in Table 3. All of these methods perform a set of actions in the scope of a single atomic action. Performing this set of actions outside the map would introduce race conditions due to making multiple (non-atomic) calls on the map.

### Table 3: ConcurrentMap methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>putIfAbsent(K key, V value) : V</td>
<td>If the key is not in the map then put the key/value pair, otherwise do nothing. Returns old value or null if not previously in the map.</td>
</tr>
<tr>
<td>removeObject (Object key, Object value) : boolean</td>
<td>If the map contains key and it is mapped to value then remove the entry, otherwise do nothing.</td>
</tr>
<tr>
<td>replace(K key, V value) : V</td>
<td>If the map contains key then replace with newValue, otherwise do nothing.</td>
</tr>
<tr>
<td>replace(K key, V oldValue, V newValue) : boolean</td>
<td>If the map contains key and it is mapped to oldValue then replace with newValue, otherwise do nothing.</td>
</tr>
</tbody>
</table>

There are two ConcurrentMap implementations available as shown in Table 4.

### Table 4: ConcurrentMap implementations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcurrentHashMap</td>
<td>ConcurrentHashMap provides two levels of internal hashing. The first level chooses an internal segment, and the second level hashes into buckets in the chosen segment. The first level provides concurrency by allowing reads and writes to occur safely on each segment in parallel.</td>
</tr>
<tr>
<td>ConcurrentSkipListMap</td>
<td>ConcurrentSkipListMap (added in Java SE 6) 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 aren’t modifying the same portion of the map.</td>
</tr>
</tbody>
</table>

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 Queue interface was added to java.util in Java SE 5 and while it can be used in single-threaded scenarios, it is primarily used with multiple producers or one or more consumers, all writing and reading from the same queue.

The BlockingQueue interface is in java.util.concurrent and 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).

Concurrent maps
The java.util.concurrent package contains an extension to the Map interface called ConcurrentMap, which provides some extra methods described in Table 3. All of these methods perform a set of actions in the scope of a single atomic action. Performing this set of actions outside the map would introduce race conditions due to making multiple (non-atomic) calls on the map.

### Table 2: Concurrent Lists and Sets

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyOnWriteArraySet</td>
<td>CopyOnWriteArraySet 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). Iterators on the data structure always see a snapshot of the data from when the iterator was created.</td>
</tr>
<tr>
<td>CopyOnWriteArrayList</td>
<td>Similar to CopyOnWriteArraySet, CopyOnWriteArrayList uses copy-on-write semantics to implement the List interface.</td>
</tr>
<tr>
<td>ConcurrentSkipListSet</td>
<td>ConcurrentSkipListSet (added in Java SE 6) provides concurrent access along with sorted set functionality similar to TreeSet. Due to the skip list based implementation, multiple threads can generally read and write within the set without contention as long as they aren’t modifying the same portions of the set.</td>
</tr>
</tbody>
</table>

Concurrent maps
The java.util.concurrent package contains an extension to the Map interface called ConcurrentMap, which provides some extra methods described in Table 3. All of these methods perform a set of actions in the scope of a single atomic action. Performing this set of actions outside the map would introduce race conditions due to making multiple (non-atomic) calls on the map.

### Table 3: ConcurrentMap methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>putIfAbsent(K key, V value) : V</td>
<td>If the key is not in the map then put the key/value pair, otherwise do nothing. Returns old value or null if not previously in the map.</td>
</tr>
<tr>
<td>removeObject (Object key, Object value) : boolean</td>
<td>If the map contains key and it is mapped to value then remove the entry, otherwise do nothing.</td>
</tr>
<tr>
<td>replace(K key, V value) : V</td>
<td>If the map contains key then replace with newValue, otherwise do nothing.</td>
</tr>
<tr>
<td>replace(K key, V oldValue, V newValue) : boolean</td>
<td>If the map contains key and it is mapped to oldValue then replace with newValue, otherwise do nothing.</td>
</tr>
</tbody>
</table>

There are two ConcurrentMap implementations available as shown in Table 4.

### Table 4: ConcurrentMap implementations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcurrentHashMap</td>
<td>ConcurrentHashMap provides two levels of internal hashing. The first level chooses an internal segment, and the second level hashes into buckets in the chosen segment. The first level provides concurrency by allowing reads and writes to occur safely on each segment in parallel.</td>
</tr>
<tr>
<td>ConcurrentSkipListMap</td>
<td>ConcurrentSkipListMap (added in Java SE 6) 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 aren’t modifying the same portion of the map.</td>
</tr>
</tbody>
</table>

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 Queue interface was added to java.util in Java SE 5 and while it can be used in single-threaded scenarios, it is primarily used with multiple producers or one or more consumers, all writing and reading from the same queue.

The BlockingQueue interface is in java.util.concurrent and 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. Table 5 demonstrates the Queue and BlockingQueue methods in terms of key operations and the strategy for dealing with these special conditions.

### Table 5: Queue and BlockingQueue methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Strategy</th>
<th>Insert</th>
<th>Remove</th>
<th>Examine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>throw exception</td>
<td>add</td>
<td>remove</td>
<td>element</td>
</tr>
<tr>
<td>Blocking Queue</td>
<td>block forever</td>
<td>put</td>
<td>take</td>
<td>n/a</td>
</tr>
<tr>
<td>Blocking Queue</td>
<td>block with timer</td>
<td>offer</td>
<td>poll</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Several Queue implementations are provided by the JDK and their relationships are described in Table 6.

### Table 6: Queue Implementations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue</td>
<td>PriorityQueue is the only non-concurrent queue implementation and can be used by a single thread to collect items and process them in a sorted order.</td>
</tr>
<tr>
<td>ConcurrentLinkedQueue</td>
<td>An unbounded linked list queue implementation and the only concurrent implementation not supporting BlockingQueue.</td>
</tr>
<tr>
<td>ArrayBlockingQueue</td>
<td>A bounded blocking queue backed by an array.</td>
</tr>
<tr>
<td>LinkedBlockingQueue</td>
<td>An optionally bounded blocking queue backed by a linked list. This is probably the most commonly used Queue implementation.</td>
</tr>
<tr>
<td>PriorityBlockingQueue</td>
<td>An unbounded blocking queue backed by a heap. Items are removed from the queue in an order based on the Comparator associated with the queue (instead of FIFO order).</td>
</tr>
<tr>
<td>DelayQueue</td>
<td>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.</td>
</tr>
<tr>
<td>SynchronousQueue</td>
<td>A D-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.</td>
</tr>
</tbody>
</table>

Deques
A double-ended queue or Deque (pronounced “deck”) was added in Java SE 6. Deques support not just adding from one end and removing from the other but adding and removing items from both ends. Similarly to BlockingQueue, there is a BlockingDeque interface that provides methods for blocking and timeout in the case of special conditions. Table 7 shows the Deque and BlockingDeque methods. Because Deque extends Queue and BlockingDeque extends BlockingQueue, all of those methods are also available for use.

### Table 7: Deque and BlockingDeque methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>First or Last</th>
<th>Strategy</th>
<th>Insert</th>
<th>Remove</th>
<th>Examine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>Head</td>
<td>throw exception</td>
<td>addFirst</td>
<td>removeFirst</td>
<td>getFirst</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>offerFirst</td>
<td>pollFirst</td>
<td>peekFirst</td>
</tr>
<tr>
<td></td>
<td>Tail</td>
<td>throw exception</td>
<td>addLast</td>
<td>removeLast</td>
<td>getlast</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>offerLast</td>
<td>pollLast</td>
<td>peekLast</td>
</tr>
<tr>
<td>Blocking</td>
<td>Head</td>
<td>block forever</td>
<td>putFirst</td>
<td>takeFirst</td>
<td>n/a</td>
</tr>
<tr>
<td>Queue</td>
<td></td>
<td>block with timer</td>
<td>offerFirst</td>
<td>pollFirst</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Tail</td>
<td>block forever</td>
<td>putLast</td>
<td>takeLast</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block with timer</td>
<td>offerLast</td>
<td>pollLast</td>
<td>n/a</td>
</tr>
</tbody>
</table>

One special use case for a Deque is when add, remove, and examine operations all take place on only one end of the pipe. This special case is just a stack (first-in-last-out retrieval order). The Deque interface actually provides methods that use the terminology of a stack: push(), pop(), and peek(). These
methods in the table 9: interaction across threads. Table 9 shows methods on Thread that can be used for direct one thread to directly call a method on another Thread object.

The most obvious way to communicate between threads is for In Java, the java.lang.Thread class is used to represent an application or JVM thread. Code is always being executed in the context of some Thread class (use Thread.currentThread() to obtain your own Thread).

Table 9: Thread coordination methods

<table>
<thead>
<tr>
<th>Thread Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Start a Thread instance and execute its run() method.</td>
</tr>
<tr>
<td>join</td>
<td>Block until the other Thread exits.</td>
</tr>
<tr>
<td>interrupt</td>
<td>Interrupt the other 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.</td>
</tr>
<tr>
<td>stop, suspend, resume, destroy</td>
<td>These methods are deprecated and should not be used. They perform dangerous operations depending on the state of the thread in question. Instead, use interrupt() or a volatile flag to indicate to a thread what it should do.</td>
</tr>
</tbody>
</table>

Uncaught exception handlers

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

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.

Many recent JVMs can detect monitor deadlocks and will print deadlock information in thread dump produced from a signal, jstack, or other thread dump tool.

In addition to deadlock, some other threading situations are starvation and livelock. Starvation occurs when threads hold a lock for long periods such that some threads “starve” without making progress. 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.

wait / notify

The wait / notify idiom is appropriate whenever one thread needs to signal to another that a condition has been met, especially as an alternative to sleeping in a loop and polling the condition. For example, one thread might wait for a queue to contain an item to process. Another thread can signal the waiting threads when an item is added to the queue.

The canonical usage pattern for wait and notify is as follows:

```java
class Latch {
    private final Object lock = new Object();
    private volatile boolean flag = false;
    public void waitTillChange() {
        synchronized(lock) {
            while(! flag) {
                try {
                    lock.wait();
                } catch(InterruptedException e) {
                }
            }
        }
    }
    public void change() {
        synchronized(lock) {
            flag = true;
            lock.notifyAll();
        }
    }
}
```

Some important things to note about this code:

- Always call wait, notify, and notifyAll inside a synchronized lock or an IllegalMonitorStateException will be thrown.
- 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 or notifyAll. Failing to do so will cause a notification but no thread will ever be able to escape its wait loop.

Condition

In Java SE 5, a new java.util.concurrent.locks.Condition class was added. Condition implements the wait/notify semantics in an API but with several additional features such as the ability to create multiple Conditions per Lock, interruptible waiting, access to statistics, etc. Conditions are obtained from a Lock instance as follows:

```java
class LatchCondition {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private volatile boolean flag = false;
    public void waitTillChange() {
        lock.lock();
        try {
            while(! flag) {
                condition.await();
            }
        } finally {
            lock.unlock();
        }
    }
    public void change() {
        lock.lock();
        try {
            condition.signal();
        } finally {
            lock.unlock();
        }
    }
}
```
Coordination classes
The java.util.concurrent package contains several classes pre-built for common forms of multi-thread communication. These coordination classes cover most common scenarios where wait/notify and Condition might be used and are strongly preferred for their safety and ease of use.

CyclicBarrier
The CyclicBarrier is initialized with a participant count. 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. The barrier can be reused indefinitely. Used to coordinate the start and stop of groups of threads.

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

Semaphore
A Semaphore manages a set of “permits” that can be checked out with acquire() which will block until one is available. Threads call release() to return the permit. A semaphore with one permit is equivalent to a mutual exclusion block.

Exchanger
An Exchanger waits for threads to meet at the exchange() method and swap values atomically. This is similar to using a SynchronousQueue but data values pass in both directions.

Callable and Future
A Callable is like the familiar Runnable but can return a result and throw an exception:

- V call() throws Exception;

It is common in the executor framework to submit a Callable and receive a Future. A Future is a marker representing a result that will be available at some point in the future. The Future has methods that allow you to either poll or block while waiting for the result to be ready. You can also cancel the task before or while it’s executing through methods on Future.

If you need the functionality of a Future where only Rungnables are supported (as in Executor), you can use FutureTask as a bridge. FutureTask implements both Future and Runnable so that you can submit the task as a Runnable and use the task itself as a Future in the caller.

ExecutorService implementations
The primary implementation of ExecutorService is ThreadPoolExecutor. This implementation class provides a wide variety of configurable features:

- Thread pool – specify “core” thread count (optionally pre-started), and max thread count
- Thread factory – generate threads with custom characteristics such as a custom name
- Work queue – specify the queue implementation, which must be blocking, but can be bounded or unbounded
- Rejected tasks – specify the policy for tasks that cannot be accepted due to a full input queue or unavailable worker
- Lifecycle hooks – overridden to extend to override key points in the lifecycle like before or after task execution
- Shutdown – stop incoming tasks and wait for executing tasks to complete

ScheduledThreadPoolExecutor is an extension of ThreadPoolExecutor that provides the ability to schedule tasks for completion rather than using FIFO semantics. For cases where java.util.Timer is not sophisticated enough, the ScheduledThreadPoolExecutor often provides sufficient flexibility.

The Executors class contains many static methods (see Table 10) for creating prepackaged ExecutorService and ScheduledExecutorService instances that will cover a wide variety of common use cases.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newSingleThreadExecutor</td>
<td>Returns an ExecutorService with exactly one thread</td>
</tr>
<tr>
<td>newFixedThreadPool</td>
<td>Returns an ExecutorService with a fixed number of threads</td>
</tr>
<tr>
<td>newCachedThreadPool</td>
<td>Returns an ExecutorService with a varying size thread pool</td>
</tr>
<tr>
<td>newSingleThreadScheduledExecutor</td>
<td>Returns a ScheduledExecutorService with a single thread</td>
</tr>
<tr>
<td>newScheduledThreadPool</td>
<td>Returns a ScheduledExecutorService with a core set of threads</td>
</tr>
</tbody>
</table>

Many concurrent Java programs need a pool of workers executing tasks from a queue. The java.util.concurrent package provides a solid foundation for this style of work management.

ExecutorService
The Executor and more expansive ExecutorService interfaces define the contract for a component that can execute tasks. Users of these interfaces can get a wide variety of implementation behaviors behind a common interface.

The most generic Executor interface accepts jobs only in the form of Rungnables:

- void execute(Runnable command)

The ExecutorService extends Executor to add methods that take both Rungnables and Callable task and collections of tasks:

- Future<?> submit(Runnable task)
- Future<T> submit(Callable<T> task)
- Future<T> submit(Runnable task, T result)
- List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
- List<Future<T>> invokeAny(Collection<? extends Callable<T>>, long timeout, TimeUnit unit)

Table 10: Executors factory methods
The following example creates a fixed thread pool and submits a long-running task to it:

```java
int processors = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(processors);
Future<Integer> futureResult = executor.submit(
    new Callable<Integer>() {
        public Integer call() {
            // Long running computation that returns an integer
        }
    });
Integer result = futureResult.get(); // block for result
```

In this example the call that submits the task to the executor will not block but return immediately. The last line will block on the `get()` call until the result is available. `ExecutorService` covers almost all situations where you would previously create Thread objects or thread pools. Any time your code is constructing a `Thread` directly, consider whether you could accomplish the same goal with an `ExecutorService`.

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()`. The number of available processors may change during the lifetime of a JVM.

### About the Author

**Alex Miller** is a Tech Lead with Terracotta Inc, the makers of the open-source Java clustering product Terracotta. Prior to Terracotta, Alex worked at BEA Systems and was Chief Architect at MetaMatrix. His interests include Java, concurrency, distributed systems, query languages, and software design. Alex enjoys tweeting as @puredanger and writing his blog at http://tech.puredanger.com and is a frequent speaker at user group meetings and conferences. In St. Louis, Alex is the founder of the Lambda Lounge group for the study of functional and dynamic languages and the Strange Loop developer conference.

## Recommended Book

*Java Concurrency in Practice* arms readers with both the theoretical underpinnings and concrete techniques for building reliable, scalable, maintainable concurrent applications. Rather than simply offering an inventory of concurrency APIs and mechanisms, it provides design rules, patterns, and mental models that make it easier to build concurrent programs that are both correct and performant.

### Buy Now

books.dzone.com/books/javaconcurrency

---

**Professional Cheat Sheets You Can Trust**

“Exactly what busy developers need: simple, short, and to the point.”

James Ward, Adobe Systems

### Upcoming Titles

- RichFaces
- Agile Software Development
- BIRT
- JSF 2.0
- Adobe AIR
- BPM&BPMN
- Flex 3 Components

### Most Popular

- Spring Configuration
- jQuery Selectors
- Windows Powershell
- Dependency Injection with EJB 3
- Netbeans IDE JavaEditor
- Getting Started with Eclipse
- Very First Steps in Flex

---

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. “DZone is a developer’s dream,” says PC Magazine.

---

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

---

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference:

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference:

---


Version 1.0