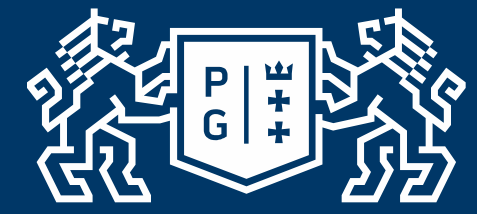

POLITECHNIKA
GDAŃSKA

Java High Level Concurrency Objects

Paweł Kaczmarek, Jan Cychnerski

2017

POLITECHNIKA GDAŃSKA

WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

java.util.concurrent package (since version 5.0)

- **Lock** objects support locking idioms that simplify many concurrent applications.
- **Executors** define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
- **Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- **Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.
- **ThreadLocalRandom** provides efficient generation of pseudorandom numbers from multiple threads.

Lock vs synchronized

- **synchronized** easier to use, but has limitations
- **Lock** - analogous to **synchronized**, but
  - ability to back out of an attempt to acquire a lock
    - **tryLock**
  - backs out if another thread sends an interrupt before the lock is acquired
    - **lockInterruptibly**
- demo

Executor

- There's a close connection between the task being done by a new thread, as defined by its **Runnable** object, and the thread itself, as defined by a **Thread** object
  - works well for small applications
  - separate thread management and creation from the rest of the application in large-scale applications
  - Objects that encapsulate these functions are known as executors.
- Interface hierarchy
- **Executor, ExecutorService, ScheduledExecutorService**
- Implementing classes: **ForkJoinPool, ScheduledThreadPoolExecutor, ThreadPoolExecutor**

Executor interfaces

- **Executor**
  - **void execute(Runnable command)**
- **ExecutorService**
  - **<T> Future<T> submit(Callable<T> task)**
  - Submits a value-returning task for execution and returns a Future representing the pending results of the task.
  - **invokeAll**, **invokeAny**, **isTerminated**, ...
- **ScheduledExecutorService**
  - impl class **ScheduledThreadPoolExecutor**
  - **schedule**, **scheduleAtFixedRate**, ...

Executors helper class

- Static methods that create and return
  - **ExecutorService** (**ScheduledExecutorService**) set up with commonly useful configuration settings
  - "wrapped" **ExecutorService**
  - **ThreadFactory** that sets newly created threads to a known state
- Examples
  - **newCachedThreadPool()** - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
  - **newFixedThreadPool(int nThreads)** - Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

Thread pools

- Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads.
- E.g. factory methods in Executors
- **newFixedThreadPool, newCachedThreadPool, newSingleThreadExecutor**
- Demo

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.
- **ForkJoinPool** class
  - implements the core work-stealing algorithm
- **ForkJoinTask** (e.g. **RecursiveTask**)
  - for tasks that run within a **ForkJoinPool**
  - is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a **ForkJoinPool**, at the price of some usage limitations.
- Demo

Atomic Variables

- trivial operations on atomic variables - not thread safe
- using synchronized
  - for more complicated classes, we might want to avoid the liveness impact of unnecessary synchronization
- java.util.concurrent.atomic
  - defines classes that support atomic operations on single variables
  - e.g. **AtomicInteger**
- Demo

Volatile and atomicity

- Volatile variables in the Java language can be thought of as "synchronized lite"
  - less coding than synchronized
  - do a subset of the things that synchronized
- Volatile variables share the visibility features of synchronized, but none of the atomicity features.
  - threads will automatically see the most up-to-date value for volatile variables
  - volatile alone is not strong enough to implement a counter, a mutex, or any class that has invariants that relate multiple variables (not suitable for x++)

Volatile and atomicity

- Conditions for using volatile
  - Writes to the variable do not depend on its current value.
  - The variable does not participate in invariants with other variables.
- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).
- Exemplary usages
  - status flags, one-time safe publication,

Atomic Variables compared to volatile

- A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form:
  - boolean compareAndSet(expectedValue, updateValue);
- provide access and updates to a single variable of the corresponding type

POLITECHNIKA GDAŃSKA

WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Concurrent Collections

- BlockingQueue
- ConcurrentMap
- ConcurrentNavigableMap

BlockingQueue (interface)

- 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
- Classes
  - **ArrayBlockingQueue** (with defined capacity)
  - **DelayQueue** (unbounded, an element can only be taken when its delay has expired)
  - other
- Implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.
- The bulk Collection operations **addAll**, **containsAll**, **retainAll** and **removeAll** are not necessarily performed atomically

BlockingQueue (interface)

- Methods come in four forms, with different ways of handling operations that cannot be satisfied immediately
  - throws an exception - add, remove, element (get but don't remove)
  - returns a special value (either null or false) - offer, poll, peek
  - blocks the current thread indefinitely until the operation can succeed - put, take
  - blocks for only a given maximum time limit - offer (e, time, unit), poll (time, unit)

ConcurrentMap

- A subinterface of java.util.Map that defines useful atomic operations
- A Map providing thread safety and atomicity guarantees.
- ConcurrentHashMap - standard general-purpose implementation (a concurrent analog of HashMap).
  - retrieval operations do not entail locking
  - there is not any support for locking the entire table in a way that prevents all access

ConcurrentNavigableMap

- extends ConcurrentMap<K,V>, NavigableMap<K,V>
  - A subinterface of ConcurrentMap that supports approximate matches
- NavigableMap
  - methods returning the closest matches for given search targets
  - Methods lowerEntry, floorEntry, ceilingEntry, and higherEntry return Map.Entry objects associated with keys respectively
  - firstEntry, pollFirstEntry, lastEntry, and pollLastEntry that return and/or remove the least and greatest mappings, if any exist, else returning null
- public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V> implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable
  - concurrent analog of TreeMap

ThreadLocalRandom

- for applications that expect to use random numbers from multiple threads or ForkJoinTasks
- less contention and, ultimately, better performance compared to Math.random()
- int r = ThreadLocalRandom.current().nextInt(1, 100)
  - nextInt(int origin, int bound) - returns a pseudorandom int value between the specified origin (inclusive) and the specified bound (exclusive)

Bibliography

- http://www.ibm.com/developerworks/java/library/j-jtp06197/index.html
- Oracle Java (8 SE) Tutorial

POLITECHNIKA
GDAŃSKA

HISTORIA MĄDROŚCIĄ
PRZYSZŁOŚĆ WYZWANIEM