**Concurrency**

**Processes and Threads**

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

## Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. Processes are often seen as synonymous with programs or applications

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

# Defining and Starting a Thread

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

* *Provide a Runnable object.* The [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the[HelloRunnable](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java) example:
* public class HelloRunnable implements Runnable {
* public void run() {
* System.out.println("Hello from a thread!");
* }
* public static void main(String args[]) {
* (new Thread(new HelloRunnable())).start();
* }
* }
* *Subclass Thread.* The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the [HelloThread](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java)example:
* public class HelloThread extends Thread {
* public void run() {
* System.out.println("Hello from a thread!");
* }
* public static void main(String args[]) {
* (new HelloThread()).start();
* }
* }

Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. This lesson focuses on the first approach, which separates the Runnable task from the Thread object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

InterruptedException is an exception that sleep throws when another thread interrupts the current thread while sleep is active.

**Difference between Thead.interrupted() and Thread.isInterrupted()**

Tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method.

Tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false

## The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. InvokingThread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static methodThread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

# Joins

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify

**Thread Interference**

Consider a simple class called [Counter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java)

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

Counter is designed so that each invocation of increment will add 1 to c, and each invocation of decrementwill subtract 1 from c. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of Counter to interleave, since both operations on c are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expressionc++ can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression c-- can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

**Synchronized Methods**

The Java programming language provides two basic synchronization idioms: *synchronized methods* and*synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the synchronized keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

* First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
* Second, when a synchronized method exits, it automatically establishes a happens-before relationship with*any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

## Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

public void addName(String name) {

synchronized(this) {

lastName = name;

nameCount++;

}

nameList.add(name);

}

## Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

# Atomic Access

Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads.

**Volatile:**

So what happens? Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in its own memory. The volatile keyword is used to say to the jvm "Warning, this variable may be modified in an other Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword could be used on every kind of variable, either primitive or objects! Maybe the subject of another article, more detailed...

# Deadlock

*Deadlock* describes a situation where two or more threads are blocked forever

## **Starvation**

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## **Livelock**

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

# Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed.

This can be achieved by calling wait() on the object. Which will be notified when another thread calls notify()

public synchronized void guardedJoy() {

// This guard only loops once for each special event, which may not

// be the event we're waiting for.

while(!joy) {

try {

wait();

} catch (InterruptedException e) {}

}

System.out.println("Joy and efficiency have been achieved!");

}

The other thread

public synchronized notifyJoy() {

joy = true;

notifyAll();

}

# Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed.

# A Strategy for Defining Immutable Objects

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   * Don't provide methods that modify the mutable objects.
   * Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

# High Level Concurrency Objects

# Lock Objects

More sophisticated locking idioms are supported by the [java.util.concurrent.locks](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html) package.

Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated [Condition](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html) objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Let's use Lock objects to solve the deadlock problem we saw in [Liveness](https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html). Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friendobjects must acquire locks for *both* participants before proceeding with the bow. Here is the source code for the improved model, [Safelock](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Safelock.java). To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.Random;

public class Safelock {

static class Friend {

private final String name;

private final Lock lock = new ReentrantLock();

public Friend(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public boolean impendingBow(Friend bower) {

Boolean myLock = false;

Boolean yourLock = false;

try {

myLock = lock.tryLock();

yourLock = bower.lock.tryLock();

} finally {

if (! (myLock && yourLock)) {

if (myLock) {

lock.unlock();

}

if (yourLock) {

bower.lock.unlock();

}

}

}

return myLock && yourLock;

}

public void bow(Friend bower) {

if (impendingBow(bower)) {

try {

System.out.format("%s: %s has"

+ " bowed to me!%n",

this.name, bower.getName());

bower.bowBack(this);

} finally {

lock.unlock();

bower.lock.unlock();

}

} else {

System.out.format("%s: %s started"

+ " to bow to me, but saw that"

+ " I was already bowing to"

+ " him.%n",

this.name, bower.getName());

}

}

public void bowBack(Friend bower) {

System.out.format("%s: %s has" +

" bowed back to me!%n",

this.name, bower.getName());

}

}

static class BowLoop implements Runnable {

private Friend bower;

private Friend bowee;

public BowLoop(Friend bower, Friend bowee) {

this.bower = bower;

this.bowee = bowee;

}

public void run() {

Random random = new Random();

for (;;) {

try {

Thread.sleep(random.nextInt(10));

} catch (InterruptedException e) {}

bowee.bow(bower);

}

}

}

public static void main(String[] args) {

final Friend alphonse =

new Friend("Alphonse");

final Friend gaston =

new Friend("Gaston");

new Thread(new BowLoop(alphonse, gaston)).start();

new Thread(new BowLoop(gaston, alphonse)).start();

}

}

# Executors

separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors.*

**Executor Interfaces**

The java.util.concurrent package defines three executor interfaces:

* Executor, a simple interface that supports launching new tasks.
* ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
* ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

**All Known Implementing Classes:**

[AbstractExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractExecutorService.html), [ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html), [ScheduledThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html), [ThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html)

## **The ExecutorService Interface**

The [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) interface supplements execute with a similar, but more versatile submit method. Likeexecute, submit accepts Runnable objects, but also accepts [Callable](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html) objects, which allow the task to return a value. The submit method returns a [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally,ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle [interrupts](https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html) correctly.

## **The ScheduledExecutorService Interface**

The [ScheduledExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html) interface supplements the methods of its parent ExecutorService withschedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

Ex:

|  |  |
| --- | --- |
| 01 | package com.journaldev.threadpool; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.util.concurrent.ExecutorService; |

|  |  |
| --- | --- |
| 04 | import java.util.concurrent.Executors; |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | public class SimpleThreadPool { |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | public static void main(String[] args) { |

|  |  |
| --- | --- |
| 09 | ExecutorService executor = Executors.newFixedThreadPool(5); |

|  |  |
| --- | --- |
| 10 | for (int i = 0; i < 10; i++) { |

|  |  |
| --- | --- |
| 11 | Runnable worker = new WorkerThread('' + i); |

|  |  |
| --- | --- |
| 12 | executor.execute(worker); |

|  |  |
| --- | --- |
| 13 | } |

|  |  |
| --- | --- |
| 14 | executor.shutdown(); |

|  |  |
| --- | --- |
| 15 | while (!executor.isTerminated()) { |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 | System.out.println('Finished all threads'); |

|  |  |
| --- | --- |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |

|  |  |
| --- | --- |
| 20 | } |

# Thread Pools

Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

# Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

## **Process:**

A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. Process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In employing multiple processes with a single CPU,context switching between various memory context is used. Each process has a complete set of its own variables.

## **Thread:**

A thread of execution results from a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. Example of threads in same process is automatic spell check and automatic saving of a file while writing. Threads are basically processes that run in the same memory context. Threads may share the same data while execution.

## **Task:**

A task is a set of program instructions that are loaded in memory.

# Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html) class, an extension of theAbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html) processes.

## **Blurring for Clarity**

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

public class ForkBlur extends RecursiveAction {

private int[] mSource;

private int mStart;

private int mLength;

private int[] mDestination;

// Processing window size; should be odd.

private int mBlurWidth = 15;

public ForkBlur(int[] src, int start, int length, int[] dst) {

mSource = src;

mStart = start;

mLength = length;

mDestination = dst;

}

protected void computeDirectly() {

int sidePixels = (mBlurWidth - 1) / 2;

for (int index = mStart; index < mStart + mLength; index++) {

// Calculate average.

float rt = 0, gt = 0, bt = 0;

for (int mi = -sidePixels; mi <= sidePixels; mi++) {

int mindex = Math.min(Math.max(mi + index, 0),

mSource.length - 1);

int pixel = mSource[mindex];

rt += (float)((pixel & 0x00ff0000) >> 16)

/ mBlurWidth;

gt += (float)((pixel & 0x0000ff00) >> 8)

/ mBlurWidth;

bt += (float)((pixel & 0x000000ff) >> 0)

/ mBlurWidth;

}

// Reassemble destination pixel.

int dpixel = (0xff000000 ) |

(((int)rt) << 16) |

(((int)gt) << 8) |

(((int)bt) << 0);

mDestination[index] = dpixel;

}

}

...

Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

protected static int sThreshold = 100000;

protected void compute() {

if (mLength < sThreshold) {

computeDirectly();

return;

}

int split = mLength / 2;

invokeAll(new ForkBlur(mSource, mStart, split, mDestination),

new ForkBlur(mSource, mStart + split, mLength - split,

mDestination));

}

If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in aForkJoinPool is straightforward, and involves the following steps:

1. Create a task that represents all of the work to be done.
2. // source image pixels are in src
3. // destination image pixels are in dst
4. ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
5. Create the ForkJoinPool that will run the task.
6. ForkJoinPool pool = new ForkJoinPool();
7. Run the task.

pool.invoke(fb);

## **Standard Implementations**

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the ForkBlur.java example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the [java.util.Arrays](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class for its parallelSort()methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of [Project Lambda](http://openjdk.java.net/projects/lambda/) scheduled for the Java SE 8 release. For more information, see the [Lambda Expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) section.

**Concurrent Collections**

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

* [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
* [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html) is a subinterface of [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html), which is a concurrent analog of [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html).
* [ConcurrentNavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is[ConcurrentSkipListMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), which is a concurrent analog of [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html).

The java.util.concurrent.ConcurrentNavigableMap class is a [**java.util.NavigableMap**](http://tutorials.jenkov.com/java-collections/navigablemap.html) with support for concurrent access, and which has concurrent access enabled for its submaps. The "submaps" are the maps returned by various methods like headMap(), subMap() and tailMap().

Rather than re-explain all methods found in the NavigableMap I will just look at the methods added byConcurrentNavigableMap.

All of these collections help avoid [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

# Atomic Variables

The [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomiccompareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the [Counter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java) class we originally used to demonstrate thread interference:

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

One way to make Counter safe from thread interference is to make its methods synchronized, as in[SynchronizedCounter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SynchronizedCounter.java):

class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the int field with an AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in [AtomicCounter](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/AtomicCounter.java):

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger c = new AtomicInteger(0);

public void increment() {

c.incrementAndGet();

}

public void decrement() {

c.decrementAndGet();

}

public int value() {

return c.get();

}

}

**Generating a random number between 2 numbers.**

Random r = new Random();

int Low = 10;

int High = 100;

int R = r.nextInt(High-Low) + Low;

# Concurrent Random Numbers

A random number generator isolated to the current thread. Like the global [Random](http://docs.oracle.com/javase/7/docs/api/java/util/Random.html) generator used by the [Math](http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html) class, a ThreadLocalRandom is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of ThreadLocalRandom rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention. Use of ThreadLocalRandom is particularly appropriate when multiple tasks (for example, each a [ForkJoinTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)) use random numbers in parallel in thread pools.

int r = ThreadLocalRandom.current() .nextInt(4, 77);