# Concurrent and Distributed Systems

Talk Eleven

Modern Concurrent Programming in Java

**Maxim Mozgovoy**

## Modern Instruments of Concurrency

Multicore machines and Internet/cloud systems became ubiquitous, so concurrent/parallel programming is more widespread than ever.

This led to the update of instruments,
used in everyday concurrent programming.

For example, let us have a look at newer Java concurrency tools.

Note: For Java 1 to 8, Java 1.x == Java x (e.g., Java 7 == Java 1.7)

# The Evolution of Concurrency in Java

| | |
|---|---|
| Java 1 (1996): | class Thread, interface Runnable |
| Java 2 (1998): | synchronized (thread-safe) collection classes, such as Vector, synchronizedMap… |
| Java 5 (2004): | Separate java.util.concurrent package with many new instruments, such as Executors, Lock objects, Atomic variables, and new thread-safe collections. |
| Java 7 (2011): | Improvements for java.util.concurrent package: Fork/Join framework, efficient multithreaded random number generator. |
| Java 9 (2017): | Reactive Streams (asynchronous stream processing) capabilities. |

## Third-Party Frameworks

There are also numerous third-party solutions, such as:

Apache Hadoop: a framework for data-intensive distributed development that implements a distributed file system, a job management engine, scheduler, etc.

GridGain: a framework, aimed at fast distributed data processing (claims to have much faster computations than Hadoop, but can work with Hadoop file system).

Akka: a toolkit that implements *Actors* (active objects), available for some other languages, such as Scala and Erlang.

## Java 2: Synchronized Collections (1)

Java 1.2 introduced a collections framework (JCF) with many collection classes (lists, sets, maps...)

JCF is aware of thread safety. What is a thread-safe collection:

`Collection c = new Collection();` // some collection

...

Thread 1: `c.remove(5);` // remove 5th element

Thread 2: `c.insert(3, 100);` // insert 100 at position 3

If simultaneous execution of `remove()` and `insert()` can cause troubles, the collection is **not** thread-safe.

# Java 2: Synchronized Collections (2)

The basic principle of thread safety:
declare each collection method as **synchronized**:

```
class Collection {
   synchronized public void put(int index,
                                Object o)    { … }
   synchronized public Object get(int index) { … }
   …
}
```

Some Java collections are thread-safe (Vector, Hashtable),
some are not (ArrayList, TreeMap).

Thread safety requires computational resources,
so thread-safe collections work slower.

# Java 2: Synchronized Collections (3)

All collections from `java.util` package except `Vector` and `Hashtable` are not thread-safe.

You can convert a non-thread safe collection into a thread-safe collection with `synchronizedName()` helper functions:

```
import java.util.*;

List sl =
  Collections.synchronizedList(new ArrayList());
Map sm =
  Collections.synchronizedMap(new HashMap());

// also available:
synchronizedCollection(), synchronizedSet(),
synchronizedSortedMap(), synchronizedSortedSet()
```

## Java 5: Concurrent Collections

Java 5 implements several collections designed specifically for concurrent programming, like `ArrayBlockingQueue` (useful for master-worker pattern) or `ConcurrentHashMap`.

These collection provide **much** better performance for multi-threaded applications, so you should use them if possible.

Performance is achieved with fine-grained locking strategy: independent locks for different data blocks are used, which prevents complete collection locking in most cases.

However, there are also important differences in interface and behavior of these classes, so please consult documentation.

## Useful Methods of Thread Class

| | |
|---|---|
| `join()` | waits until the thread has finished work |
| `sleep(long ms)` | waits for `ms` milliseconds |
| `interrupt()` | requests thread interruption |

Some methods are deprecated and should not be used:

`stop()`, `suspend()`, `resume()`

Note: `interrupt()` method does not interrupt a thread automatically: it delivers an interruption request to the thread, but it is the thread's responsibility to handle this request.

## Locks: More Synchronization Options (1)

Recall the traditional way to do synchronization:

```
Object mutex = new Object();

class MyThread implements Runnable {
     public void run() {
          synchronized(mutex) {
               …
          }
     }
}
```

Two different threads cannot enter
the synchronized block at the same time.

## Locks: More Synchronization Options (2)

Mutex is a very simple tool. A more advanced alternative is Lock interface of Java 5 (see, e.g., ReentrantLock class)
A big advantage of Lock is its ability to try locking the section before you proceed:

```
synchronized(mutex)  {  // old version: if the
     …                  // thread cannot go here,
}                       // it will be suspended

import java.util.concurrent.locks.*;  // new version
Lock lock = new ReentrantLock();
if(lock.tryLock()) {          // Success!  The section is locked,
       …                      // do your actions, then
       lock.unlock();         // release the lock
} else { /* cannot establish the lock, do something else */ }
```

## Locks: More Synchronization Options (3)

Of course, Lock can work in the same way as synchronized():

```
Lock lock = new ReentrantLock();
…
lock.lock();      // locked: other threads will be suspended
f();              // do stuff here
lock.unlock();
```

This is the same as:

```
Object lock = new Object();
…
synchronized(lock) { f(); }
```

## Locks: More Synchronization Options (4)

Practical guidelines suggest releasing locks in `finally` clause:

```
Lock lock = new ReentrantLock();
…
lock.lock();
try {
    f();                  // do stuff here
}
finally {                 // ensure that lock is released
    lock.unlock();        // even if f() throws an exception
}
```

## Locks: More Synchronization Options (5)

Function tryLock() can help to avoid deadlocks and livelocks.

Suppose we want to enter the block

synchronized(m) { … },

but we know that there is a chance of deadlock/livelock.

With tryLock() we can try first, and if the section is already blocked, do something else
(ask the user to wait, select another strategy, etc.)

## Executors (1)

Executors provide a more convenient way of managing threads than Thread / Runnable.

Executors take care of thread creation and invocation, thus allowing to write less code (with fewer errors), and obtain some hi-level capabilities.

Consider a simple example: *a prime-finding factory*.

The task is to retrieve prime numbers located between the two given numbers.

Let's see how to solve the task with an executor.

## Executors (2)

```
import java.util.concurrent.*;
import java.util.*;

// class PrimeExec
// convenience function: return true if n is prime

static private boolean isPrime(long n)   {
  for(long i = 2; i <= Math.sqrt(n); ++i)
    if(n % i == 0)
      return false;
    return true;
}
```

## Executors (3)

// Callable<X> is a modern alternative to Runnable interface

// X is a return type (it isn’t possible to return values with Runnable)

// class FindPrimes corresponds to a single thread

// (it is a modern version of class FindPrimes implements Runnable)

```
static class FindPrimes implements
                                 Callable<Vector> {
  private final long from, to; // borders
                                 // to be searched
  public FindPrimes(long from_, long to_)
                      { from = from_; to = to_; }
```

# Executors (4)

// class FindPrimes (cont.)

// call() method is the thread's main algorithm

// (similar to Thread/Runnable's void run() method)

// If your class implements Callable<X>, call() should be defined as

// `public X call() { … }`

```
  public Vector call()  {
     Vector v = new Vector();
     for(long i = from; i <= to; ++i)
       if(isPrime(i))
         v.add(i);
     return v;           // return found primes
  }
} // end of class FindPrimes declaration
```

## Executors (5)

Now the class FindPrimes can find all the primes between the given borders. Let's call it with an executor:

```
public static void main(String args[])
                              throws Exception {
  // list all the tasks to be performed
  FindPrimes[] tasks = new
      FindPrimes[]{ new FindPrimes(3, 100),
                    new FindPrimes(500, 1000),
                    new FindPrimes(5000, 9000)};

  // the tasks will be executed
  // in the specified order!
  ...
```

## Executors (6)

```
...
// create an executor
// (use at most 2 concurrent threads)
ExecutorService executor =
              Executors.newFixedThreadPool(2);

// run all tasks and collect results
List results =
      executor.invokeAll(Arrays.asList(tasks));

// wait until all the tasks are finished
executor.shutdown();

... // here print the results
```

# Executors (7)

// each call() returns a Vector of long numbers

// however, executor gets these results wrapped into the objects

// of type Future (so results is the List of Future objects)

// use Future.get() to retrieve the wrapped value

```
   for(Future r : (List<Future>)results) {
         Vector v = (Vector)r.get();
               for(Object o : v)
                     System.out.print(o + " ");
   }
}     // end of main()
```

printout: 3 5 7 11 13 17 19 23 29 31 37 41 …

# Executors (8)

Executors provide two main advantages:

1) They decouple task submission from task execution, which adds flexibility to program architecture.

2) Thread pools actually reuse previously created threads, so the cost of thread creation and teardown is reduced.

## Fork/Join (1)

One of the trends in concurrent programming is to make parallelization as invisible to the programmer as possible.

General idea: you program the algorithm, and the system tries to parallelize it automatically.

The code should be designed in a special “parallelizable” way. (Example: MapReduce framework)

Java 7 implements a similar idea via Fork/Join framework.

## Fork/Join (2)

A typical structure of Fork/Join program looks like this:

```
Solve(Problem P) {
        if(P is small)
                Solve it
        else
                Divide P into pieces P1, P2, ..., $P_N$
                Invoke Solve(P1), Solve(P2), ..., Solve($P_N$)
                Combine results
}
```

The Fork/Join framework runs different Solve() calls in different threads, thus parallelizing the program!

# Fork/Join (3)

By default Fork/Join runs as many threads
as many cores/processors the current computer has.

So it doesn't overload the machine.

Let's modify our prime-finding program to use Fork/Join:

```
import java.util.concurrent.*;
import java.util.*;

class PrimeFork {
    static private boolean isPrime(long n)
    { /* same as before */ }
```

## Fork/Join (4)

// the main algorithm should be declared in a class that extends
// RecursiveAction or RecursiveTask<X> if result of type X is needed

```
static class FindPrimes extends RecursiveAction {
  private final long from, to;
  private Vector results;
  public FindPrimes(long from_, long to_)
              { from = from_; to = to_;
                results = new Vector(); }
  public Vector Results() { return results; }

  // the main method should be named compute()
  protected void compute() { … }
 ...
```

## Fork/Join (5)

```
  // "small" tasks contain less than 10 numbers

  protected void compute() {
    if(to - from + 1 < 10) solveSmall();
    else                   solveBig();
  }

  private void solveSmall() {
    for(long i = from; i <= to; ++i)
      if(isPrime(i))
        results.add(i);
}
```

## Fork/Join (6)

```
  private void solveBig() {
    long midpoint = from + (to - from) / 2;
    FindPrimes fp1 = new                    // left half
               FindPrimes(from, midpoint);
    FindPrimes fp2 = new                    // right half
               FindPrimes(midpoint + 1, to);

    invokeAll(fp1, fp2); // solve both problems
    results.addAll(fp1.Results()); // combine the
    results.addAll(fp2.Results()); // results
  }
} // end class FindPrimes
```

## Fork/Join (7)

Using this code in main() is simple:

```
FindPrimes p = new FindPrimes(1, 1000);
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(p);

for(Object o : p.Results())
  System.out.print(o + " ");
```

Printout: 2 3 5 7 11 13 17 19 23 29 31 37 41 …

## Atomic Variables (1)

Let’s return to the problem of non-atomicity:

Suppose that Integer S = 0 is a global variable.

If <u>Thread 1</u> executes S += 5 and <u>Thread 2</u> executes S += 10, the resulting value of S may differ from 15.

(“Money” example).

This happens because += operation can be complex (non-atomic), and there can be some interference between the threads, concurrently executing +=.

# Atomic Variables (2)

Java ≥ 5 provides several data types with operations that are guaranteed to be atomic.

(See package java.util.concurrent.atomic)

For example, the type AtomicInteger implements the following methods:

```
int addAndGet(int delta)   // add delta, return result
int incrementAndGet()      // add one, return result
int get()                  // return the stored value
void set(int newVal)       // set new value
```

…

## Conclusions

- Notable additions have been made to the practice of C&D programming in the past decade.

- Some are still experimental, while others are widely accepted (and became, e.g., a part of Java standard).

- New Java additions include:
  - Synchronized and concurrent (thread-safe) collections.
  - Better locking mechanisms.
  - Executors (thread managers).
  - Fork/Join framework for automated parallelism.
  - Atomic variables.