**Please see the API and JavaDoc for the most current version of the spec.**

# JSR107: JCACHE
# Java™ Caching API

*JSR107 Expert Group*

Specifications Leads:
Greg Luck, Terracotta Inc.
Yannis Cosmadopoulos, Oracle Corporation

*Early Draft 1*

Comments to: jsr107@googlegroups.com

**NOTICE**

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Sun's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and(iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.
2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:
   (i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; (ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensees control; and
   (iii) includes the following notice:"This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your early draft implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.
Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java" , "javax" , "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof

**TRADEMARKS**

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

**DISCLAIMER OF WARRANTIES**

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO

# Contents

# Chapter 1 - Introduction

This document is the specification of the Java API for the temporary caching. The technical objective of this work is to provide a caching facility for the Java application developer.

This specification describes an API for caching, and an SPI so providers can implement their own caches.

Leading experts throughout the entire Java community have come together to build this Java caching standard.

## *1.1 Status*

This is early draft 1, still in progress.

The latest API can be found online at:

https://github.com/jsr107/jsr107spec

The reference implementation can be obtained from:

https://github.com/jsr107/RI

Finally the TCK can be obtained from:

https://github.com/jsr107/jsr107tck

The expert group seeks feedback from the community on any aspect of this specification, please send comments to:

jsr-107-comments@jcp.org

or, for a publicly readable forum to:

jsr107@googlegroups.com

## *1.2 Purpose*

Caching is a tried and true method for dramatically speeding up applications. Applications often use temporary data which is expensive to create, but has a lifetime over which it can be reused. For example, a servlet might create a web page from data obtained from multiple databases, network connections, and expensive computations; the data sets might be reusable over the same or different periods of time.

This specification standardizes caching of Java objects in a way that allows an efficient implementation, and removes from the programmer the burden of implementing cache expiration, mutual exclusion, spooling, and cache consistency.

## *1.3 Goals*

The goals of the API are:

**Object Cache** The API will cache Java objects.

**Support for By-Value caching and optionally, By-Reference.** In the latter references are cached on heap and in the former both keys and values are transformed into values.

**Support for Flexible Implementations** The specification will support both in-process and distributed implementations.

**Java SE** The specification will work with Java SE.

**Java EE** The specification will work with Java EE. This specification is targeted at inclusion in Java EE 7.

**Annotations** The specification will define optional runtime cache annotations.

**Transactions** Optional support for transactions, both local and XA will be defined.

## 1.4 Conventions

The regular Times font is used for information that is normative for this specification.

*The italic Times font is used for paragraphs that contain non-normative information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.*

The Courier New font is used for code examples.

In addition, the keywords 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this document are to be interpreted as described in RFC 2119.

Java code and sample data fragments are formatted as shown in figure 1.1:

```
package com.example.hello;
public class Hello {
	public static void main(String args[] {
		System.out.println("Hello Worlds");
	}
}
```

*Figure 1: Example Java Code*

## 1.5 Expert Group Members

This specification is being developed under the Java Community Process v2.7.

The following are active expert group members:

- Greg Luck
- Cameron Purdy, Oracle
- Yannis Cosmadopoulos, Oracle
- Manik Surtani, Red Hat, Inc.
- Nikita Ivanov, Grid Gain
- Chris Berry
- Andy Piper

- Jon Stevens
- Rick Hightower
- Ben Cotton
- David Mossakowski, Citigroup
- Bongjae Chang
- Steve Millidge
- Gabe Montero, IBM
- Eric Dalquist
- Pete Muir, Red Hat, Inc.
- William Newport, Goldman Sachs
- Ryan Gardner, Dealers.com
- Steve Harris, Terracotta

The following are members of JSR342 (Java EE 7) and are official observers of JSR107 which means inter alia they get read only access to the mailing list :

- Linda Demichiel, Oracle
- Roberto Chinicci, Oracle

### *1.6 Acknowledgements*

During the course of the JSR we have received many excellent suggestions on the JSR mailing lists. Thanks to those people.

# Chapter 2 - Caches

A cache is a place to store temporary data. It presents a Map-like API. Like a map, data is stored as values by key.

Like a map data is accessible via a key, which is a unique identifier. There can only be one entry in a cache for a given key.

The primary programming artifact is the Cache interface. The caller interacts with the Cache.

*It is often assumed that data from a database is being cached. If you are using JPA then that it is certainly the case. However fundamentally anything that is expensive or time consuming to produce can be stored in a cache. Some common use cases are:*

- *client side caching of Web service calls*
- *caching of expensive computations such as rendered images*
- *caching of database rows*
- *caching of NoSQL data in-process*
- *servlet response caching*
- *caching of domain object graphs*

## 2.1 Requirements of Keys and Values

Keys and values are not restricted in type unless specified with a generic constructor when the cache is created.

Both keys and values must be non null. An attempt to modify the cache using a null key or value will result in a NullPointerException, as will attempts to fetch from the cache using a null key.

Caches must support storeByValue. The implication of this is that there must be a way to convert keys and values to an non-object representation. *This typically means an off-heap representation capable of being sent over a network or stored in a persistent store.*

*One way to achieve this is for keys and values implement Serializable however implementers may choose other methods. Any scheme to support this other than Serializable will be implementation specific and will therefore not necessarily be portable between implementations.*

## 2.3 Generics

The Cache interface uses generics.

## 2.4 Looking up a Cache

A CacheManager manages Caches which are identified and accessed by name. A reference to a cache is obtained with:

```
Cache cache = cacheManager.getCache("Greg's Cache");
```

## 2.5 Package

The top level package name is “javax.cache”.

### *2.6 Lifecycle*

A Cache goes through the lifecycle as defined by `javax.cache.Status`.

A Cache can only progress in a forward direction through these statuses. Once shutdown it cannot be restarted.

### *2.7 Dependency on CacheManager*

A cache is always obtained from a CacheManager.

Caches are bound into the lifecycle of a CacheManager. When a CacheManager shuts down all contained caches must be shut down.

# Chapter 3 Cache Configuration

### *3.2 Resource Limits*

*While a fully configured cache will require a limit on the number of entries or the resources consumed by a cache, this specification does not specify cache storage topology or structure. Implementations typically place resource constraints per node and/or storage type. However because these are not defined in the specification, there is no standard way to configure them.*

*It is expected therefore that implementations will provide a way to place limits on resources.*

# Chapter 3 - Cache Operations

This chapter deals with operations on a cache, and additional APIs.

The Cache Interface is patterned on ConcurrentMap but does not extend it. This is because some methods of ConcurrentMap (and Map) have signatures that make them inefficient in a distributed environment. As a an example, Map.put returns a value which in the typical case is ignored.

## *Cache Operations*

This is the API for Cache.

## *Cache Entry Expiration*

Entries in the cache may expire based on the last modification and/or access time as controlled by javax.cache.CacheConfiguration.ExpiryType. The following table specifies what operations cause the corresponding clock to start ticking.

An invocation resets the expiry clock for the purposes of determining expiry
 * where the expiry policy is {@link CacheConfiguration.ExpiryType#ACCESSED}.
 * It is has no effect on expiry based on modification.

| **Method** | **MODIFIED** | **ACCESSED** |
|---|---|---|
| boolean containsKey(K key) | NO | YES |
| V get(K key) | NO | YES |
| Map<K,V> getAll(Collection<? extends K> keys) | NO | YES |
| V getAndPut(K key, V value) | YES | YES |
| V getAndRemove(K key) | NO | NO |
| V getAndReplace(K key, V value) | YES | YES |
| CacheManager getCacheManager() | NO | NO |
| CacheConfiguration getConfiguration() | NO | NO |
| String getName() | NO | NO |
| CacheStatistics getStatistics() | NO | NO |
| Iterator<Cache.Entry<K, V>> iterator() | NO | NO<br>(but yes on it.next) |
| Future<V> load(K key) | YES | YES |

| | | |
|---|---|---|
| Future<Map<K,V>> loadAll(Collection<? extends K> keys) | YES | YES |
| void put(K key, V value) | YES | YES |
| void putAll(Map<? extends K,? extends V> map) | YES | YES |
| boolean putIfAbsent(K key, V value) | YES (only if was absent) | YES |
| boolean registerCacheEntryListener(CacheEntryListener<? super K,? super V> cacheEntryListener, NotificationScope scope, boolean synchronous) | NO | NO |
| boolean remove(K key) | NO | NO |
| boolean remove(K key, V oldValue) | NO | YES<br>(only if not matched; not removed) |
| void removeAll() | NO | NO |
| void removeAll(Collection<? extends K> keys) | NO | NO |
| boolean replace(K key, V value) | YES | YES |
| boolean replace(K key, V oldValue, V newValue) | YES<br>(only if matched; replaced) | YES |
| boolean unregisterCacheEntryListener(CacheEntryListener<?,?> cacheEntryListener) | NO | NO |
| <T> T unwrap(Class<T> cls) | NO | NO |

### *Read-Through Caching*

A read-through cache behaves exactly the same way as a non-read-through cache except that get() and getAll() will invoke the CacheLoader if the entries are missing.

The effect on each method invocation when a cache is in read-through mode is described in the following table:

| **Method** | **Invoke Read-Through** |
|---|---|
| boolean containsKey(K key) | No |

| | |
|---|---|
| V get(K key) | Yes |
| Map<K,V> getAll(Collection<? extends K> keys) | Yes. Invokes loadAll |
| V getAndPut(K key, V value) | No |
| V getAndRemove(K key) | No |
| V getAndReplace(K key, V value) | No |
| Object invokeEntryProcessor(K key, EntryProcessor<K, V> entryProcessor); | No |
| Iterator<Cache.Entry<K, V>> iterator() | No |
| Future<V> load(K key) | Yes. Even when the cache is not read-through |
| Future<Map<K,V>> loadAll(Collection<? extends K> keys) | Yes. Uses the CacheLoader.loadAll() method. Even when the cache is not read-through |
| void put(K key, V value) | No |
| void putAll(Map<? extends K,? extends V> map) | No |
| boolean putIfAbsent(K key, V value) | No |
| boolean remove(K key) | No |
| boolean remove(K key, V oldValue) | No |
| void removeAll() | No |
| void removeAll(Collection<? extends K> keys) | No |
| boolean replace(K key, V value) | No |
| boolean replace(K key, V oldValue, V newValue) | No |

### *Write-Through Caching*

It is a proxy. So any mutation always invoked the CacheWriter.
We might want a removeAllNoWriteThrough to give us the capability of just zapping the cache when it is in write-through mode.

### *CacheLoader*

The CacheLoader interface can be used to both pre-load a cache and to allow special action on a cache miss. In the later case this could be used for example to implement read-through.
TODO: need to specify how CacheLoader gets associated with a cache.

### *CacheEntryListener*

Listeners which are sub-interfaces of CacheEntryListener may be registered which are fired when certain cache events occur:

CacheEntryCreatedListener - fired when an entry is created.
CacheEntryUpdatedListener - fired when an entry is updated.
CacheEntryRemovedListener - fired when an entry is removed.
CacheEntryReadListener - fired when an entry is read.

Listeners are registered with one of the following notification scopes:

local - events originated in this JVM
remote - events originated in a remote JVM

If a CacheLoader is registered and causes the creation or update, the event is fired.

See https://jsr107.ci.cloudbees.com/job/jsr107api/ws/target/apidocs/javax/cache/event for a full description.

### *Classloaders*

See http://www.objectsource.com/j2eechapters/Ch21-ClassLoaders_and_J2EE.htm for the problem.

# Chapter 4 - Cache Managers

A CacheManager is a container of and a collection of caches.

As a container it managed all aspects of the cache lifecycle.

It provides the following capabilities:

1. a means of looking up caches by name
2. acts as an XA Resource for transaction managers
3. corresponds to a caching unit for configuration purposes
4. starts all configured caches when itself is started
5. shuts down all contained caches to be shutdown when the CacheManager is shut down
6. is the integration point for clustering in distributed caches. On startup it will allocate any necessary resources and on shutdown will release them
7. provides a default cache template so that new caches can be created with meaningful defaults
8. can be either used as a singleton in which case there will be only one CacheManager or can be used with multiple CacheManagers per VM for more complex situations.
9. allows iteration, addition and deletion of caches from it.

## *Singleton versus Instance*

*Typically a given application will only require one CacheManager. For this reason CacheManagers can be used in the simplest case as Singletons.*

## *Singleton Construction*

A singleton CacheManager is constructed, and referenced, using:

```
CacheManager.getInstance()
```

In this form the configuration must be in a well-known location.

### Instance Construction

Multiple instances of CacheManager may be created using:

```
new CacheManager()
```

See Chapter 7 for details of implementation bootstrapping.

## *Default CacheManager*

For ease of use, implementers should provide default configuration so that in the absence of an implementation specific configuration file being provided by a user, a CacheManager can be created and caches added to it.

It is anticipated that Java SE will add this specification and the Reference Implementation so that this capability will be present in Java SE.

## *Caches must be in a CacheManager*

Because a cache may need the facilities of the CacheManager such as an integration with a cache cluster, caches may not be started outside a CacheManager. They must be added first. If they are in the configuration the CacheManager will do this automatically on startup. They may also be added programmatically to a running CacheManager.

## Chapter 5 - Transactions

Transactions are an optional requirement of this specification. Transactions have the meaning provided by the JTA specification[10]. If implemented, transactions must work as specified here.

Two transaction modes are supported:

1. Global Transactions, also known as XA Transactions.
2. Local Transactions, where the transaction boundary is the CacheManager

*The motivation for providing transactions is that it is often very important for caches to stay strongly consistent with databases, message queues and other XA Resources. Without transaction support cache entries will not be guaranteed to be consistent with these.*

A given Cache can support one of Local or XA transactions but not both at the same time.

### *All or nothing*

If a transactions are enabled on a cache, all operations on it must happen within a transaction context otherwise a javax.cache.transaction.TransactionException will be thrown.

### *XA Transactions*

XA Transactions for caches will work as per the JTA specification and the isolation level chosen.

XA transactions require the presence of a Transaction Manager.

An attempt to operate on an XA cache outside of an XA transaction context will cause a CacheException.

An example using programmatic transaction control is:

```
//Get a global transaction assuming in a Java EE app server
UserTransaction utxg = jndiContext.lookup("java:comp/UserTransaction");

// start the transactions
utxg.begin();

// do work
cache1.put("key1", "value");
cache2.remove("key3");
cache3.put("key5", "value4");

// commit the transactions
utxg.commit();
```

#### Enlistment

*Enlistment is the process by which the Transaction Manager is told that an XA Resource is going to participate* *in a transaction.*

*The javax.transaction.xa package defines the contract between the transaction manager and the resource manager, which allows the transaction manager to enlist and delist resource objects (supplied by the resource manager driver) in JTA transactions.*

*Enlistments is done using TransactionManager.getTransaction().enlistResource(XAResource xaRes). The way in which a reference is obtained to a TransactionManager is not defined by the JTA specification. Java EE application servers typically use a well-known JNDI path to obtain that reference which is vendor specific.*

*On the first XA Resource operation start() is called by the TransactionManager.*

*XA is connection oriented. Caches are connection agnostic which creates an impedance mismatch.*

*Because we do not close connections, XAResource.end() is never called.*

*We expect the Transaction Managers to call end() for us before the two-phase commit protocol is started (even though this is not specified in the JTA specification). This is the behaviour of most existing TransactionManagers.*

*The JTA spec requires "Interleaving multiple transaction contexts using the same resource may be done by the transaction manager as long as XAResource.start and XAResource.end are invoked properly for each transaction context switch. Each time the resource is used with a different transaction, the method XAResource.end must be invoked for the previous transaction that was associated with the resource, and XAResource.start must be invoked for the current transaction context."*

*Because we do not call end if the XAResource was at the CacheManager level this would imply only a single transaction could be done at a time across the CacheManager. This is a possible implementation.*

*It is suggested that a new XAResource is created for each transaction so that a single cache manager will have as many XAResources open as there are transactions. In this way concurrent transactions are supported. The interleaving issue is also avoided.*

*Another possible implementation is to create an XAResource per cache operation. This is highly concurrent but requires more calls to the expensive enlistResource() method.*

## Recovery

Caches must implement recovery protocols as defined by JTA. In particular XAResource.recover() must be supported.

In the case of a local in-process cache where there is no durable store, and the process has restarted thus coming up with an empty cache, it is acceptable for recover() to return an empty array of javax.transaction.xa.Xid[]. The Transaction Manager may report a Heuristic Exception in this case. *This will not prevent pending transactions being correctly recovered for other XAResources. Further because the local cache is empty any attempt to read an affected value will be a cache miss so the user logic will go elsewhere for the value.*

### *Local Transactions*

A cache supporting transactions will support Local Transactions with the four isolation levels.

Local transactions do not require the presence of a Transaction Manager.

*Local Transactions allow single phase commit across multiple cache operations against one or more caches in the same CacheManager, whether distributed or local. This lets you apply multiple changes to a CacheManager all within a single transaction. If you also want to apply changes to other resources such as a database then you need to open a transaction to them and manually handle commit and rollback to ensure consistency. (Or use XA Transactions).*

*For example, we have two puts to Cache A and one remove to Cache B, and 4 puts to Cache C. These can all be accommodated in a single local transaction.*

The JTA API is used to control local transactions. The javax.transaction.UserTransaction interface provides the application the ability to control transaction boundaries programmatically.

```
//Get a transaction
UserTransaction utx = cacheManager.getUserTransaction();
// start a transaction
utx.begin();
// do work
cache1.put("key1", "value");
cache2.remove("key3");

// commit the work
utx.commit();
```

*The best practice as with all local transactions is to place these steps in a try-catch block and call rollback() if an exception is thrown. See the JTA spec for details*[10]*.*

An attempt to operate on a cache outside of a local transaction context will cause a javax.cache.transaction.TransactionException.

It is possible for a single thread to have begun a XA Transaction and a local transaction. Cache operations will then be accepted for both XA and local transaction caches because both transaction contexts exist. However the transactions are separate.

So another programmatic (bean managed in EJB language)  example showing this where cache1 and cache2 are configured for Local Transactions and cache3 is configured for XA Transactions would be:

```
//Get a local transaction
UserTransaction utxl = cacheManager.getUserTransaction();
```

```
//Get a global transaction assuming in a Java EE app server
UserTransaction utxg = jndiContext.lookup("java:comp/UserTransaction");

// start the transactions
utxl.begin();
utxg.begin();

// do work
cache1.put("key1", "value");
cache2.remove("key3");
cache3.put("key5", "value4");

// commit the transactions
utxl.commit();
utxg.commit();
```

Though this works, it is not particularly useful as one transaction can succeed on commit and the other can fail.

Local Transactions has it's own exceptions that can be thrown, which are all subclasses of CacheException. They are:

- TransactionException - a general exception
- TransactionInterruptedException - if Thread.interrupt() got called while the cache was processing a transaction.
- TransactionTimeoutException - if a cache operation or commit is called after the transaction timeout has elapsed.

### *Recovery*

This is relevant to XA Transactions. For local transactions if you crash before commit() then the changes will depend on your isolation level.

# Chapter 6 - Isolation Levels

The isolation level for a cache must be set at creation time and remains immutable for the lifetime of the cache.

The isolation levels READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ and SERIALIZABLE are required.

## *READ_COMMITTED*

Mutating changes are not visible to other transactions in a  local cache or a distributed cache until COMMIT has been called.

*Until then Implementations are free to either:*

- *return the old copy*
- *block until commit or rollback is called*

*Both approaches satisfy the READ_COMMITTED isolation level.*

## *READ_UNCOMMITTED*

Cache mutations are immediately visible to other transactions in a  local cache or a distributed cache, subject to any propagation delay of the implementation, as if transactions were not being used.

On commit, no value changes will be observed.

On rollback, the values will be reverted to their previous values, which will of course be a visible change.

*On timeout, the JTA specification states that rollback is called. So on timeout the old values will be reverted too. Exactly when the rollback occurs will be implementation dependent.*

## *SERIALIZABLE*

Mutating changes are not visible to other transactions in a  local cache or a distributed cache until COMMIT has been called.

Further no changes to the cache made by other transactions are visible to this transaction until it completes.

The SERIALIZABLE isolation level offers one further protection over REPEATABLE_READ, protection from Phantom reads.

An alternative is to exclusively write lock a collection of keys of interest before starting your transaction. We could use lockAll(Collection keys). This would create a ReadWrite lock. Other transactions would block until this transaction.

*This behaviour could be achieved pessimistically with a ReadWrite lock over the entire cache or also achieved optimistically by triggering a RollbackException if any changes made to the keys used (for reads or writes) in* *this transaction have been made.*

## *REPEATABLE_READ*

Mutating changes are not visible to other transactions in a  local cache or a distributed cache until COMMIT has been called.

Further no changes to the cache made by other transactions for keys once they have or written by this transaction are visible to this transaction until it completes.

*This behaviour could be achieved pessimistically with a ReadWrite lock acquired over the keys as they are used or also achieved optimistically by triggering a RollbackException if any changes made to the keys used (for writes) in this transaction have been made.*

# Chapter 7 - Caching Annotations

This chapter talks about the annotations defined in the Java Caching 1.0 specification. Annotations are an optional part of this specification and can be implemented as a stand-alone support library. The JavaDoc for the annotations classes should be read for detailed descriptions of functionality.

## *7.1 - Annotations Overview*

The annotations and support classes are in the javax.cache.`annotation package`.

### 7.1.1 - Annotation Inheritance and Ordering

This specification defers to section 2.1 of the Common Annotations for Java specification for annotation inheritance. Order of interceptor execution with regards to annotations outside of this specification is not defined and left to the annotation support implementation.

### 7.1.2 - Multiple Annotations

Only one method level caching annotation can be specified on a method and only one parameter level caching annotation can be specified on a parameter. If more than one annotation is specified on a method or on a parameter then a CacheAnnotationConfigurationException must be thrown either at application initialization time or on method invocation.

### 7.1.3 - Transactions

If a cache is transactional, then a transaction context must exist when a caching annotated method is executed. If a transaction does not exist when the method is executed the Cache will throw a CacheException.

## *7.2 - Annotations*

In an application, the method and configuration of processing caching annotations on classes is left to the implementation.

### 7.2.1 - @CacheDefaults

This is a class level annotation used to define default property values for all caching related annotations used in a class. The cacheName, cacheResolverFactory, and cacheKeyGenerator properties may be specified though all are optional.

If @CacheDefaults is specified on a class but no method level caching annotations exist then the @CacheDefaults annotation is ignored.

The following example shows specifying a cache named “cities” which will be used as the default cache name for the class. The @CacheResult annotation on the getCity method will use this cache name at runtime.

**Code Example 7-1** @CacheDefaults Annotation Example

```
@CacheDefaults(cacheName=”cities”)
public class CitySource {
    @CacheResult
    public City getCity(int lat, int lon) {
        //...
    }
}
```

### 7.2.2 - @CacheResult

This is a method level annotation used to mark methods whose returned value is cached using a key generated from the method parameters and returned from cache on later calls with the same parameters.

The @CacheKeyParam annotation can be used to select a subset of the parameters for key generation.

**Options**

1. Toggle caching of null return values via the cacheNull property.
2. Optional caching and re-throwing of exceptions with their own named cache, includes the ability to only cache specific exceptions.
3. Optional skipping of the pre-execution Cache.get call, useful when the annotated method should always be executed and the returned value placed in the cache.

@CacheResult will be ignored if placed on static methods.

### 7.2.3 - @CachePut

This is a method level annotation used to mark methods where one of the method arguments should be stored in the cache. One parameter must be annotated with @CacheValue marking it the parameter to be cached. If no @CacheValue annotation is specified a CacheAnnotationConfigurationException must be thrown either at application initialization time or on method invocation.

The @CacheKeyParam annotation can be used to select a subset of the parameters for key generation, the @CacheValue annotated parameter is never included in key generation.

**Options**

1. Toggle caching of null parameter values via the cacheNull property.
2. Specify if the Cache.put call will happen before or after method execution.
3. If caching happens after invocation then an exception thrown by the annotated method can cancel the Cache.put call

@CachePut will be ignored if placed on static methods.

### 7.2.4 - @CacheRemoveEntry

This is a method level annotation used to mark methods where the invocation results in an entry being removed from the specified Cache.

The @CacheKeyParam annotation can be used to select a subset of the parameters for key generation.

**Options**

1. Specify if the Cache.remove call will happen before or after method execution
2. If removal happens after invocation then an exception thrown by the annotated method can cancel the Cache.remove call

@CacheRemoveEntry will be ignored if placed on static methods.

### 7.2.5 - @CacheRemoveAll

This is a method level annotation used to mark methods where the invocation results in all entries being removed from the specified Cache.

**Options**

1. Specify if the Cache.removeAll call will happen before or after method execution
2. If removal happens after invocation then an exception thrown by the annotated method can cancel the Cache.removeAll call

@CacheRemoveAll will be ignored if placed on static methods.

### 7.2.5 - @CacheKeyParam

This is a parameter level annotation used to mark parameters that are used to generate the CachKey via the CacheKeyGenerator. At execution time the values of the parameters annotated with @CacheKeyParam are placed in the CacheKeyInvocationContext.getKeyParameters() array.

Usable with @CacheResult, @CachePut, and @CacheRemoveEntry

### 7.2.6 - @CacheValue

This is a parameter level annotation used to mark the parameter to be cached for a method annotated with @CachePut. A parameter annotated with @CachePut will never be included in the CacheKeyInvocationContext.getKeyParameters() array.

Usable with @CachePut

## *7.3 - Cache Resolution*

All of the method level annotations allow for specification of a CacheResolverFactory and cache name which are used to determine the Cache to interact with at runtime.

### 7.3.1 - Cache Name

If no cache name is specified either on the method level annotation or the @CacheDefaults annotation the name is generated as the following:

package.name.ClassName.methodName(package.ParameterType,package.ParameterType)

The @CacheResult annotation has an additional exceptionCacheName property, if this property is not specified there is no default exception cache name and no exception cache is used.

### 7.3.2 - CacheResolverFactory

The specified CacheResolverFactory must be called exactly once per annotated method to determine the CacheResolver to use for each execution of the annotated method. When an annotated method is executed the previously retrieved CacheResolver is used to determine the Cache to use based on the CacheInvocationContext.

If javax.cache.annotation.CacheResolverFactory is specified on the annotation and the @CacheDefaults then the default CacheResolverFactory logic must be used.

Default CacheResolverFactory Rules:

1. Get the CacheManager to use via Caching.getCacheManager()
2. Call CacheManager.get(String) with the cache name
3. If a Cache is not returned
    a. Create the Cache using CacheManager.createCacheBuilder
4. Create a CacheResolver that wraps the found/created Cache and always returns the Cache.

If the CacheResolverFactory throws an exception the exception must be propagated up to the application code that triggered the execution of the CacheResolverFactory.

### 7.3.3 - CacheResolver

The CacheResolver is returned by the CacheResolver factory and is meant to be called on every invocation of the annotated method it was returned for, returning the Cache to use for that invocation.

If the CacheResolver throws an exception the exception must be propagated up to the application code that triggered the execution of the CacheResolverFactory.

## *7.4 - Key Generation*

The @CacheResult, @CachePut, and @CacheRemoveEntry annotations all require a cache key to be generated and all of these annotations allow for specification of a CacheKeyGenerator implementation.

The specified CacheKeyGenerator will be called once for every annotated method invocation. Information about the annotated method and the current invocation is provided by the CacheKeyInvocationContext. The method parameters the developer specified to be used in the key are contained in the CacheInvocationParameter array returned by the getKeyParameters() method. A custom CacheKeyGenerator can use whatever information at its disposal to generate the CacheKey.

If javax.cache.annotation.CacheKeyGenerator is specified on the annotation and the @CacheDefaults then the default CacheKeyGenerator logic must be used.

Default CacheKeyGenerator Rules:

1. Create an Object[] using CacheInvocationParameter.getValue() from the array returned by CacheKeyInvocationContext.getKeyParameters()
2. Create a CacheKey instance that wraps the Object[] and uses Arrays.deepHashCode to calculate its hashCode and Arrays.deepEquals for comparison to other keys.

If the CacheKeyGenerator throws an exception the exception must be propagated up to the application code that triggered the execution of the CacheKeyGenerator.

## *7.5 - Annotation Support Classes*

### 7.5.1 - CacheMethodDetails

Static information about a method with a caching annotation. Used by the CacheResolverFactory to determine the CacheResolver to use at runtime.

### 7.5.2 - CacheInvocationContext

Runtime information about the execution of a method with a caching annotation. Used by the CacheResolver to determine the Cache to use. Extends CacheMethodDetails so all static information is also available.

### 7.5.3 - CacheKeyInvocationContext

Runtime information about the execution of a method where key generation will take place (annotated with one of @CacheResult, @CachePut, or @CacheRemoveEntry). Used by the CacheKeyGenerator to create the CacheKey to use. Extends CacheInvocationContext so all standard runtime and static information is also available

### 7.5.4 - CacheInvocationParameter

Runtime information about a parameter for a method execution. Includes parameter annotations, position, type and value. Provided by CacheInvocationContext and CacheKeyInvocationContext

### 7.5.5 - CacheKey

Created by the CacheKeyGenerator interface the CacheKey is used as the key in any cache interacted with by the annotations. All CacheKeys must be immutable and serializable.

# Chapter 8 - Container and Provider Contracts for Deployment and Bootstrapping

In Java SE environments, the CacheManagerFactory.getCacheManager method may require the creation of a new CacheManager. To do this it locates an instance of the javax.cache.CacheManager.CacheManagerFactoryProvider interface.

A cache provider implementation running in a Java SE environment should also act as a service provider by supplying a service provider configuration file as described in the JAR File Specification

The provider configuration file serves to export the provider implementation class to the CacheManagerFactory bootstrap class, positioning the provider as the factory for cache managers. The provider supplies the provider configuration file by creating a text file named javax.cache.spi.CacheManagerFactoryProvider and placing it in the META-INF/services directory of one of its JAR files. The contents of the file should be the name of the provider implementation class of the javax.cache.CacheManager.CacheManagerFactoryProvider interface.

Example:
A persistence vendor called ACME caching products ships a JAR called acme.jar that contains its cache provider implementation. The JAR includes the provider configuration file.
acme.jar

```
    META-INF/services/javax.cache.spi.CacheManagerFactoryProvider
    com/acme/cache/CacheManagerFactoryProvider.class
    ...
```

The contents of the META-INF/services/javax.cache.spi.CacheManagerFactoryProvider file is nothing more than the name of the implementation class:
com.acme.cache.CacheManagerFactoryProvider

Cache provider jars may be installed or made available in the same ways as other service providers, e.g. as extensions or added to the application classpath according to the guidelines in the JAR File Specification. If more than one provider jar is registered the first one found by java.util.ServiceLoader will be used. If no provider is found, CacheManagerFactory.getCacheManager will thow an IllegalStateException.

## *8.2 Use of Caching*

The API provides a static means of accessing caching support through the class `javax.class.Caching`. Examples include

```
  // get the default cache manager
  CacheManager defaultCacheManager = Caching.getCacheManager();
  // the default cache manager can also be fetched by name
assert defaultCacheManager ==
Caching.getCacheManager(javax.cache.Caching.DEFAULT_CACHE_MANAGER_NAME);

  //Can get a non default named CacheManager
  CacheManager myCacheManager = Caching.getCacheManager("MyManager");
```

TODO: we need to talk about class loaders here particularly in the context of isolation and reclaiming resources.

# Chapter 9 - Glossary

| | |
|---|---|
| **CacheManager** | A container for caches, which holds references to them. |
| **Cache** | A collection of related items. |
| **Caching Unit** | A logical grouping under the control of a CacheManager |
| **Key** | A way of unambiguously identifying a unique item in a Cache. |
| **Value** | The value stored in a Cache. Any Java Object can be a value. |
| **CacheLoader** | A user-defined Class which is used to load key/value pairs into a Cache on demand. |
| **CacheWriter** | A user-defined Class which is used to write key/value pairs into a cache after a put operation. |
| **Cache Store** | A place where cache data is kept. Caches may have multiple stores. |
| **CacheEventListener** | A user-defined Class which listens to Cache events. |
| **Eviction Policy** | Method by which elements are evicted from the cache. E.g. FIFO, LFU, … |

# Chapter 10 - Bibliography


**[1]** Enterprise JavaBeans, v. 3.2. EJB Core Contracts and Requirements.

**[2]** JSR-250: Common Annotations for the Java Platform 1.1. *http://jcp.org/en/jsr/detail?id=250*.

**[3]** JSR-175: A Metadata Facility for the Java Programming language.h*ttp://jcp.org/en/jsr/detail?id=175*.

**[4]** JSR-221: JDBC 4.1 Specification. *http://java.sun.com/products/jdbc*.

**[5]** Enterprise JavaBeans, Simplified API, v 3.0. *http://java.sun.com/products/ejb*.

**[6]** JAR File Specification, *http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html*.

**[7]** JSR-220: Enterprise JavaBeans™ *,Version 3.0,* Java Persistence API

**[8]** JSR-336:Java SE

**[9]** JSR342: Java™ Platform, Enterprise Edition 7 (Java EE 7) Specification, http://www.jcp.org/en/jsr/detail?id=342

**[10]** JTA Specification 1.0.1, http://www.oracle.com/technetwork/java/javaee/jta/index.html

**[11]** CDI Specification, http://www.jcp.org/en/jsr/detail?id=299

**[12]** Unified EL is part of JSR-245, JSP2.1. See http://www.jcp.org/en/jsr/detail?id=245

**[13]** @CacheResult and Ehache prior art http://code.google.com/p/ehcache-spring-annotations/

**[14]** @CacheResult with Grails prior art http://gpc.github.com/grails-springcache/docs/manual/guide/4.%20Content%20Caching.html

**[15]** Information on properly implementing equals and hashCode (http://java.sun.com/developer/Books/effectivejava/Chapter3.pdf)

# Appendix A - Revision History

This appendix lists the significant changes that have been made during the development of JSR107.

### *Early Draft 1*

Created Document.

# Appendix B - Open Issues

Here we list open issues

## Statistics effects

In below

- Yes: indicates can modify statistics counter in RI
- <YES?>: indicates where RI does not, but maybe should
- <YES?>coh : indicates where RI does not, but maybe should (and does in Coherence implementation)

Points of reference:
Coherence does not maintain removals count
Coherence does not maintain getStartAccumulationDate()
Coherence uses double rather than float for averages

Greg's review comments in blue.

| Method | Puts | Removals | Hits | Misses |
|---|---|---|---|---|
| boolean containsKey(K key) No | | | <YES?>coh | <YES?>coh |
| V get(K key) | | | Yes | Yes |
| Map<K,V> getAll(Collection<? extends K> keys) | | | Yes | Yes |
| V getAndPut(K key, V value) No | Yes | | <YES?> | <YES?> |
| V getAndRemove(K key) | | Yes | Yes | Yes |
| V getAndReplace(K key, V value) | Yes | | Yes | Yes |
| Object invokeEntryProcessor(K key, EntryProcessor<K, V> entryProcessor); Not sure. This is really different. | <YES?> | <YES?> | <YES?> | <YES?> |

| | | | | |
|---|---|---|---|---|
| Iterator<Cache.Entry<K, V>> iterator() ? | | <YES?> | <YES?> | |
| Future<V> load(K key) | | | | |
| Future<Map<K,V>> loadAll(Collection<? extends K> keys) | | | | |
| void put(K key, V value) | Yes | | | |
| void putAll(Map<? extends K,? extends V> map) | Yes | | | |
| boolean putIfAbsent(K key, V value) No | Yes | | <YES?> | <YES?> |
| boolean remove(K key) | | Yes | | |
| boolean remove(K key, V oldValue) No | | Yes | <YES?> | <YES?> |
| void removeAll() | | Yes | | |
| void removeAll(Collection<? extends K> keys) | | Yes | | |
| boolean replace(K key, V value) N | Yes | | <YES?> | <YES?> |
| boolean replace(K key, V oldValue, V newValue) N | Yes | | <YES?> | <YES?> |

## Listeners

All listeners fire after the cache operation has completed and in the line of execution.
This means that a listener can not veto the operation taking place.
It can not alter the arguments to the operation.
It can not alter the return values except that it may throw a RuntimeException which will be wrapped as a

CacheEntryListenerException and propagated to the caller.
For a single operation a registered listener will be invoked at most once. If the VM(s) participating in the operation survive the operation it will fire exactly once.

Open Issue: do we stipulate that it fires in the VM where the request originated (in a distributed cache it could fire either in the calling VM or in the VM holding the primary data copy).

The table below summarises when listeners must be invoked. Conditions are on the state of the entry before the operation.

| **Method** | **CacheEntryCreatedListener** | **CacheEntryExpiredListener** | **CacheEntryReadListener** | **CacheEntryRemovedListener** | **CacheEntryUpdatedListener** |
|---|---|---|---|---|---|
| boolean containsKey(K key) | | | | | |
| V get(K key) | | | if there | | |
| Map<K,V> getAll(Collection<? extends K> keys) | | | if there | | |
| V getAndPut(K key, V value) | if missing | | if there | | if there |
| V getAndRemove(K key) | | | if there | if there | |
| V getAndReplace(K key, V value) | | | if there | | if there |
| Object invokeEntryProcessor(K key, EntryProcessor<K, V> entryProcessor); | ??? | | ??? | ??? | ??? |
| Iterator<Cache.Entry<K, V>> iterator() | | | if it.next().getValue() | if it.remove() | if it.next().setValue() |
| Future<V> load(K key) | if missing | | | | if there |
| Future<Map<K,V>> loadAll(Collection<? extends K> keys) | if missing | | | | if there |
| void put(K key, V value) | if missing | | | | if there |
| void putAll(Map<? extends K,? extends V> map) | if missing | | | | if there |
| boolean putIfAbsent(K key, V value) | if missing | | | | |
| boolean remove(K key) | | | | if there | |
| boolean remove(K key, V oldValue) | | | if there | if there && equal | |

| | | | | | |
|---|---|---|---|---|---|
| void removeAll() | | | | if there | |
| void removeAll(Collection<? extends K> keys) | | | | if there | |
| boolean replace(K key, V value) | | | | | if there |
| boolean replace(K key, V oldValue, V newValue) | | | | | if there && equal |

# Project Management

## *Resources Needed*

### People

Documentation Person - to format and proof read. 1 FTE for 2 weeks
TCK Team - to increase coverage and tighten up tests. 2 FTE months

### Tools

?