ABSTRACT

Thoughtcloud.It is a social thinking platform that intends to bring people together by the way they think. The concept came into picture by a thought about connecting people who are thinking about the exact same thing at the same time.ThoughtCloud.It relies its core strength on -Preciseness - Restricted to three words long thoughts only.Lucidity - Easy and interactive design makes it a child's play to use it. Innovation - Innovative feature of real time thought mapping by demography. Thought Cloud c:geo is a simple to use but powerful geocaching client with a lot of additional features. All you need to get started is an account on geocaching.com. Find caches using the live map or by using one of the many search functions. Navigate to a cache or a waypoint of a cache with the built-in compass function, the map or hand over the coordinates to various external apps (e.g. Radar, Google Navigation, StreetView, Locus, Navigon, Sygic and many more). Store cache information to your device directly from geocaching.com as well as via GPX file import to have it available whenever you want. You can manage your stored caches in different lists and can sort and filter them according to your needs Stored caches together with offline map files or static maps can be used to find caches without an internet connection (e.g. when roaming). Logs can be posted online or stored offline for later submission or exported via field notes.Search and discover trackables, manage your trackable inventory and drop a trackable while posting a cache log.

Keywords : Keyword1, Keyword2, Keyword3

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.2 Background

XXXXXXX

1.3 Need

1.3.1 Advantages

Figure 1.1: My figure

XXXXXXX
[12pt,a4paper]report
[utf8]inputenc
[top=30mm, bottom=25mm, left=35mm,right=20mm]geometry a4paper
graphicx
booktabs array paralist verbatim subfig
fancyhdr 1
sectsty
[nottoc,notlof,notlot]tocbibind [titles,subfigure]tocloft

Chapter 2

Scheduling

2.1 Executors

Executor is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous IO, and lightweight task frameworks. Depending on which concrete Executor class is being used, tasks may execute in a newly created thread, an existing task-execution thread, or the thread calling execute, and may execute sequentially or concurrently. ExecutorService provides a more complete asynchronous task execution framework. An ExecutorService manages queuing and scheduling of tasks, and allows controlled shutdown. The ScheduledExecutorService subinterface and associated interfaces add support for delayed and periodic task execution. ExecutorServices provide methods arranging asynchronous execution of any function expressed as Callable, the result-bearing analog of Runnable. A Future returns the results of a function, allows determination of whether execution has completed, and provides a means to cancel execution. A RunnableFuture is a Future that possesses a run method that upon execution, sets its results.Implementations. Classes ThreadPoolExecutor and ScheduledThreadPoolExecutor provide tunable, flexible thread pools. The Executors class provides factory methods for the most common kinds and configurations of Executors, as well as a few utility methods for using them. Other utilities based on Executors include the concrete class FutureTask providing a common extensible implementation of Futures, and ExecutorCompletionService, that assists in coordinating the processing of groups of asynchronous tasks.

2.1.1 Queues

The ConcurrentLinkedQueue class supplies an efficient scalable thread-safe non-blocking FIFO queue. Five implementations in java.util.concurrent sup-

port the extended Blocking Queue interface, that defines blocking versions of put and take: Linked Blocking Queue, Array Blocking Queue, Synchronous Queue, Priority Blocking Queue, and Delay Queue.The different classes cover the most common usage contexts for producer-consumer, messaging, parallel tasking, and related concurrent designs.The Blocking Deque interface extends BlockingQueue to support both FIFO and LIFO (stack-based) operations. Class Linked Blocking Deque provides an implementation.

2.1.2 Timing

The TimeUnit class provides multiple granularities (including nanoseconds) for specifying and controlling time-out based operations. Most classes in the package contain operations based on time-outs in addition to indefinite waits. In all cases that time-outs are used, the time-out specifies the minimum time that the method should wait before indicating that it timed-out. Implementations make a ”best effort” to detect time-outs as soon as possible after they occur. However, an indefinite amount of time may elapse between a time-out being detected and a thread actually executing again after that time-out. All methods that accept timeout parameters treat values less than or equal to zero to mean not to wait at all. To wait ”forever”, you can use a value of Long.MAX$_{V}ALUE$.

2.1.3 Synchronizers

Besides Queues, this package supplies Collection implementations designed for use in multithreaded contexts: ConcurrentHashMap, Concurrent Skip ListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, and CopyOnWriteArraySet. When many threads are expected to access a given collection, a ConcurrentHashMap is normally preferable to a synchronized HashMap, and a ConcurrentSkipListMap is normally preferable to a synchronized TreeMap. A CopyOnWriteArrayList is preferable to a synchronized ArrayList when the expected number of reads and traversals greatly outnumber the number of updates to a list.The ”Concurrent” prefix used with some classes in this package is a shorthand indicating several differences from similar ”synchronized” classes. For example java.util.Hashtable and Collections.synchronizedMap(new HashMap()) are synchronized. But ConcurrentHashMap is ”concurrent”. A concurrent collection is thread-safe, but not governed by a single exclusion lock. In the particular case of ConcurrentHashMap, it safely permits any number of concurrent reads as well as a tunable number of concurrent writes.

”Synchronized” classes can be useful when you need to prevent all access to a collection via a single lock, at the expense of poorer scalability. In other cases in which multiple threads are expected to access a common collection, ”concurrent” versions are normally preferable. And unsynchronized collections are preferable when either collections are unshared, or are accessible only when holding other locks.Most concurrent Collection implementations (including most Queues) also differ from the usual java.util conventions in that their Iterators provide weakly consistent rather than fast-fail traversal. A weakly consistent iterator is thread-safe, but does not necessarily freeze the collection while iterating, so it may (or may not) reflect any updates since the iterator was created.

2.2 Memory Consistency Properties

The synchronized and volatile constructs, as well as the Thread.start() and Thread.join() methods, can form happens-before relationships. In particular: Each action in a thread happens-before every action in that thread that comes later in the program’s order.An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.A call to start on a thread happens-before any action in the started thread.All actions in a thread happen-before any other thread successfully returns from a join on that thread.The methods of all classes in java.util.concurrent and its subpackages extend these guarantees to higher-level synchronization. In particular:Actions in a thread prior to placing an object into any concurrent collection happen-before actions subsequent to the access or removal of that element from the collection in another thread.Actions in a thread prior to the submission of a Runnable to an Executor happen-before its execution begins. Similarly for Callables submitted to an ExecutorService.Actions taken by the asynchronous computation represented by a Future happen-before actions subsequent to the retrieval of the result via Future.get() in another thread.Actions prior to ”releasing” synchronizer methods such as Lock.unlock, Semaphore.release, and CountDownLatch.countDown happen-

before actions subsequent to a successful ”acquiring” method such as Lock.lock, Semaphore.acquire, Condition.await, and CountDownLatch.await on the same synchronizer object in another thread.For each pair of threads that successfully exchange objects via an Exchanger, actions prior to the exchange() in each thread happen-before those subsequent to the corresponding exchange() in another thread.Actions prior to calling CyclicBarrier.await and Phaser.awaitAdvance (as well as its variants) happen-before actions performed by the barrier action, and actions performed by the barrier action happen-before actions subsequent to a successful return from the corresponding await in other threads.

2.2.1 A subsection

More text.