[TopCoder Software]

Simple Cache ver 2.0 - Component Specification

Please note that the version 2.0 builds on version 1.1 and as such this design will combine the base documentation from ver 1.1 specifications. To that effect all the ver 2.0 documentation will be provided in blue so that it is easier to differentiate the changes/additions.

1. Design

The Simple Cache component holds commonly used objects that can be retrieved by key. It is similar to a Map class, in that is implements get() and put() methods, among others.

However, the Simple Cache component is designed for use in high-performance, multi-threaded environments, in order to provide significant additional functionality that enhances its performance as a cache.

To this end, this component provides for:

- A cache size limit, and different strategies for evicting cached objects
  - This will encompass the ability to measure the size in terms of the number of objects in the cache (i.e. size of cache)
  - It will also involve the ability to measure the size of the cache in terms of the bytes that the entries actually occupy in the cache. (i.e. byte capacity of the cache)
- Memory considerations - the cache provides facilities to ensure it does not consume too much memory
- Expiration of cache entries after a given length a time
- Thread-safe access
- The ability to compress objects as they are placed into the cache and then decompress them upon being retrieved. This will allow for more objects to be stored in the cache.

The SimpleCache class is the heart of the Simple Cache component. It exposes an interface that is build for ease of use as well as flexibility. Single object operations as well as set operations have been implemented. Here is a listing of the available functionality:

- Subset of the Map interface: `get()`, `put()`, `remove()`, for single objects.
- Specialized removal functionality for more pinpointed and powerful removal capabilities:
  - `removeLarger(Object value)` which will remove all entries in the cache that are larger (in bytes) than the input object.
  - `removeOlder(Object key)` which will remove all entries in the cache that have been in the cache longer than object value specified by the input key.
  - `removeLike(Object object)` which will remove all entries in the cache that are of the same class type as the input object.
  - `removeSet(Set keys)` which will remove all the values specified by the keys in the input set. This is simply a batch remove.
  - `removeByPattern(String regex)` which will remove all the object values in the cache whose keys are matched by the input regular expression.
- The ability to list all the keys as well as all the values in the cache in the form of `keyset()` and `values()` methods just like in a Map interface.
- The ability to set the eviction strategy to be used by cache in the form of a `setEvictionStrategy()` method.
- The ability to get the size information about the cache based on number of entries as well as the byte size of the entries:
  - getByteSize() will return the current size of the cache in bytes.

    - getMaxByteSize() which will return the maximum allowed cache capacity.
    - getSize() will return the current number of entries in the cache
    - getMaxSize() will return the maximum allowed number of entries in the cache.
- Specialized add (put) functionality for more powerful value addition capabilities:
    - put(Map cacheEntry) which will add all the entries in the input Map to the cache. This is simply a batch put.
- The ability to set/get the compression processing done in the cache.
    - setCompressionFlag() will allow for turning on and off compression
    - getCompressionFlag() will allow to test of the cache is compressing.

The cache stores and retrieves objects by key. In the interest of simplicity, the cache interface will not support the eviction as well as the size functionality. This functionality will be present in the concrete class since it doesn't relate directly to its intended usage as a cache.

Cache size limit

When it comes to the size of the cache, there are two different ways dealing with it. We can look at the size by using either a number of items in the cache or the number of bytes that all the entries take up in the cache. When dealing with a size that is used to control the number of items in the cache we really have no way of controlling the memory consumption since the entries could range from very small to very large. By introducing the capacity of the cache in bytes and an upper limit of maximum capacity we have a fine-tuned control on the memory consumption of the cache.

We need to distinguish between the two ways:

1. The maximum number of entries that the cache holds at any one time is configurable. If another object (or objects) is added to a full SimpleCache, an object (or objects) must be evicted.
2. The maximum number of bytes (i.e. capacity) of the cache is also configurable. If another object (or objects) is added to a cache that would result in over flown capacity, then an object (or objects) must be evicted to make space for the new object.

The cache has a CacheEvictionStrategy object that determines the next object to be evicted. The following implementations are provided

- LRUCacheEvictionStrategy: evicts the least-recently accessed object (default)
- FIFOCacheEvictionStrategy: simpler strategy that evicts the least recently added object
- BOFCacheEvictionStrategy: evicts the biggest object first. This is a simple, greedy strategy of freeing up memory.

Other implementations can be created and plugged in to the SimpleCache object (e.g. random eviction).

Memory considerations

A cache speeds up access to frequently used data, but necessarily uses additional memory to cache this data. By design, the cache is not allowed to grow so large so as to exhaust available memory.

For this reason, SimpleCache maintains only soft references to the objects that it caches, using java.lang.ref.SoftReference objects. This allows the JVM to reclaim memory used by cached objects if the JVM faces an out-of-memory situation. It does add some complexity to the SimpleCache implementation, since it is not notified of these removals, but must detect later.

Expiration of cached objects
The cache may be configured to expire and remove cached objects after a certain amount of time. A "timeout at" timestamp is associated to each cached object - see the SimpleCache.Entry class. A daemon thread periodically scans all cache entries and removes the timed out ones.

Thread-safe access
Internally, the SimpleCache implementation is synchronized as needed so that it may be used as-is in a multi-threaded environment. The new setter capabilities added to the simple cache in ver 2.0 are also thread-safe. Synchronization is to be done on the internal map object used to store the cache entries.
`CompressionHandler` implementations are thread-safe since they act as utility methods with no changing state. Once a `CompressionHandler` is initialized there is no change to its state.

Compression Handlers
Different types of objects with different data characteristics could compress with different compression-ratio efficiency. While we could use one generic compression handler to compress all entries in the cache, in practice if we are dealing with special data types (say text oriented entries vs. binary data) we might want to use different compression handlers that specialize in compressing that particular type of data for maximum efficiency. This design offers a very flexible compromise of having the ability to configure any number of handlers in a chain to deal with very specific data types (either by class type match or by regex class name match) At the same time ease of use is assured with all this behavior being optional.

Dealing with Cache size when soft references are reclaimed
One issue that is of concern is the fact that when a soft reference is silently reclaimed by Garbage Collector then the number of entries in the cache as well as the memory usage of the cache will not be properly reflected.
The issue is that the Cache doesn't get notified when such removal happens and thus it still 'thinks' that the entry (and its associated size) is part of the cache. Currently, I do not see a way to deal with this problem in a foolproof way except for scanning the cache constantly to see what has been removed (de-referenced) but such an approach flies against the requirement of making the cache fast.
Currently the design follows this analysis:
The memory size and the number of entries should be treated as an upper bound thus it will mean no less than. It is possible that since an object has been removed than the actual count is off by 1 and the actual size is off by the size(object). This means that the cache is at most that size. This is an acceptable behavior when dealing with eviction strategies, which will simply evict based on the recorded size rather than the actual size. Since the cache itself only uses the size to determine when to invoke the eviction strategy no incorrect behavior will result.

1.1 Reference any design patterns used

Strategy Design pattern:

- The CacheEvictionStrategy interface and its implementations are an example of the Strategy design pattern.
- The MemoryUtilizationHandler interface defines a contract for plugging-in different implementations of discovering the size of an object in memory.
- The CompressionHandler interface allows for the plugging-in of diverse implementations of compression process.

Chain Of Responsibility pattern: Simple cache utilizes (somewhat) the Chain Of responsibility pattern since it queries a list of compression handlers to see which one can handle the compression of the input object. When one such handler is found it is used to compress the entry.

Iterator pattern: Used to give user application access to all the values in the cache.

1.2 Reference any standards used in the design

None.

1.3 Explain any required algorithms for the implementation (provide pseudo code)

1.3.1 Working with entries

Getting from a key to its corresponding value takes a couple steps, like what follows, and each step deserves a couple notes:

```
Reference ref = (Reference)map.get(key);
// ref could be null - there may be no value for that key
Entry entry = (Entry)ref.get();
// entry could be null if the JVM reclaimed it (soft reference)
long timesOutAt = entry.getTimesOutAt();
// now might be after timesOutAt, in which case this is removed
Object value = entry.getValue()
// finally!
```

The `CacheCleanupThread` pays attention to the timeout, so that the `get()` method doesn't have to. Both must watch for reclaimed soft references though.

If the cache is configured to compress entries then we need to ensure that if an entry is compressed then we need to have an additional step that decompresses the entry before it can be given back to the calling application:

```
// get the handler used to compress/decompress
CompressionHandler ch = ((CompressionHandler)entry.getHandler());
// decompress the value in the cache
Object value = ch.decompress(entry.getValue());
```

When compression is on we need to ensure that we set in the Entry object a reference as to which handler should be used for decompression. Otherwise we might not know which handler to use to decompress the entry.

```
//look for the compression handler that can handle this object
Object result = null;
CompressionHandler compressor = null;
for(int i = 0; i < compressionHandlerList.length; i++){
   compressor = (CompressionHandler) (compressionHandlerList.get(i));
   try{
       result = compressor.compress(value);
   }catch(TypeNotMatchedException tnme){
       // this compressor doesn't know how to compress this;
                   compressor = null;
   }
   if(compressor != null){
       break; // found it
   }
}
// Create an entry for it
Entry entry = new Entry(result, , ,compressor, );
```

Entry MetaData
There also is an introduced notion of Entry metadata, which keeps track of some useful facts about the value like its size, time when it was put in the cache, as well as time when it was last accessed. This will be especially useful when we need to initialize a plug-in eviction strategy which needs to set up its data based on entries that already exist in the cache. This way metadata helps to establish facts about entries that would otherwise be costly to calculate (like size) or impossible to calculate (like put time and last get time)

1.3.2 Cleanup Thread

SimpleCache has an inner class that is a Thread, which periodically looks for and removes timed out entries. A couple points to note on the implementation:

- The constructor calls "setDaemon(true)" so that this thread does not prevent the JVM from terminating.
- The finalize() method of SimpleCache calls this thread's halt() method.
- The thread deals with the case where entry in the SoftReference has been reclaimed.
- The thread handles all timeouts; thus, the SimpleCache's get() method does not have to.

1.3.3 Interaction with CacheEvictionStrategy

SimpleCache contains a CacheEvictionStrategy object, which encapsulates its strategy for choosing the next entry to evict when the cache must make room. SimpleCache notifies the CacheEvictionStrategy when events occur, like gets and puts, so that it can in return give suggestions for the next item to evict.

The implementation of SimpleCache keeps the CacheEvictionStrategy as up-to-date as possible concerning the state of the cache. The following are guidelines the cache eviction strategy follows:

- Calls notifyOfCacheGet(key) when a key is accessed, but its value is not changed or removed
- Calls notifyOfCachePut(key) when a key's value is changed.
- Calls notifyOfCachePut(key, CacheEntryInfo) when a key's value is changed and we need to also supply metadata about the entry (but not the value) such as size etc… This should actually be the preferred way of doing put notification as of ver 2.0.
- Calls notifyOfCacheRemove(key) when a key is removed, or its value is removed (reclaimed, timed out)
- Calls notifyOfCacheClear() when the cache is cleared

Note also that CacheEvictionStrategy.getNextKeyToEvict() is allowed to return null (to indicate "no preference"), and may return a key that no longer exists in the cache (it can't avoid this possibility anyway, because of reclamation of soft references). SimpleCache expects and handles these possibilities.

In addition we have the ability (since ver 2.0) to switch strategies midstream so to speak. This means that we can plug-in at run time a new eviction strategy. The problem to solve is the fact that since when we plug-in a new strategy and the cache possibly (most likely) has entries already the eviction strategy must be synched up with the cache entries. For this purpose a new method has been added to the interface:

```
public void init(HashMap entriesMap)
```

This allows for the strategy to be given all the current keys in the cache with all the

necessary metadata about each key's entry in the cache.

1.3.4 FIFOCacheEvictionStrategy

This implementation utilizes a queue. The notifyOfCacheRemove(key) method using a LinkedList is O(n), since it must search the list for the key.

1.3.5 LRUCacheEvictionStrategy

Every time a key is accessed, it is moved to the "head of the line"; the last in line is always the one to evict.

1.3.6 BOFCacheEvictionStrategy

This implementation should utilize a fast strategy for creating a sorted list where we sort on the size of the entry, with the largest entry at the tail of the list. This means that insertion algorithm should be utilizing binary search or a variant for insertions. Evictions will be O(1).

1.3.7 Dealing with evictions in general

When dealing with eviction that must remove more than one entry the cache (and not the eviction strategy implementation) must simply loop until enough entries have been evicted so then enough space has been created to insert the new entries. This can be broken down into two scenarios:

- When we are dealing with capacity then we need to ensure that the entry we are trying to put in will not go over the maximum capacity. This means that we keep on evicting the entries until we have enough space in the cache to do the insertion.
- When we are dealing with size defined by the number of entries then we simply evict as many entries as we need to in order to ensure (for batch insertions) that we have enough space.

Since we have two ways of defining size for a cache (number of entries, number of bytes all entries occupy) we can also configure the maximum size in 4 different ways:

1. Number of entries only: The restriction will only be made on number of entries.
2. Byte-size capacity only: The restriction will be made on the actual number of bytes consumed by the cache. This is the sum of all the entries and only the actual value in the cache matters.
3. Both number of entries and byte-size capacity: Whichever of the limits is met first is the one that will be acted on but both must be fulfilled in order for an entry to be put into the cache.
4. No restrictions.

In general please refer to method documentation in Poseidon's Class Diagram for implementation information.

1.4 Component Class Overview

Note **blue** means here existing class modification; **red** means brand new class, **black** means no changes.

Cache

This is the basic contract for a cache. This contract deals with the basics of cache manipulation.

SimpleCache
This is an implementation of the Cache interface which has some additional functionality added to it.
This implementation can be configured through a configuration file, which allows the user to set the following aspects of the cache:

(a) maxCacheByteCapacity
(b) maxCacheSize
(c) Compresssion Handler List and the compression flag
(d) memoryUtilizationHandler
(e) Eviction strategy

We also have the ability to set the CacheEvictionStrategy at run time and we have the following additional removal functionality:

(a) largest, oldest, class type based, and regular expression based removals

This is a thread-Safe implementation

CacheIterator
This is a nested class (i.e. inner class helper) within the SimpleCache class which represents an implementation of an external iterator over the collection of cache entry values. Since this is an external iterator in the sense that we give this instance out to the requesting user who can then manipulate the entries of the cache as specified by the Iterator interface contract, each remove() and hasNext() will actually trigger EvictionStrategy notification in the cache.
This Iterator is not a *fast-fail* Iterator and as such if entries are actually removed from the underlying collection of values the iterator will still function and will not fail. This implementation actually 'shadows' the key Iterator and as such any changes to the cache entries will affect thsi Iterator.

CacheEntryInfo
This is a MetaData bean for a cache entry. This is a simple, immutable, data transfer object, which holds information about an entry in the cache.

CacheEvictionStrategy
CacheEvictionStrategy decides which entry to evict when cache is full. Such a strategy is synched with the underlying cache through notifications of operations on the actual cache itself. Thus if a cache has a new element inserted then the eviction strategy would be informed of such an operation so that it can keep track of the state of the cache.
To keep things simple only the keys are reported to the eviction strategy. Additionally some metadata (CacheEntryInfo) about the entry can also be passed to the strategy. This would contain info like the size of the underlying entry (which the key represents) in bytes, which might be useful to the strategy, as well as time of creation (i.e. when was the entry put into the cache) and when the entry was last accessed.
To allow for run-time ability to switch strategies a strategy may also be initialized with keys from eth cache with additional metadata information about each entry represented by the key.

FIFOCacheEvictionStrategy
An implementation of CacheEvictionStrategy interface with the FIFO (First In First Out) Strategy.

LRUCacheEvictionStrategy
An implementation of CacheEvictionStrategy interface with the LRU (Least Recently Used) Strategy

BOFCacheEvictionStrategy

This represents the Biggest Object First eviction strategy. When asked for the next object to evict, it will simply return the key to the largest (in bytes) object currently in the cache.

CacheCleanupThread
Inner thread (inner class to SimpleCache), which cleans up entries in the cache that have expired. Started as a daemon thread.

MemoryUtilizationHandler
This is a contract for a utility, which deals with memory utilization by a given object. This means that this interface deals with the definition for a memory utilization discovery (i.e. how much space, in bytes, does an object take up in memory.

SimpleMemoryUtilizationHandler
This is a default implementation of the MemoryUtilizationHandler interface. This implementation delegates the memory utilization processing to the TopCoder *Memory Usage* component.

CompressionHandler
This is a contract for all compression handlers used in deflating and inflating entries.

BaseCompressionHandler
This is a CompressionUtility based compression handler. This means that it utilizes internally the CompressionUtility 2.0 TopCoder component to do the actual compression/decompression.

ObjectByteConverter
This is a ganeral contract for a utility, which can convert any java object into a byte array (i.e. a stream) and then given a byte array can obtain the object representation.

SerializableObjectByteConverter
An ObjectByte converter which uses serialization to obtain the byte array representation of an object and which then uses the process of deserialization to convert the byte array back to an Object.

Entry
Represents the entry in a cache. This entry can be either original uncompressed value object or an actual compressed value in which case there must be an associated compression handler in this entry.
An entry has a size, which is the number of bytes that the value takes up in memory in its present state (i.e. either original or compressed). This size is pre-computed and assigned to an entry for efficiency considerations since computing the size of any object is possibly an intense process.
This is an inner class to SimpleCache and thus an assumption is made that all input values to the entry are valid and correct.

1.5 Component Exception Definitions

Note **blue** means here existing class modification; **red** means brand new class, **black** means no changes.

MemoryUtilizationException
This is a run-time exception is thrown whenever memory utilization handler fails. This signals to the caller that memory usage cannot be computed.

ObjectByteConversionException
This is a run-time conversion exception, which signals the failure of the byte array to object (and vice versa) conversion process.
CompressionException
This is a compression exception, which signals the failure of the compression (including decompression) process.

TypeNotMatchedException
This is a run-time an exception, which signifies the inability to find a matching class for a particular type. This is used when trying to match compression handlers to the types of object they 'know' how to compress/decompress.

CacheInstantiationException
A general purpose run-time Exception, which signifies the failure to create a SimpleCache. This can be due the inability to instantiate necessary components for the operation of the cache or simply due to configuration issues.

NullPointerException
This is thrown whenever we have a null argument passed to the API. Please note that in ver 2.0 all IllegalArgumentException thrown if any argument is null should be replaced with NullPointerException:

- Null "evictionStrategy" argument to SimpleCache constructor
- Null "key" argument to any method in SimpleCache or FIFOCacheEvictionStrategy

NoSuchElementException
This is an exception that is thrown when the user tries to access an element in an Iterator after the end of the underlying collection has already been met. This is utilized in the inner CacheIterator class.

IllegalStateException
This exception is used to signal an impossible state of an Object. In this design this is used by the CacheIterator to signal that we are trying to remove an non-existing value.

IllegalArgumentException
An IllegalArgumentException is thrown in the following situations:

- Nonpositive "maxCacheSize" ,"timeoutMS", maxCacheCapacity argument to the SimpleCache constructor

Note that null is a legal "value" argument to put(); calling it with a null "value" is like calling remove() with the same key.

The CacheEvictionStrategy.getNextKeyToEvict() may return null. SimpleCache takes this to mean "no preference".

Similarly, getNextKeyToEvict() attempts to return keys that still exist in the cache, but it may return keys that do not, or keys to entries that have timed out or been reclaimed by the JVM. This is not an exceptional situation; SimpleCache calls getNextKeyToEvict() again.

1.6 Thread Safety

This component is thread-safe. First of all, the internal map utilized for cache entries has its access synchronized. All eviction strategies are also synchronized internally on their own data structures.

It is also important that a non-modifiable copy be returned on occasions where the cache returns ant internal information such as keys or values. Of course the non-modification is achieved on a reference to list level rather than on the level of individual objects (this would be too expensive)
The Iterator instance which is returned as part of the SimpleCache.values() method is not supposed to be used by multiple threads. In such a sense the Iterator is not guaranteed to be in a consistent state when used by more than one thread at the same time. Also, usage of multiple Iterators by many threads could create unexpected results. Nevertheless the CacheIterator has been designed to be well behaved in a sense that its behavior is well defined.
All setter/getter nethods should be synchronized as follows:

- SetEvictionStrategy should synchronize on the actual evictionStrategy member at the moment of the switch
- To ensure the thread-safety of the setting the compression flag synchronize on the map itself.

Please refer to the Poseidon class/method documentation for more info as each class is very clearly marked with information about synchronized access.

2. Environment Requirements

2.1 Environment

- Development language: Java1.4
- Compile target: Java1.4

2.2 TopCoder Software Components:

The following components have been utilized:

- **Configuration Manager 2.1.3**
  - Used to read configuration for the cache
- **BaseException**
  - Used for exception chaining. Even though the target is JDK 1.4+ the designer has decided to use BaseException in case future enhancements to the BaseException are implemented that might be useful to the component.
- **Compression Utility 2.0**
  - Used for compression/decompression tasks needed by the cache.
- **Memory Usage 1.0**
  - Used for discovering the size (deep size) of an arbitrary object.

2.3 Third Party Components:

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.cache20

3.2 Configuration Parameters

We need to be able to configure the following aspects of the cache:

1. SimpleCache main info such as
   a. **CacheEvictionStrategy**
      i. Class name of the strategy to use.
      ii. Example: “com.topcoder.util.cache20.BOFCacheEvictionStartegy”

iii. It is an optional item and can be omitted

b. MaxCacheSize
- i. Max number of items to be held in the cache
- ii. Example: “1000”
- iii. It is an optional item and can be omitted

c. MaxCacheByteCapacity
- i. Maximum memory consumption allowed for the cache in bytes.
- ii. Example: 10000000 (i.e. about 10 Mb)
- iii. It is an optional item and can be omitted

d. TimeoutMS
- i. Cache refresh interval in milliseconds
- ii. Example: 3600000 (i.e. 1 hour)
- iii. It is an optional item and can be omitted

e. MemoryUtilizationHandler
- i. class name of the handler to use
- ii. Example: “com.topcoder.util.cache20.SimpleMemoryUtilizationHandler”
- iii. It is an optional item and can be omitted

f. CompressionFlag
- i. true/false if compression is needed or not
- ii. Example: true
- iii. It is an optional item and can be omitted (defaults to false)

g. CompressionHandlers
- i. Please refer to compression handlers below.

2. Compression Handler list configuration: each item can be set as follows:

a. CompressionHandlerClass
- i. class name of the handler
- ii. Example: “com.compression.SomeHandler”

Each compression handler entry can have 0 or more class types associated with it, which signifies the objects types that the handler can handle:

AcceptedObjectTypes

Example:

“java.util.Vector”

“java.util.HashMap”

This means that the above SomeHandler would be configured to accept the two object types (and all their descendants)

Each compression handler entry can have 0 or 1 regular expression that can also be used to check ability to compress. This signifies also the objects that the handler can compress:

AcceptedObjectTypesRegex

Example:

“java.util.*”

This would match all classes that have the java.util prefix as being compressible by the handler

For each compression handler we can also specify an optional codec name:

CompressionCodecClass

Example:

“com.topcoder.util.compression.LZ77Codec”

Finally for each handler we can also specify an optional ObjectByteConverter to be used when mapping an Object into a stream of bytes and back into an object.

ObjectByteConverterClass

Example:
"com.topcoder.util.cache20.SerializableObjectByteConverter"

Note that the SimpleCache class is also configurable at runtime through the ctor.
For an example of an actual configuration file please refer to
.\docs\SimpleCacheConfig.xml configuration file.

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

1. Configure the cache file if external configuration is required. In such a case instantiate the cache with a proper namespace
2. We can use the `SimpleCache` right out of the box. Instantiate a `SimpleCache` and then use it.

For more information please refer to the demo section (4.3).

4.3 Demo

It is quite difficult to make up a caching scenario that is realistic but here we will try to simply demonstrate the usage of the cache using results of expensive operations. Lets us assume that we are caching the results of database queries. We will offer two demos: One, which will expose the details of the methods, and the second a nice relaxed usage scenario
For the demo lets assume the following simple `Record` class:

```
// Simulated data base record. This will also simulate a variable but
// predictable memory load since each record will be driven by its id for
// memory consumption.
class Record implements Serializable{
       int id;
       String record;
       Int[] memoryLoad = null;
       public Record(int id, String record){
              this.id = id;
              this.record = record;      // simulate record content
              memoryLoad = new int[id];  // simulate memory load
       }

       // a simulated db reader
       public static Record readFromDB(int id){
              return new Record(id, "Record "+id);
       }
       public int getID(){
              return id;
       }
       public String getRecordInfo(){
              return record;
       }
}
```

4.3.1 *Creating a default based cache*

Here we will simply create a simple out–of-the-box cache with unlimited number of records but with limited memory consumption with an upper bound of 2 Mb, which will be refreshed every 3 hours and which will use the BOF eviction strategy. We will also use default compression on the entries.

```
//
// create a new SimpleCache instance
recordCache = new SimpleCache(
             SimpleCache.NO_MAX_SIZE,          // no size limit
              3*60*60*1000,                    // 3 hour refresh
             new BOFCacheEvictionStrategy(),   // eviction strategy
             2*1024*1024,                      // capacity of 2 Mb
             null,                             // default memory handler
             null,                             // default compression
             true);                            // compression is on
```

4.3.2 *Working with the cache entries*

Here we will simulate some work done on the database in the sense that we will read a number of records. Here we will assume that we have 1000 records, which will be read from the database

```
//
// Populate the cache with 1000 expensive to get records. We will use the
// record id as the key for the object.
for(int i=1; i<=1000; i++){
      Record rec = new Record.readFromDB(i);
      recordCache.put(“”+i, rec);
}

// Lets check that all records have been successfully entered. This of
// course assumes that the combined size of all compressed entries is
// less than 2 Mb
//    We should get 1000 entries
System.out.println(“The number of entries in the cache is: “
             + recordCache.getSize());

// Lets check how much memory this takes up
System.out.println(“The memory consumption is: “
             + recordCache.getByteSize()
             + “ out of max “
             + recordCache.getMaxCacheCapacity() + “ bytes.”);

// Lets check if compression is on. It should be on
System.out.println(“Compression flag is: “
                  + recordCache.getCompressionFlag() ? “on” : “Off”);


//
// lets view the keyset of this cache. We should get keys from 1 to 1000
Set keyset = recordCache.keyset();
Iterator it = keyset.iterator();
while(it.hasNext()){
      System.out.println(“Key is: “ + (String) (it.next()));
}


//
// lets get all the values in the cache. For this demo we should get
// values from “Record 1” to “Record 1000”
```

```
Iterator recordIterator = recordCache.values();
while(recordIterator.hasNext()){
	System.out.println("record is: "
		+ ((Record) (recordIterator.next())).getRecordInfo());
}


// Lets get a specific record (id=1000) from the cache
// this should return "Record 1000"
Record rec = (Record)(recordCache.get("1000"));
System.out.println("record is: " + rec.getRecordInfo());


//
// Removal functionality

// Remove the first record
recordCache.remove("1");


// Remove records "2" through "9"
Set keysToRemove = new Set();
for(int key=2; key<=9; key++){
	keysToRemove.add(""+key);
}
recordCache.removeSet(keysToRemove);
// Remove records whose keys match the regular expression
// "/^1[0-9]{2}/" which should remove all the values whose keys
// match any string from "100" through "199"
recordCache.removeByPattern("/^1[0-9]{2}/"");


// remove records older than the records with key "99"
// Since the records have been added from "1" to "1000" in a timed
// sequence this means that any entry whose id is < "99" is older.
recordCache.removeOlder("99");

// remove records larger that the records with key "950"
// Since the records have been added with memory capacity based on the id
// number we should have records with ids > "950" removed from the cache.
// Assumption is that if compression is used then even compressed the
// records would be relative.
recordCache.removeLarger(readFromDB(950));

// Here we will simulate adding some rouge objects of type Vector
recordCache.put("rouge001", new Vector());
recordCache.put("rouge002", new Vector());
// Now we remove these object based on their class type
recordCache.removeLike("java.util.Vector");

// We will clear the cache completely
recordCache.clear();

// We will turn off compression
recordCache.setCompressionFlag(false);

// We will repopulate the cache with records
for(int key = 1; key <=1000; key ++){
	Record rec = new Record.readFromDB(key);
	recordCache.put(""+key , rec);
}

// We will switch the eviction strategy mid-stream
```

```
recordCache.setEvictionStrategy(new FIFOCacheEvictionStrategy());
```

4.3.3 A very basic SimpleCache demo… configured

Assume that a configuration file exists which actually does the complete initialization aspect.

```
public class UserManager {

	private SimpleCache userInfoCache;
	...

	public UserManager() {
		// Use a configured cache with default namespace
		// We could have done instead:
		// userInfoCache = new SimpleCache("con.acme.Cache");
		userInfoCache = new SimpleCache(null);
		...
	}

	...

	public UserInfo getUserInfo(String username) {
		UserInfo cachedInfo = (UserInfo)userInfoCache.get(username);
		if(cachedInfo == null) {
			...
			cachedInfo = // do actual, expensive lookup
			...
			userInfoCache.put(username, cachedInfo);
		}
		return cachedInfo;
	}

	public void createUser(String username, UserInfo info) {
		// create user in database
		...
		// you could let getUserInfo retrieve this and cache,
		// but you might also add it to the cache ahead of
		// time:
		userInfoCache.put(username, info);
	}

	public void deleteUser(String username) {
		// delete user from database
		...
		// tell the cache that this user is gone:
		userInfoCache.remove(username);
	}

	public void shutdown() {
		// free up resources
		userInfoCache.clear();
	}
	...
}
```

5. Future Enhancements

- Currently the size of the cache (number of entries and the physical memory consumption) is not up to date when soft references are silently reclaimed. In the future we might get a better approximation by noting in the cache that an entry has bee reclaimed (for example when client calls get() we can see that

it has been reclaimed and then we update the size and num of entries accordingly)