Caches

Example

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .removalListener(MY_LISTENER)
        .build(
            new CacheLoader<Key, Graph>() {
              @Override
              public Graph load(Key key) throws AnyException {
                return createExpensiveGraph(key);
              }
            });
```

Applicability

Caches are tremendously useful in a wide variety of use cases. For example, you should consider using caches when a value is expensive to compute or retrieve, and you will need its value on a certain input more than once.

A `Cache` is similar to `ConcurrentMap`, but not quite the same. The most fundamental difference is that a `ConcurrentMap` persists all elements that are added to it until they are explicitly removed. A `Cache` on the other hand is generally configured to evict entries automatically, in order to constrain its memory footprint. In some cases a `LoadingCache` can be useful even if it doesn't evict entries, due to its automatic cache loading.

Generally, the Guava caching utilities are applicable whenever:

- You are willing to spend some memory to improve speed.
- You expect that keys will sometimes get queried more than once.
- Your cache will not need to store more data than what would fit in RAM. (Guava caches are **local** to a single run of your application. They do not store data in files, or on outside servers. If this does not fit your needs, consider a tool like Memcached.)

If each of these apply to your use case, then the Guava caching utilities could be right for you!

Obtaining a `Cache` is done using the `CacheBuilder` builder pattern as demonstrated by the example code above, but customizing your cache is the interesting part.

Note: If you do not need the features of a `Cache`, `ConcurrentHashMap` is more memory-efficient – but it is extremely difficult or impossible to duplicate most `Cache` features with any old `ConcurrentMap`.

Population

The first question to ask yourself about your cache is: is there some *sensible default* function to load or compute a value associated with a key? If so, you should use a `CacheLoader`. If not, or if you need to override the default, but you still want atomic "get-if-absent-compute" semantics, you should pass a `Callable` into a `get` call. Elements can be inserted directly, using `Cache.put`, but automatic cache loading is preferred as it makes it easier to reason about consistency across all cached content.

From a CacheLoader A `LoadingCache` is a `Cache` built with an attached `CacheLoader`. Creating a `CacheLoader` is typically as easy as implementing the method `V load(K key) throws Exception`. So, for example, you could create a `LoadingCache` with the following code:

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .maximumSize(1000)
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) throws AnyException {
               return createExpensiveGraph(key);
             }
           });


...
try {
  return graphs.get(key);
} catch (ExecutionException e) {
  throw new OtherException(e.getCause());
}
```

The canonical way to query a `LoadingCache` is with the method `get(K)`. This will either return an already cached value, or else use the cache's `CacheLoader` to atomically load a new value into the cache. Because `CacheLoader` might throw an `Exception`, `LoadingCache.get(K)` throws `ExecutionException`. (If the cache loader throws an *unchecked* exception, `get(K)` will throw an `UncheckedExecutionException` wrapping it.) You can also choose to use `getUnchecked(K)`, which wraps all exceptions in `UncheckedExecutionException`, but this may lead to surprising behavior if the underlying `CacheLoader` would normally throw checked exceptions.

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .expireAfterAccess(10, TimeUnit.MINUTES)
       .build(
           new CacheLoader<Key, Graph>() {
             public Graph load(Key key) { // no checked exception
               return createExpensiveGraph(key);
```

```
          }
        });

...
return graphs.getUnchecked(key);
```

Bulk lookups can be performed with the method `getAll(Iterable<? extends K>)`. By default, `getAll` will issue a a separate call to `CacheLoader.load` for each key which is absent from the cache. When bulk retrieval is more efficient than many individual lookups, you can override `CacheLoader.loadAll` to exploit this. The performance of `getAll(Iterable)` will improve accordingly.

Note that you can write a `CacheLoader.loadAll` implementation that loads values for keys that were not specifically requested. For example, if computing the value of any key from some group gives you the value for all keys in the group, `loadAll` might load the rest of the group at the same time.

From a Callable All Guava caches, loading or not, support the method `get(K, Callable<V>)`. This method returns the value associated with the key in the cache, or computes it from the specified `Callable` and adds it to the cache. No observable state associated with this cache is modified until loading completes. This method provides a simple substitute for the conventional "if cached, return; otherwise create, cache and return" pattern.

```
Cache<Key, Value> cache = CacheBuilder.newBuilder()
    .maximumSize(1000)
    .build(); // look Ma, no CacheLoader
...
try {
  // If the key wasn't in the "easy to compute" group, we need to
  // do things the hard way.
  cache.get(key, new Callable<Value>() {
    @Override
    public Value call() throws AnyException {
      return doThingsTheHardWay(key);
    }
  });
} catch (ExecutionException e) {
  throw new OtherException(e.getCause());
}
```

Inserted Directly Values may be inserted into the cache directly with `cache.put(key, value)`. This overwrites any previous entry in the cache for the specified key. Changes can also be made to a cache using any of the `ConcurrentMap` methods exposed by the `Cache.asMap()` view. Note that no method on the `asMap` view will ever cause entries to be automatically loaded into the cache. Further, the atomic operations on that view operate outside the scope

of automatic cache loading, so `Cache.get(K, Callable<V>)` should always be preferred over `Cache.asMap().putIfAbsent()` in caches which load values using either `CacheLoader` or `Callable`. Note that `Cache.get(K, Callable)` may also insert values into the underlying cache.

Eviction

The cold hard reality is that we almost *certainly* don't have enough memory to cache everything we could cache. You must decide: when is it not worth keeping a cache entry? Guava provides three basic types of eviction: size-based eviction, time-based eviction, and reference-based eviction.

Size-based Eviction

If your cache should not grow beyond a certain size, just use `CacheBuilder.maximumSize(long)`. The cache will try to evict entries that haven't been used recently or very often. *Warning*: the cache may evict entries before this limit is exceeded – typically when the cache size is approaching the limit.

Alternately, if different cache entries have different "weights" – for example, if your cache values have radically different memory footprints – you may specify a weight function with `CacheBuilder.weigher(Weigher)` and a maximum cache weight with `CacheBuilder.maximumWeight(long)`. In addition to the same caveats as `maximumSize` requires, be aware that weights are computed at entry creation time, and are static thereafter.

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumWeight(100000)
        .weigher(new Weigher<Key, Graph>() {
           public int weigh(Key k, Graph g) {
             return g.vertices().size();
           }
         })
        .build(
            new CacheLoader<Key, Graph>() {
              public Graph load(Key key) { // no checked exception
                return createExpensiveGraph(key);
              }
            });
```

Timed Eviction

`CacheBuilder` provides two approaches to timed eviction:

- `expireAfterAccess(long, TimeUnit)` Only expire entries after the specified duration has passed since the entry was last accessed by a read or a write. Note that the order in which entries are evicted will be similar to that of size-based eviction.

- `expireAfterWrite(long, TimeUnit)` Expire entries after the specified duration has passed since the entry was created, or the most recent replacement of the value. This could be desirable if cached data grows stale after a certain amount of time.

Timed expiration is performed with periodic maintenance during writes and occasionally during reads, as discussed below.

Testing Timed Eviction Testing timed eviction doesn't have to be painful... and doesn't actually have to take you two seconds to test a two-second expiration. Use the Ticker interface and the `CacheBuilder.ticker(Ticker)` method to specify a time source in your cache builder, rather than having to wait for the system clock.

Reference-based Eviction

Guava allows you to set up your cache to allow the garbage collection of entries, by using weak references for keys or values, and by using soft references for values.

- `CacheBuilder.weakKeys()` stores keys using weak references. This allows entries to be garbage-collected if there are no other (strong or soft) references to the keys. Since garbage collection depends only on identity equality, this causes the whole cache to use identity (`==`) equality to compare keys, instead of `equals()`.
- `CacheBuilder.weakValues()` stores values using weak references. This allows entries to be garbage-collected if there are no other (strong or soft) references to the values. Since garbage collection depends only on identity equality, this causes the whole cache to use identity (`==`) equality to compare values, instead of `equals()`.
- `CacheBuilder.softValues()` wraps values in soft references. Softly referenced objects are garbage-collected in a globally least-recently-used manner, *in response to memory demand*. Because of the performance implications of using soft references, we generally recommend using the more predictable maximum cache size instead. Use of `softValues()` will cause values to be compared using identity (`==`) equality instead of `equals()`.

Explicit Removals

At any time, you may explicitly invalidate cache entries rather than waiting for entries to be evicted. This can be done:

- individually, using `Cache.invalidate(key)`
- in bulk, using `Cache.invalidateAll(keys)`
- to all entries, using `Cache.invalidateAll()`

Removal Listeners

You may specify a removal listener for your cache to perform some operation when an entry is removed, via `CacheBuilder.removalListener(RemovalListener)`. The `RemovalListener` gets passed a `RemovalNotification`, which specifies the `RemovalCause`, key, and value.

Note that any exceptions thrown by the `RemovalListener` are logged (using `Logger`) and swallowed.

```
CacheLoader<Key, DatabaseConnection> loader = new CacheLoader<Key, DatabaseConnection> () {
  public DatabaseConnection load(Key key) throws Exception {
    return openConnection(key);
  }
};
RemovalListener<Key, DatabaseConnection> removalListener = new RemovalListener<Key, Database
  public void onRemoval(RemovalNotification<Key, DatabaseConnection> removal) {
    DatabaseConnection conn = removal.getValue();
    conn.close(); // tear down properly
  }
};

return CacheBuilder.newBuilder()
  .expireAfterWrite(2, TimeUnit.MINUTES)
  .removalListener(removalListener)
  .build(loader);
```

Warning: removal listener operations are executed synchronously by default, and since cache maintenance is normally performed during normal cache operations, expensive removal listeners can slow down normal cache function! If you have an expensive removal listener, use `RemovalListeners.asynchronous(RemovalListener, Executor)` to decorate a `RemovalListener` to operate asynchronously.

When Does Cleanup Happen?

Caches built with `CacheBuilder` do *not* perform cleanup and evict values "automatically," or instantly after a value expires, or anything of the sort. Instead, it performs small amounts of maintenance during write operations, or during occasional read operations if writes are rare.

The reason for this is as follows: if we wanted to perform `Cache` maintenance continuously, we would need to create a thread, and its operations would be competing with user operations for shared locks. Additionally, some environments restrict the creation of threads, which would make `CacheBuilder` unusable in that environment.

Instead, we put the choice in your hands. If your cache is high-throughput, then you don't have to worry about performing cache maintenance to clean up expired

entries and the like. If your cache does writes only rarely and you don't want cleanup to block cache reads, you may wish to create your own maintenance thread that calls `Cache.cleanUp()` at regular intervals.

If you want to schedule regular cache maintenance for a cache which only rarely has writes, just schedule the maintenance using `ScheduledExecutorService`.

Refresh

Refreshing is not quite the same as eviction. As specified in `LoadingCache.refresh(K)`, refreshing a key loads a new value for the key, possibly asynchronously. The old value (if any) is still returned while the key is being refreshed, in contrast to eviction, which forces retrievals to wait until the value is loaded anew.

If an exception is thrown while refreshing, the old value is kept, and the exception is logged and swallowed.

A `CacheLoader` may specify smart behavior to use on a refresh by overriding `CacheLoader.reload(K, V)`, which allows you to use the old value in computing the new value.

```
// Some keys don't need refreshing, and we want refreshes to be done asynchronously.
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .refreshAfterWrite(1, TimeUnit.MINUTES)
        .build(
            new CacheLoader<Key, Graph>() {
              public Graph load(Key key) { // no checked exception
                return getGraphFromDatabase(key);
              }

              public ListenableFuture<Graph> reload(final Key key, Graph prevGraph) {
                if (neverNeedsRefresh(key)) {
                  return Futures.immediateFuture(prevGraph);
                } else {
                  // asynchronous!
                  ListenableFutureTask<Graph> task = ListenableFutureTask.create(new Callable
                    public Graph call() {
                      return getGraphFromDatabase(key);
                    }
                  });
                  executor.execute(task);
                  return task;
                }
              }
            });
```

Automatically timed refreshing can be added to a cache using `CacheBuilder.refreshAfterWrite(long,`

`TimeUnit)`. In contrast to `expireAfterWrite`, `refreshAfterWrite` will make a key *eligible* for refresh after the specified duration, but a refresh will only be actually initiated when the entry is queried. (If `CacheLoader.reload` is implemented to be asynchronous, then the query will not be slowed down by the refresh.) So, for example, you can specify both `refreshAfterWrite` and `expireAfterWrite` on the same cache, so that the expiration timer on an entry isn't blindly reset whenever an entry becomes eligible for a refresh, so if an entry isn't queried after it comes eligible for refreshing, it is allowed to expire.

Features

Statistics

By using `CacheBuilder.recordStats()`, you can turn on statistics collection for Guava caches. The `Cache.stats()` method returns a `CacheStats` object, which provides statistics such as

- `hitRate()`, which returns the ratio of hits to requests
- `averageLoadPenalty()`, the average time spent loading new values, in nanoseconds
- `evictionCount()`, the number of cache evictions

and many more statistics besides. These statistics are critical in cache tuning, and we advise keeping an eye on these statistics in performance-critical applications.

asMap

You can view any `Cache` as a `ConcurrentMap` using its `asMap` view, but how the `asMap` view interacts with the `Cache` requires some explanation.

- `cache.asMap()` contains all entries that are *currently loaded* in the cache. So, for example, `cache.asMap().keySet()` contains all the currently loaded keys.
- `asMap().get(key)` is essentially equivalent to `cache.getIfPresent(key)`, and never causes values to be loaded. This is consistent with the `Map` contract.
- Access time is reset by all cache read and write operations (including `Cache.asMap().get(Object)` and `Cache.asMap().put(K, V)`), but not by `containsKey(Object)`, nor by operations on the collection-views of `Cache.asMap()`. So, for example, iterating through `cache.asMap().entrySet()` does not reset access time for the entries you retrieve.

Interruption

Loading methods (like `get`) never throw `InterruptedException`. We could have designed these methods to support `InterruptedException`, but our support

would have been incomplete, forcing its costs on all users but its benefits on only some. For details, read on.

`get` calls that request uncached values fall into two broad categories: those that load the value and those that await another thread's in-progress load. The two differ in our ability to support interruption. The easy case is waiting for another thread's in-progress load: Here we could enter an interruptible wait. The hard case is loading the value ourselves. Here we're at the mercy of the user-supplied `CacheLoader`. If it happens to support interruption, we can support interruption; if not, we can't.

So why not support interruption when the supplied `CacheLoader` does? In a sense, we do (but see below): If the `CacheLoader` throws `InterruptedException`, all `get` calls for the key will return promptly (just as with any other exception). Plus, `get` will restore the interrupt bit in the loading thread. The surprising part is that the `InterruptedException` is wrapped in an `ExecutionException`.

In principle, we could unwrap this exception for you. However, this forces all `LoadingCache` users to handle `InterruptedException`, even though the majority of `CacheLoader` implementations never throw it. Maybe that's still worthwhile when you consider that all *non-loading* threads' waits could still be interrupted. But many caches are used only in a single thread. Their users must still catch the impossible `InterruptedException`. And even those users who share their caches across threads will be able to interrupt their `get` calls only *sometimes*, based on which thread happens to make a request first.

Our guiding principle in this decision is for the cache to behave as though all values are loaded in the calling thread. This principle makes it easy to introduce caching into code that previously recomputed its values on each call. And if the old code wasn't interruptible, then it's probably OK for the new code not to be, either.

I said that we support interruption "in a sense." There's another sense in which we don't, making `LoadingCache` a leaky abstraction. If the loading thread is interrupted, we treat this much like any other exception. That's fine in many cases, but it's not the right thing when multiple `get` calls are waiting for the value. Although the operation that happened to be computing the value was interrupted, the other operations that need the value might not have been. Yet all of these callers receive the `InterruptedException` (wrapped in an `ExecutionException`), even though the load didn't so much "fail" as "abort." The right behavior would be for one of the remaining threads to retry the load. We have a bug filed for this. However, a fix could be risky. Instead of fixing the problem, we may put additional effort into a proposed `AsyncLoadingCache`, which would return `Future` objects with correct interruption behavior.