DZone Refcardz

CONTENTS INCLUDE:

Core Java Concurrency

By Alex Miller

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, Atomics, 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

Concept	Description
Java Memory Model	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, how to make fields immutable, and more.
Monitor	In Java, every object contains a "monitor" that can be used to provide mutual exlusion 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 exlusion, the monitor allows cooperation through the `wait` and `notify` operations.
Atomic field assignment	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.
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 are performed.
Data race	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.
Safe publication	It is unsafe to publish a reference to an object before construction of the object is complete. One way that the `this` 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.
Final fields	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.
Final fields, continued	At the end of construction, an object undergoes "final field freeze", which guarantees that if the object is safely published, all threads will see the values set during construction even in the absence of synchronization. Final field freeze includes not just the final fields in the object but also all objects reachable from those final fields.
Immutable objects	Final field semantics can be leveraged to create thread-safe immutable objects that can be shared and read without synchronization. To make an immutable object you should guarantee that: • The object is safely published (the `this` reference does not escape during construction) • All fields are declared final • Object reference fields must not allow modifications anywhere in the object graph reachable from the fields after construction. • The class should be declared final (to prevent a subclass from subverting these rules)

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.

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:

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

  public 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. Example of using an explicit `ReentrantReadWriteLock` to allow multiple concurrent readers:

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

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

  public int current() {
    lock.readLock().lock();
    try {
      return value;
    } finally {
      lock.readLock().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:

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

  public void stopProcessing() {
    stop = true;
  }

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

Hot Tip

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.

```
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.

```
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;
 }
}
```

CONCURRENT COLLECTIONS

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 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

Class	Description
`CopyOnWriteArraySet`	`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.
`CopyOnWriteArrayList`	Similar to `CopyOnWriteArraySet`, `CopyOnWriteArrayList` uses copy-on-write semantics to implement the List interface.
`ConcurrentSkipListSet`	`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.

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

Method	Description
`putIfAbsent(K key, V value) : V`	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.
`remove(Object key, Object value) : boolean`	If the map contains key and it is mapped to value then remove the entry, otherwise do nothing.
`replace(K key, V value) : V`	If the map contains key then replace with newValue, otherwise do nothing.
`replace(K key, V oldValue, V newValue) : boolean`	If the map contains key and it is mapped to oldValue then replace with newValue, otherwise do nothing.

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

Table 4: ConcurrentMap implementations

Method	Description
`ConcurrentHashMap`	`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.
`ConcurrentSkipListMap`	`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.

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

Method	Strategy	Insert	Remove	Examine
`Queue`	Throw Exception	add	remove	element
	Return special value	offer	poll	peek
`Blocking Queue`	Block forever	put	take	n/a
	Block with timer	offer	poll	n/a

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

Table 6: Queue Implementations

Method	Description
`PriorityQueue`	`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.
`ConcurrentLinkedQueue`	An unbounded linked list queue implementation and the only concurrent implementation not supporting `BlockingQueue.`
`ArrayBlockingQueue`	A bounded blocking queue backed by an array.
`LinkedBlockingQueue`	An optionally bounded blocking queue backed by a linked list. This is probably the most commonly used `Queue` implementation.
`PriorityBlockingQueue`	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).
`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.

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

Interface	First or Last	Strategy	Insert	Remove	Examine
Queue	Head	Throw exception	`addFirst`	`removeFirst`	`getFirst`
		Return special value	`offerFirst`	`pollFirst`	`peekFirst`
	Tail	Throw exception	`addLast`	`removeLast`	`getLast`
		Return special value	`offerLast`	`pollLast`	`peekLast`
`Blocking Queue`	Head	Block forever	`putFirst`	`takeFirst`	n/a
		Block with timer	`offerFirst`	`pollFirst`	n/a
	Tail	Block forever	`putLast`	`takeLast`	n/a
		Block with timer	`offerLast`	`pollLast`	n/a

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 map to `addFirst()`, `removeFirst()`, and `peekFirst()` methods in the `Deque` interface and allow you to use any `Deque` implementation as a stack. Table 8 describes the `Deque` and `BlockingDeque` implementations in the JDK. Note that `Deque` extends `Queue` and `BlockingDeque` extends `BlockingQueue`

Table 8: Deques

Class	Description
`LinkedList`	This long-standing data structure has been retrofitted in Java SE 6 to support the `Deque` interface. You can now use the standard Deque methods to add or remove from either end of the list (many of these methods already existed) and also use it as a non-synchronized stack in place of the fully synchronized Stack class.
`ArrayDeque`	This implementation is not concurrent and supports unbounded queue length (it resizes dynamically as needed).
`LinkedBlockingDeque`	The only concurrent deque implementation, this is a blocking optionally-bounded deque backed by a linked list.

THREADS

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`).

Thread Communication

The most obvious way to communicate between threads is for one thread to directly call a method on another Thread object. Table 9 shows methods on Thread that can be used for direct interaction across threads.

Table 9: Thread coordination methods

Thread Method	Description
`start`	Start a Thread instance and execute its run() method.
`join`	Block until the other Thread exits
`interrupt`	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.
`stop, suspend, resume, destroy`	These methods are all 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.

Uncaught exception handlers

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

```
Thread t = new Thread(runnable);
t.setUncaughtExceptionHandler(new Thread.
 UncaughtExceptionHandler() {
   void uncaughtException(Thread t, Throwable e) {
     // get Logger and log uncaught exception
   }
 });
t.start();
```

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 dumps 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.

THREAD COORDINATION

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:

```
public 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:

```
public 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 {
```

```
        flag = true;
        condition.signalAll();
      } 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 perferred 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.

TASK EXECUTION

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 `Runnables`:

- `void execute(Runnable command)`

The `ExecutorService` extends `Executor` to add methods that take both `Runnable` 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> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`
- `T invokeAny(Collection<? extends Callable<T>> tasks)`
- `T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`

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 `Runnables` 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 10: Executors factory methods

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.

The following example creates a fixed thread pool and submits a long-running task to it:

```
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` produced by one of the `Executor` factory methods; often this will be simpler and more flexible.

CompletionService

Beyond the common pattern of a pool of workers and an input queue, it is common for each task to produce a result that must be accumulated for further processing. The `CompletionService` interface allows a user to submit `Callable` and `Runnable` tasks but also to take or poll for results from the results queue:

- `Future<V> take() – take if available`
- `Future<V> poll() – block until available`
- `Future<V> poll(long timeout, TimeUnit unit) – block until timeout ends`

The `ExecutorCompletionService` is the standard implementation of `CompletionService`. It is constructed with an Executor that provides the input queue and worker thread pool.

Hot Tip 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

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

ISBN-13: 978-1-934238-62-2
ISBN-10: 1-934238-62-7
50795
9 781934 238622
$7.95

Version 1.0