Building blocks (delegating thread safety to existing thread safe classes)

Synchronized collections

Source of magic - **Collections.synchronizedXxx** factory methods Encapsulates the state of collection and sycnrhonizes all methods accessing state.

Problems of synchronized collections
* Returned iterators use modification count 1) are fail-fast, throw ConcurrentModificationException 2) no sync for counts, may see stale data
* That's the reason you May need
Additional client-side locking to guard compound actions (iteration, navigation, conditional operations)
* Client side locking for compound actions may hurt performance
* Client side locking for compound actions may lead to high contention and hurt scalability
* Client side locking *acting* on elements may lead to deadlock (*acting* may lock on smth else)
Copying (may require syncrhonization as well), has own performance penalty
* Also, there are hidden iterators

Concurrent collections

Replacing synchronized collections with concurrent offers dramatic scalability improvements with little risk.
Examples:

- ConcurrentHashMap, CopyOnWriteArrayList
- BlockingQueue, ConcurrentLinkedQueue, PriorityQueue
- ConcurrentSkipListMap, ConcurrentSkipListSet

ConcurrentHashMap

Uses a finer-grained locking mechanism called lock striping to allow a greater degree of shared access.

- many readers can acceess list concurrently
- readers may access map concurrently with writers
- a limited number of threads can access map concurrently
- some atomic operation: putIfAbsent, removeIfEqual, replaceIfEqual
- Iterators are weakly consistent:
  tolerates concurrent modification
  traverses elements as they existed when the iterator was constructed
  (not guaranteed) reflects modification to the collection after the construction of iterator

Disadvantages:

- size() may return an approximation instead of an exact count - weakened semantics to improve performance for the most important operations: get, put, containsKey, remove
- not possible to lock the whole collection for an atomic operation (possible for synchronized collections) (reasonable tradeoff)

CopyOnWriteArraylist

What is it

- Derives thread safety from the fact that properly constructed and safely published effectively immutable object is thread safe (reading operations)
- Mutability is implemented by creating and republishing a new copy.

Eliminates doing anything to safely iterate collection

- the need to copy collection
- the need to synchronize

Blocking queues, producer-consumer pattern

Methods:

- blocking **put** (blocks on full queue) and **get** (blocks on empty queue)
- timed **offer** and **poll**

Queues:

- bounded
- unbounded (can never be full)

Producer-consumer patterh

Consumers place data onto the queue as it is available, producers retrieve data when they are ready

- Removes code dependencies between consumer and producer (types, amount, the presence)
- And at the same time it simplifies workload management
- Sufficient internal syncrhonization to facilitate serial thread confinement for handing off ownership from consumers to producers
  Publishing safely
  Only 1 consumer will access it
  Producer thread does not access it after hand off

Implemenations:

- LinkedBlockingQueue, ArrayBlockingQueue - FIFO queues
- PriorityBlockingQueues - other than FIFO order

- SinchronousQueue
  What is it
  * No storage for elements
  * Maintains list of queued threads waiting to enqueue or dequeue an element
  Advantages
  * Reduces latency for moving data
  * Feeds back more information about the state of the task to the producer
  * No storage capacity

Work stealing pattern

Each consumer has it's own deque. When a consumer exhausts the work in its own deque, it steals the work from other deques, from the side other than is accessed by the owner of that deque. It's more scalable than producer-consumer:

- Reduces contention accessing single queue
- Reduces contention accessing deque of another thread
- Fits when consumers are producers of the same units of work that can be published back to the deque

Implementations:

- ArrayDeque, LinkedBlockingDeque

Blocking and interruptible methods

Reasons threads block: waiting for IO completion, waiting to acquire lock, waiting to wake up from Thread.sleep, waiting for result of a computation in another Thread.

When a thread blocks, it has on of 3 states:

- BLOCKED
- WAITING
- TIMED_WAITING

When the external event completes, the thread is placed back in the RUNNABLE state.

Blocking can't be stopped but can be interrupted. Interruption is cooperative

- Initiate interruption. Call t.interrupt() to set interrupt flag of thred t.
- Handle interruption. If your method **m** invokes method that throws InterruptedException then **m** is blocking too.
  - Propagate the **InterruptedException**. Or catch it and rethrow after cleaning up.
  - Restore the interrupt after catching **InterruptedException**.

Don't just catch InterruptedException and do nothing after it. This deprices code higher the stack. Allowed only if you're extending Thread class.

Synchronizers

A synchronizer is any object that coordinates the control flow of threads based on its state. They all have properties:

- encapsulate state that determines whether threads arriving at the synchronizer should be allowed to pass or forced to wait
- provide methods to manipulate that state
- provide methods to wait efficitnetly for the synchronizer to enter the desired state

Latches

Delays the progress of threads until it reaches its terminal state. Ensures that some activities do not proceed until other one-time activities (events) complete.

- Computation does not proceed until resources it needs have been initialized.
- Ensuring that a service does not start until other services on which it depends have started, and that no other services that depend on it do not start until it is.
- All parties included in activity are ready to proceed.

Implementation: CountDownLatch.

FutureTask

- A computation represented by a FutureTask is implemented with Callable, it can have one of three state: waiting to run, runnning, completed.
- **get()** blocks if computation not ready or return the result (safe publication is guaranteed)
- Throws **ExecutionException** which wraps a **Throwable**, that is: a **checked exception**, a **RuntimeException** or an **Error**.

Semaphores

Controls the number of activities that can access a certain resource or perform the certain action at the same time (resource pools, bounds on collections).

- Manages a set of virtual permits
- The initial number of permits is passed to the Semaphore constructor.
- Activities **acquire** permits and **release** when they are done with them.
- If no permits is available **acquire** blocks until one is.

Degenerate case is a binary semaphore (initial count equal to 1). A binary semaphore can be used as a mutex with non-reentrant semantics.

Barriers

For waiting an envet of many threads waiting at the same barrier point at the same time.

Differences from latches:

- Waiting for a specific event — other threads
- Reusable, reinitializes itself when barrier achieved

Implementation, CyclicBarrier. Wait with **await** method, blocks until everybody arrived:

- When all threads meet at a barrier point, the barrier is passed, threads released, barrier is reset and can be used again
  await returns unique index for each thread, can be used to elect a leader that takes some special action in the next iteration
  barrier action may be passed to constructor, a **Runnable** that is executed when the barrier is successfully passed and before the blocked threads are released
- If a call to **await** method times out or a thread blocked is interrupted, the barrier is considered broken, await calls terminate with **BrokenBarrieriException**.