Java Technologies
Lecture VII

Valdas Rapševičius
Vilnius University
Faculty of Mathematics and Informatics

2015.05.18
Session Objectives

• Concurrency is a “must to know” for well-grounded developer
• Basics
  – ProcessBuilder
  – Thread
  – Synchronization
• java.util.concurrent package
  – Atomic
  – Locks
  – Concurrent Collections
  – Callable
  – Advanced Methods
Multithreaded Applications

From Benjamin J Evans, Martijn Verburg. The Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming
Concurrent programming, there are two basic units of execution: processes and threads.

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. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.
- Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object.

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.
- Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads, as we'll demonstrate in the next section.
java.lang.ProcessBuilder

• This class is used to create operating system processes.

• Each `ProcessBuilder` instance manages a collection of process attributes:
  – command, a list of strings which signifies the external program file to be invoked and its arguments, if any
  – an environment, which is a system-dependent mapping from variables to values
  – working directory
  – source of standard input
  – destination for standard output and standard error
  – `redirectErrorStream` property

• `start()` method
  – creates a new `Process` instance with those attributes
  – can be invoked repeatedly from the same instance to create new sub-processes with identical or related attributes
ProcessBuilder example

Starting a new process which uses the default working directory and environment is easy:

```java
Process p = new ProcessBuilder("myCommand", "myArg").start();
```

Here is an example that starts a process with a modified working directory and environment, and redirects standard output and error to be appended to a log file:

```java
ProcessBuilder pb = new ProcessBuilder("myCommand", "myArg1", "myArg2");

Map<String, String> env = pb.environment();
env.put("VAR1", "myValue");
env.remove("OTHERVAR");
env.put("VAR2", env.get("VAR1") + "suffix");

pb.directory(new File("myDir"));
File log = new File("log");
pb.redirectErrorStream(true);
pb.redirectOutput(Redirect.appendTo(log));

Process p = pb.start();
```
Java Threading Model

- Shared, visible-by-default mutable state
- Pre-emptive thread scheduling

**To consider:**

- Objects can be easily shared between all threads within a process.
- Objects can be changed (“mutated”) by any threads that have a reference to them.
- The thread scheduler can swap threads on and off cores at any time, more or less.
- Methods must be able to be swapped out while they’re running.
- Objects can be locked to protect vulnerable data.

**java.util.concurrent (J2SE 5+) was developed to achieve:**

- Safety (also known as concurrent type safety)
- Liveness
- Performance
- Reusability
A thread is a thread of execution in a program

- JVM allows an application to have multiple threads of execution running concurrently
- Every thread has a priority
- Each thread may or may not also be marked as a daemon
- Every thread has a name for identification purposes
  - More than one thread may have the same name
  - If a name is not specified when a thread is created, a new name is generated for it.

At JVM start up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class)

Above continues to execute until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place

- All threads that are not daemon threads have died
  - either by returning from the call to the run method or
  - by throwing an exception that propagates beyond the run method.
Thread States

Java:

```java
Object.notify();
Object.notifyAll();
```

- **Ready to run**
  - start()
  - Chosen by scheduler
  - Scheduler swap

- **Running (executing)**
  - Data/sync received
  - Another thread closes socket
  - Done

- **Sleeping**
  - Thread.sleep()

- **Waiting**
  - Thread.wait()
Declaration

@RequiredArgsConstructor
class PrimeThread extends Thread {

    private final long minPrime;

    @Override
    public void run() {
        ...
    }

}

Execution

PrimeThread p = new PrimeThread(143);
p.start();
Runnable interface

Declaration

@RequiredArgsConstructor
class PrimeRun implements Runnable {

    long minPrime;

    public void run() {
        ...
    }
}

Execution

PrimeRun p = new PrimeRun(143);
new Thread(p).start();
Thread Groups

• An interesting functionality offered by the concurrency API of Java is the ability to group the threads

• This allows us to treat the threads of a group as a single unit and provides access to the `Thread` objects that belong to a group to do an operation with them.
  
  – For example, you have some threads doing the same task and you want to control them, irrespective of how many threads are still running, the status of each one will interrupt all of them with a single call.

• **Java provides the `ThreadGroup` class to work with groups of threads.** A `ThreadGroup` object can be formed by `Thread` objects and by another `ThreadGroup` object, generating a tree structure of threads.
printInfo(Thread.currentThread());
Thread t = new Thread(() -> printInfo(Thread.currentThread()));
printInfo(t);

private static void printInfo(final Thread t) {
    System.out.format("Id = %d\n", t.getId());
    System.out.format("Name = %s\n", t.getName());
    System.out.format("Priority = %d\n", t.getPriority());
    System.out.format("State = %s\n", t.getState().name());
    System.out.format("Group name = %s\n", t.getThreadGroup().getName());
    System.out.format("Alive = %s\n", t.isAlive());
    System.out.format("Daemon = %s\n", t.isDaemon());
    System.out.format("Interrupted = %s\n", t.isInterrupted());
}
Thread: sleep

Current thread execution can be suspended…

```java
public class SleepMessages {

    public static void main(String args[]) throws InterruptedException {

        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too" };

        for (int i = 0; i < importantInfo.length; i++) {

            //Pause for 4 seconds
            Thread.sleep(4000);

            //Print a message
            System.out.println(importantInfo[i]);

        }
    }
}
```
Thread: interrupt

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.

```java
thread.interrupt();
```

- A thread sends an interrupt by invoking `interrupt` on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

```java
try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    // We've been interrupted: no more work!
    return;
}

if (Thread.interrupted()) {
    // We've been interrupted: no more fun.
    return;
}
```
The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

```java
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.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.
MT Programming

• Can take advantage of multiprocessor hardware
• Shared memory!
  – Data is in flux
    • unless it is read-only, thread local, or protected by a lock
  – Locks are essential
  – Deadlocks
  – Races

\[
i = i + 1\quad \text{MOV EAX, [i]}
\]
\[
\text{INC EAX}
\]
\[
\text{MOV [i], EAX}
\]

• Application behavior is usually nondeterministic
• Code coverage alone is insufficient; threads racing to access memory cause the most problematic bugs
Locks

• Lock is a synchronization mechanism
  – enforces a mutual exclusion concurrency control policy
• Each Object instance is a Lock!
• General recommendations
  – use `unlockAll` instead of `notify` if you expect more than one thread is waiting for lock
  – `lock` and `unlock` methods must be called in synchronized context
  – Always call `lock` method in loop because if multiple threads are waiting for lock and one of them got lock and reset the condition and other thread needs to check the condition after they got wake up to see whether they need to wait again or can start processing
  – use same object for calling `wait()` and `notify()` method, every object has its own lock so calling `wait()` on `objectA` and `notify()` on `object B` will not make any sense.
Objects: wait() and notify()

```java
o.wait()
o.wait(long timeout)
o.wait(long timeout, int nanos)
```

Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

```java
o.notify()
o.notifyAll()
```

Wakes up a single or all threads that are waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
• The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*.

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

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

• Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```java
public void addName(String name) {
    synchronized(o) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```
• **volatile** keyword since Java 1.0
• To avoid race conditions!
• Simple way of synchronization of object fields, including primitives
  – The value seen by a thread is always reread from main memory before use.
  – Any value written by a thread is always flushed through to main memory before the instruction completes.
• No locks!

```java
private static volatile Singleton instance;
public static Singleton getInstance(){
    if(instance == null){
        synchronized(Singleton.class) {
            if(instance == null) instance = new Singleton();
        }
    }
    return instance;
}
```

• Use volatile
  – to read and write long and double variable atomically as both are 64 bit data type and by default writing is not atomic and platform dependence.
  – as an alternative way of achieving synchronization in Java in some cases, like Visibility.
  – to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment.
  – to fix double checked locking in Singleton pattern.
Double-Checked Locking

// DCL problem: resource can be null! Volatile does not help (pre 5)

class SomeClass {
    private Resource resource = null;
    public Resource getResource() {
        if (resource == null) {
            synchronized {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}

// New approach

private static class LazySomethingHolder {
    public static Something something = new Something();
}

public static Something getInstance() {
    return LazySomethingHolder.something;
}
Fully synchronized class

```java
public class ExampleTimingNode {

    private final String i;
    private final Map<Update, Long> x =
        new HashMap<>();

    public ExampleTimingNode(String i) {
        this.i = i;
    }

    public synchronized String getI () {
        return i;
    }

    public synchronized void do(Update upd) {
        long ct = System.currentTimeMillis();
        x.put(upd, ct);
    }

    public synchronized boolean conf(Update upd) {
        Long tr = x.get(upd);
        return tr != null;
    }
}
```

- All fields are always initialized to a consistent state in every constructor.
- There are no public fields.
- Object instances are guaranteed to be consistent after returning from any non-private method (assuming the state was consistent when the method was called)
- All methods provably terminate in bounded time
- All methods are synchronized
- There is no calling of another instance’s methods while in an inconsistent state
- There is no calling of any non-private method while in an inconsistent state
• A part of JSR133 (Java 5+)
• The Java Memory Model describes
  – what behaviors are legal in multithreaded code
  – how threads may interact through memory
  – the relationship between variables in a program and the low-level details of storing and retrieving them to and from memory or registers in a real computer system
• Java language constructs, including volatile, final, and synchronized
• Incorrectly synchronized code (aka Data Race)
  • there is a write of a variable by one thread,
  • there is a read of the same variable by another thread and
  • the write and read are not ordered by synchronization
• Relationships between blocks of code (since Java 5, Edges)
  – Happens-Before: that one block of code fully completes before the other can start
  – Synchronizes-With: an action will synchronize its view of an object with main memory before continuing
• java.util.concurrent
• java.util.concurrent.atomic
• java.util.concurrent.locks

• Usage Motivation:
  – Reduced programming effort
  – Increased performance
  – Increased reliability
  – Improved maintainability
  – Increased productivity

• Patterns:
  – CountDownLatch
  – CyclicBarrier

• Execution control
  – Future interface and FutureTask class
  – Executors

• Fork/join (J7)
• Several classes that have names starting with Atomic
• Same semantics as a volatile, but wrapped in a class API that includes atomic (meaning all-or-nothing) methods for suitable operations
• Simple way to avoid race conditions on shared data

```java
private final AtomicLong sequenceNumber = new AtomicLong(0);
public long nextId() {
    return sequenceNumber.getAndIncrement();
}
```
Add different types of locks (such as reader and writer locks)
Not restrict locks to blocks (allow a lock in one method and unlock in another)
If a thread cannot acquire a lock, allow the thread to back out or carry on or do something else—a try-Lock() method.
Allow a thread to attempt to acquire a lock and give up after a certain amount of time.

Implementation:

- ReentrantLock — aka lock() in synchronized blocks but more flexible
- ReentrantReadWriteLock — better performance for many readers but few writers

Simple example:

```java
private final Lock lock = new ReentrantLock();
public void doUpdate() {
    lock.lock();
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```
public void doUpdate() {

    boolean acquired = false;
    while (!acquired) {
        try {
            acquired = lock.tryLock(TTW, TimeUnit.MILLISECONDS);
            if (acquired) {
                // Do the job
                ...
            } else {
                Thread.sleep(wait);
            }
        } catch (InterruptedException e) {
            e.printStackTrace(System.err);
        } finally {
            if (acquired) lock.unlock();
        }
    }
}
Concurrent Collections

- **BlockingQueue**
  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
  - TransferQueue
    producers may wait for consumers to receive elements
  - DelayQueue
    Delayed elements, in which an element can only be taken when its delay has expired
  - SynchronousQueue
    Each insert operation must wait for a corresponding remove operation by another thread, and vice versa

- **BlockingDeque**
  Supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element

- **ConcurrentMap**
  is a subinterface of java.util.Map that defines useful atomic operations
  - ConcurrentNavigableMap
    subinterface of ConcurrentMap that supports approximate matches The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap

All of these collections help avoid Memory Consistency errors 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.
Executors

• Executes submitted Runnable tasks
  – Decouples task submission from the mechanics of how each task will be run
  – Normally should be used instead of explicitly creating threads.

• Classes
  – Obtain from static constructors in Executors class
  – ThreadPoolExecutor
    • Executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.
    • Provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead
    • Provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks
    • Maintains some basic statistics, such as the number of completed tasks
  – ScheduledThreadPoolExecutor
    • Can additionally schedule commands to run after a given delay, or to execute periodically
    • Preferable to Timer when multiple worker threads are needed, or when the additional flexibility or capabilities of ThreadPoolExecutor (which this class extends) are required.
    • No real-time guarantees about when, after they are enabled, they will commence.
    • Tasks scheduled for exactly the same execution time are enabled in first-in-first-out (FIFO) order of submission.
final MyObject out = getSampleObject();

Callable<String> cb = new Callable<String>() {
    public String call() throws Exception {
        return out.toString();
    }
};

String s = cb.call();
Future and FutureTask

• Represents an asynchronous task
  – Future – interface (see example below)
  – FutureTask – class that accepts Callable

• Main methods:
  – get(): gets the result. If the result isn’t yet available, get() will block until it is. There’s a version that takes a timeout
  – cancel(): allows the computation to be cancelled before completion.
  – isDone(): allows the caller to determine whether the computation has finished.

```java
Future<Long> fut = getNthPrime(1_000_000_000);
Long result = null;

while (result == null) {
    try {
        result = fut.get(60, TimeUnit.SECONDS);
    } catch (TimeoutException tox) { }
    System.out.println("Still not found...");
}
System.out.println("Found: " + result.longValue());
```
@AllArgsConstructor
public class FutureExample implements Callable<String> {

    private final int x;
    private final int y;

    @Override
    public String call() throws Exception {
        // do some job
    }

    public static void main(String[] args) throws Exception {
        FutureTask<String> ft = new FutureTask<>(new FutureExample(10, 12));
        Executor ex = Executors.newSingleThreadExecutor();
        ex.execute(ft);

        // Do some job in parallel
        System.out.println(ft.get());
    }
}
CountDownLatch class

- One or more threads waits until a set of operations being performed in other threads completes.
  - CountDownLatch is initialized with a given count.
  - await methods block until the current count reaches zero due to invocations of the countDown() method
  - all waiting threads are released and any subsequent invocations of await return immediately.
- Example:

```java
void main() throws InterruptedException {
    CountDownLatch doneSignal = new CountDownLatch(N);
    Executor e = ...;
    for (int i = 0; i < N; ++i) // create and start threads
        e.execute(new WorkerRunnable(doneSignal, i));
    doneSignal.await(); // wait for all to finish
}

class WorkerRunnable implements Runnable {

    private final CountDownLatch doneSignal;

    WorkerRunnable(CountDownLatch doneSignal, int i) {
        this.doneSignal = doneSignal;
    }

    public void run() {
        try {
            ...
            doneSignal.countDown();
        } catch (InterruptedException ex) {} // return;
    }
}
```
CyclicBarrier class

Allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. Cyclic – because it can be reused!

```java
final int N;
final CyclicBarrier barrier;

public void doJob() {
    barrier = new CyclicBarrier(N, () -> mergeRows(...));
    for (int i = 0; i < N; ++i)
        new Runnable() {
            public void run() {
                while (!done()) {
                    processRow(...);
                    try {
                        barrier.await();
                    } catch (InterruptedException ex) {
                        return;
                    } catch (BrokenBarrierException ex) {
                        return;
                    }
                }
            }
        }
}
```
Fork/Join Framework

• Alternative names
  – Divide and conquer
  – Small & large tasks
  – if (work is small) doit else split-work and wait for results

• Automatic scheduling of tasks on a thread pool
  – Can the problem’s subtasks work without explicit cooperation or synchronization between the subtasks?
  – Do the subtasks calculate some value from their data without altering it (“pure” functions)?
  – Is divide-and-conquer natural for the subtasks?

• Main objects
  – ForkJoinPool
  – ForkJoinTask<V> (superclass of RecursiveTask<V> and RecursiveAction)
  – RecursiveTask<V>
  – RecursiveAction
class Sum extends RecursiveTask<Long> {

    Sum(int[] arr, int lo, int hi) { ... }

    protected Long compute() {
        if(high - low <= SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for(int i=low; i < high; ++i)  sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left  = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns  = left.join();
            return leftAns + rightAns;
        }
    }
}

ForkJoinPool fjPool = new ForkJoinPool();
int sum = fjPool.invoke(new Sum(array,0,array.length));
Conclusions (Recommendations)

• By Priority
  – High Level Concurrency Abstractions
    • JSR-000166 Concurrency Utilities and java.util.concurrent
  – Low Level locking
    • synchronized() blocks and java.util.concurrent.locks
  – Low Level Utilities
    • volatile, atomic classes
  – Data Races: deliberate undersynchronization

• Document concurrency

• Reduce sync costs
  – Avoid sharing mutable objects
  – Avoid old collection classes
  – Use bulk I/O (or NIO)

• Avoid lock contention
  – Reduce lock scopes
  – Reduce lock duration