Concurrent utilities

Concurrent utilities typically provide thread safety, high scalability, or perhaps both.

Be sure which type of behavior a utility provides and use it appropriately.

Prefer library provided utilities over home-brewed ones; it's far too easy to make a small mistake, and debugging concurrency problems is exceptionally hard, at least in part because they tend to be non-deterministic (not repeatable) in nature.

Atomic types

Atomic types provide indivisible read-modify-write cycles at a library API level.

The package `java.util.concurrent.atomic` is "lock free" which ais to provide very high scalability.

A few sample operations on `AtomicInteger` illustrates general concepts. Most create a *happens-before* relationship as though the variable were `volatile`:

- `get()` : returns the current value
- `addAndGet(int x)` : add `x` to the current value, and return the updated value.
- decrementAndGet() : reduce value by 1 and return the result.

Atomic array types

Provide atomic operations along with *happens-before* relationships on array elements.

The array may be created with a length, or by duplicating an existing array.

Example operation:

```
int accumulateAndGet(int i, int x, IntBinaryOperator accFn)
```

Atomically updates `array[i]` with the result of `accFn.apply(array[i], x)`

Accumulators

If a value is subject to concurrent updates, but very rare writes, and the update operations are entirely independent of one another (for example, simple increments) then it's possible to gain scalability by giving each thread a thread-local variable, so it can have unrestricted updates.

On reading the multiple values must be collected, aggregated, and returned.

Initialize an accumulator with a binary operator suited to the data type. This is used to apply the updates.

The operation should be commutative, associative, and free of side-effects, or the behavior of the accumulator will likely be unpredictable.

Concurrent data structures

These data structures provide thread safety, but are primarily focused on scalability, minimizing or eliminating locks in their operation.

`ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`

ConcurrentHashMap locks only one "bucket" of the map rather than the entire map.

When significant contention is expected concurrent data structures are preferred, for non-contented concurrent situations, synchronized structures are likely better. Entirely single-threaded access favors normal structures.

CopyOnWriteArrayList

This structure is intended for heavily concurrent reading with occasional updates.

Read operations are lock free, however writing is accomplished in two steps:

- Read and duplicate the entire array (this is clearly expensive)
- Modify the copy
- Redirect all reads to the new version allowing the old array to be garbage collectable.

Synchronized structures

If contention is rarely expected, a simple lock is likely the best approach.

The `Collections` class provides static factory methods for proxy objects that wrap around the main interfaces of the collections API, such that access through that proxy is serialized using simple locking.

E.g.

```
List<String> syncList =
  Collections.synchronizedList(new ArrayList<String>());
```