Java™ Caching API

The Java Caching API is an API for interacting with caching systems from Java programs

JSR107 Expert Group

Specification Leads:
Greg Luck
Brian Oliver, Oracle Corporation

16 December 2013

Comments to: jsr107@googlegroups.com

Contents

1. Introduction

This specification describes the objectives and functionality of the Java Caching Application Programming Interface ("API").

The Java Caching API provides a common way for Java programs to create, access, update and remove entries from caches.

1.1. Overview

Caching is a proven technique for dramatically increasing the performance and scalability of applications.

Caching involves keeping a temporary copy of information in a low-latency structure for some period of time so that future requests for the same information may be performed faster.

Applications that repetitively make use of information that is either expensive to create or access will typically benefit from caching. For example consider a servlet that creates a web page containing information obtained from multiple databases, network servers and expensive computations; this information might be reusable for the creation of later web pages, and if so, using caching to reuse previously created information can reduce page construction time.

The Java Caching API provides a common way for applications to use and adopt caching thus allowing developers to focus on application development and avoid the burden of implementing caches themselves. This specification defines caching terminology, semantics and a corresponding set of Java interfaces.

Caching products that implement the Java Caching API do so by supplying a Caching Provider that implements the Java Caching API interfaces.

1.2. What is Caching?

The term Caching is ubiquitous in computing. In the context of application design it is often used to describe the technique whereby application developers utilize a separate in-memory or low-latency data-structure, a Cache, to temporarily store, or cache, a copy of or reference to information that an application may reuse at some later point in time, thus alleviating the cost to re-access or re-create it.

In the context of the Java Caching API the term Caching describes the technique whereby Java developers use a Caching Provider to temporarily cache Java objects.

It is often assumed that information from a database is being cached. This however is not a requirement of caching. Fundamentally any information that is expensive or time consuming to produce or access 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 data*
- *servlet response caching*
- *caching of domain object graphs*

1.3. Objectives

The Java Caching API's objectives are to:

- provide applications with caching functionality, in particular the ability to cache Java objects;
- define a common set of caching concepts and facilities;
- minimize the number of concepts Java developers need to learn to adopt caching;
- maximize the portability of applications that use caching between caching implementations;
- support both in-process and distributed cache implementations;
- support caching Java objects by-value and optionally by-reference;
- define runtime cache annotations in accordance with JSR-175: A Metadata Facility for the Java Programming Language; so that Java developers making use of optionally provided annotation processors may declaratively specify application caching requirements; and

1.4. Non-Objectives

The Java Caching API does not address:

- Resource and Memory Constraint Configuration - While many caching implementations provide support for constraining the amount of resources caches may use at runtime, this specification does not define how this functionality is configured or represented. This specification does however define a standard mechanism for developers to specify how long information should be available to be cached.
- Cache Storage and Topology - This specification does not specify how caching implementations store or represent information that is cached.
- Administration - This specification does not specify how caches are administered. It does define mechanisms to programmatically configure caches and investigate cache statistics via Java Management Extensions (JMX).
- Security - This specification does not specify how cache content may be secured or how access and operations on caches can be controlled.
- External Resource Synchronization - This specification does not specify how an application or caching implementations should keep caches and external resource content synchronized.

  While developers may utilize read-through and write-through techniques as provided by the specification, these techniques are only effective when a cache is the only application mutating an external resource. Outside of this scenario cache synchronization can't be guaranteed.

1.5. Java SE and Java EE support

The Java Caching API is designed to be suitable for use by applications using the Standard and Enterprise Editions, versions 6 or newer.

A caching implementation:

- may choose to only work on a higher version of Java.
- may support its use by applications using Java EE, however this specification does not specify any standard for how that may be done.

1.6. Package

The top level package name for the Java Caching API is `javax.cache`.

1.7. Optional Features

All features in this specification are mandatory except for those enumerated in the `OptionalFeature` enum:

- storeByReference

If implemented, optional features must be implemented exactly as described in this specification.

A developer may determine which of the optional features have been implemented by a caching provider using the capabilities API. Given a Caching Provider instance, call `cachingProvider.isSupported(OptionalFeature feature)`.

Some optional features only make sense only in some contexts. For example, storeByReference is generally not supported or supportable by distributed caching topologies.

Optional features allow for a caching implementation to support the specification without necessarily supporting all the features, and allows end users and frameworks to discover what the features are so they can dynamically configure appropriate usage.

1.8. Document Conventions

The regular Arial (11 point) font is used for information that is normative for this specification.

The italic Arial (11 point) 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` (11 Point) font is used for inline code descriptions. Java code, examples and sample data fragments also use the `Courier New` font. There are formatted as below (in 10 point font):

```
package com.example.hello;
```

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

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.

1.9. Expert Group Members

This specification is being developed under the Java Community Process v2.9.

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

The following are expert group members:

- Greg Luck
- Brian Oliver, Oracle
- Cameron Purdy, Oracle
- Galder Zamarreño, Red Hat
- Nikita Ivanov, Grid Gain
- Chris Berry
- Jon Stevens
- Rick Hightower
- Ben Cotton, Credit Suisse
- David Mossakowski, Citigroup
- Bongjae Chang
- Steve Millidge
- Gabe Montero, IBM
- Brian Martin, IBM
- Eric Dalquist
- Pete Muir, Red Hat, Inc.
- William Newport, Goldman Sachs
- Ryan Gardner, Dealer.com
- Chris Dennis, Terracotta, Inc.
- Toshio Takeda, Fujitsu
- Chang Paek, TmaxSoft, Inc.

The following are official observers:

- Linda DeMichiel, Oracle
- Bill Shannon, Oracle

1.10. Acknowledgements

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

2. Fundamentals

2.1. Core Concepts

The Java Caching API defines five core interfaces: `CachingProvider`, `CacheManager`, `Cache`, `Entry` and `ExpiryPolicy`.

A `CachingProvider` defines the mechanism to establish, configure, acquire, manage and control zero or more `CacheManager`s. An application may access and use zero or more `CachingProvider`s at runtime.

A `CacheManager` defines the mechanism to establish, configure, acquire, manage and control zero or more uniquely named `Cache`s all within the context of the `CacheManager`. A `CacheManager` is owned by a single `CachingProvider`.

A `Cache` is a Map-like data-structure that permits the temporary storage of Key-based Values, some what like `java.util.Map` data-structure. A `Cache` is owned by a single `CacheManager`.

An `Entry` is a single key-value pair stored by a `Cache`.

Each entry stored by a cache has a defined duration, called the Expiry Duration, during which they may be accessed, updated and removed. Once this duration has passed, the entry is said to be Expired. Once expired, entries are no longer available to be accessed, updated or removed, just as if they never existed in a cache. Expiry is set using an `ExpiryPolicy`.

Store-By-Value and Store-By-Reference

Entries are stored by individual Caches using one of two mechanisms.

The default mechanism, called **store-by-value**, instructs an implementation to make a copy of application provided keys and values prior to storing them in a Cache and later to return a new copy of the entries when accessed from a Cache. The purpose of copying entries as they are stored in a Cache and again when they are returned from a Cache is to allow applications to continue mutating the state of the keys and values without causing side-effects to entries held by a Cache.

A simple approach implementations can use to make copies of keys and values is Java Serialization.

To ensure application portability between implementations, it is recommended that custom key and value classes implement and adopt standard Java Serialization when using store-by-value.

The mechanism an implementation uses to make a copy of the keys and values for an entry may be customizable. However to ensure application portability implementations must allow applications to solely make use of standard Java Serialization. Implementations must not obligate applications to adopt non-standard Java Serialization.

The alternative and optional mechanism, called **store-by-reference**, instructs a Cache implementation to simply store and return references to the application provided keys and values,

instead of making a copies as required by the store-by-value approach. Should an application later mutate the keys or values provided to a Cache using store-by-reference semantics, the side-effects of the mutations will be visible to those accessing the entries from the Cache, without an application having to update the Cache.

For caches implemented on the Java heap, store-by-reference is the faster storage technique.

2.2. Caches and Maps

While Caches and Maps share somewhat similar APIs, Caches are not Maps and Maps are not Caches. The following section outlines the main similarities and differences.

Like Map-based data-structures:

- Cache values are stored and accessed through an associated key.
- Each key may only be associated with a single value in a Cache.
- Great care must be exercised if mutable objects are used as keys. The behavior of a Cache is undefined if a key is mutated in a manner that affects equals comparisons when a key is used with a Cache.
- Caches depend on the concept of equality to determine when keys and values are the same. Consequently custom key and value classes should define a suitable implementation of the `Object.equals` method.
- Custom key classes should additionally provide a suitable implementation of the `Object.hashCode` method.

  Although recommended, implementations are not required to call either the `Object.hashCode` or `Object.equals` methods defined by custom key classes. Implementations are free to use optimizations whereby the invocation of these methods is avoided.

  As this specification does not define the concept of object equivalence it should be noted applications that make use of custom key classes and rely on implementation specific optimizations to determine equivalence may not be portable.

Unlike Map-based data-structures:

- Cache keys and values must not be null.

  Any attempt to use null for keys or values will result in a NullPointerException being thrown, regardless of the use.

- Entries may expire.

The process of ensuring Entries are no longer available to an application because they are no longer considered valid is called "expiry"

- Entries may be evicted.

  Caches are typically not configured to store an entire data set. Instead they are often used to store a small, frequently used subset of the an entire dataset.

  To ensure that the size of a `Cache` doesn't consume resources without bounds, a `Cache` implementation may define a policy to constrain the amount of resources a `Cache` may use at runtime by removing certain entries when a resource limit is exceeded.

  The process of removing entries from a `Cache` when it has exceeded a resource limit is called "eviction". When an `Entry` is removed from a `Cache` due to resource constraints, it is said to be "evicted".

  While the specification does not define the capacity of a cache, a sensible implementation will define mechanisms to represent desired capacity limits, together with suitable strategies to choose and evict entries once that capacity has been reached. For example: the LRU eviction strategy attempts to evict Least-Recently-Used entries.

  Some of the reasons that capacity is not defined in the specification are:

  - implementations may utilize multi-layered tiered storage structures and thus define capacity per tier. In such circumstances it is not possible to define an overall capacity for a Cache and doing so would be ambiguous.

  - implementations may define capacity in terms of bytes rather than entry count on each tier.

  - the relative cost of entries in terms of memory used is directly related to the internal representation of the implementation of an Entry at runtime.

- To support the compare-and-swap (CAS) operations, those that atomically compare and exchange values, custom value classes should provide a suitable implementation of `Object.equals`.

  Although recommended, implementations are not required to call the `Object.equals` method defined by custom value classes. Implementations are free to implement optimizations whereby the invocation of this method is avoided.

  As this specification does not define the concept of object equivalence it should be noted applications that make use of custom value classes and rely on implementation specific optimizations to determine equivalence may not be portable.

- Implementations may require `Key`s and `Value`s to be serializable in some manner.

- Caches may be configured to control how entries are stored, either using store-by-value or optionally using store-by-reference semantics.

- Implementations may optionally enforce security restrictions. In case of a violation, a SecurityException must be thrown.

2.3. Consistency

Consistency refers to the behavior of caches and the guarantees that exist when concurrent cache mutation occur together with the visibility of the mutations when multiple threads are accessing a cache.

All implementations must support the Default Consistency model as outlined below.

2.3.1. Default Consistency

When using the default consistency mode, most cache operations are performed as if a locking mechanism exists for each key in a Cache. When a cache operation acquires an exclusive read and write lock on a key all subsequent operations on that key will block until that lock is released. The consequences are that operations performed by a thread *happen-before* read or mutation operations performed by another thread, including threads in different Java Virtual Machines.

This can be understood as a pessimistic locking approach. Lock, mutate and unlock.

For some cache operations the value returned by a cache is considered the *last value*. The last value might be an old value or a new value, especially in the case where an entry is concurrently being updated. It is implementation dependent which is returned.

This can be understood as a lock free approach with no guaranteed consistency.

Other operations follow a different convention in that mutations may only occur when the current state of an entry matches a desired state. In such operations multiple threads are free to compete to apply these changes i.e. as if they share a lock. These are:

- `boolean putIfAbsent(K key, V value);`
- `boolean remove(K key, V oldValue);`
- `boolean replace(K key, V oldValue, V newValue);`
- `boolean replace(K key, V value);`
- `V getAndReplace(K key, V value);`

This can be understood as an optimistic locking approach; only apply a change if the state matches a known state, otherwise fail. These types of operations are also known as compare-and-swap (CAS) operations, after the CPU instructions that also operate in this manner.

As these methods must interact with other cache operations acting as if they had an exclusive lock, the CAS methods cannot write new values without acting as if they also had an exclusive lock.

As a result, in default consistency, while the CAS methods can allow a higher level of concurrency they will be held up by the non-CAS methods.

The following table shows the default consistency applicable to each Cache method.

Method	**Default Consistency**
`boolean containsKey(K key)`	last value
`V get(K key)`	happen-before
`Map<K,V> getAll(Collection<? extends K> keys)`	happen-before for each key individually but not for the Collection.
`V getAndPut(K key, V value)`	happen-before
`V getAndRemove(K key)`	happen-before
`V getAndReplace(K key, V value)`	happen-before plus compare and swap
`CacheManager getCacheManager()`	N/A
`CacheConfiguration getConfiguration()`	N/A
`String getName()`	N/A
`Iterator<Cache.Entry<K, V>> iterator()`	last value
`void loadAll(Set<? extends K> keys, boolean replaceExistingValues, CompletionListener listener)`	N/A
`void put(K key, V value)`	happen-before
`void putAll(Map<? extends K,? extends V> map)`	happen-before for each key individually but not for the Collection.
`boolean putIfAbsent(K key, V value)`	happen-before plus compare and swap
`boolean remove(K key)`	happen-before
`boolean remove(K key, V oldValue)`	happen-before plus compare and swap
`void removeAll()`	last value
`void removeAll(Set<? extends K> keys)`	happen-before for each key individually but not for the Collection.
`<T> T invoke(K key, EntryProcessor<K, V, T> entryProcessor, Object... arguments)entryProcessor);`	happen-before

`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	happen-before for each key individually but not for the Collection.
`boolean replace(K key, V value)`	happen-before plus compare and swap
`boolean replace(K key, V oldValue, V newValue)`	happen-before plus compare and swap
`<T> T unwrap(Class<T> cls)`	N/A

2.3.2. Further Consistency Models

An implementation may provide support for different consistency models in addition to the required Default Consistency model.

2.4. Cache Topologies

While the specification does not mandate particular cache topologies, it is cognizant that Cache entries may well be stored locally and/or distributed across multiple processes. Implementation may choose to support neither, one, both and/or other topologies.

This notion is expressed in the specification in a number of ways:

- Most mutating methods provide a signature with a `void` or low cost return types. For example, while `java.util.Map` provides the method `V put(K key, V value)` `javax.cache.Cache` provide `void put(K key, V value)`.

  Versions with a more expensive return type are also provided. An example is the `V getAndPut(K key, V value)` method on `Cache`. It returns the old value like `Map` does.

- By having creation semantics that do not assume in-process implementation, `Configuration` is `Serializable` so that it can be sent over the network. Developers may define implementations of `CacheEntryListener`, `ExpiryPolicy`, `CacheEntryFilter`, `CacheWriter` and `CacheLoader` and associate them with a cache. To support distributed topologies, a developer defines a Factory for their creation rather than the instance. The `Factory` interface is `Serializable`.

- The use of `Iterable` throughout the specification for return types and parameters that might be large.
  Methods that return an entire collection such as the Map method `keySet()` can be problematic.

  A Cache may be so large that a key set may not fit in available memory and it might also be

very network inefficient.

`Cache`, the listener methods on `CacheEntryListener`'s subinterfaces, and the batch methods on `CacheLoader` all use `Iterable`.

- No assumption is made as to where implementations of `CacheEntryListener`, `ExpiryPolicy`, `CacheEntryFilter`, `CacheWriter` and `CacheLoader` are instantiated and executed.

  In a distributed implementation these may all reside close to the data rather than in process with the application.

- `CachingProvider.getCacheManager(URI uri, ClassLoader classLoader)` returns a CacheManager with a specific `ClassLoader` and `URI`. *This enables implementations to instantiate multiple instances.*

2.5. Execution Context

`EntryProcessor`s, `CacheEntryListener`s, `CacheLoader`s, `CacheWriter`s and `ExpiryPolicy`s ("customizations") are instantiated and operate in the context of the `CacheManager` URI and `ClassLoader` in which they were configured. This means at deployment time, instances of these customizations must be available to and have access to application classes defined by the `ClassLoader` for a Cache.

Implementations may safely assume such customizations are available to a Cache using the ClassLoader as provided by a CacheManager.

How the availability of classes is achieved is implementation and deployment dependent.

For example: In a Java EE environment, application defined Customizations may be deployed within the scope of an enterprise application ear/war/jar.

While customizations may be available in the same `ClassLoader` as an application and thus have access to all application classes, to ensure portability application customizations must avoid direct access to deployment specific resources. Instead customizations should only attempt to access and mutate the Cache information and entries provided to them.

In implementations and deployment environments that support it, customizations may additionally utilize technologies such as resource injection (e.g: CDI) to allow direct access to application and deployment specific resources. There is no requirement however that implementations support this ability.

2.6. Reentrancy

While this specification does not constrain the operations a developer may perform when using custom `EntryProcessor`s, `CacheEntryListener`s, `CacheLoader`s, `CacheWriter`s and `ExpiryPolicy`s, caching implementations may restrict reentrancy from these interfaces. For example; an implementation may restrict the ability for an `EntryProcessor` to call methods on

`Cache`, or invoke other `EntryProcessor`s. Similarly an implementation may restrict the ability for a `CacheLoader`/`CacheWriter` to access a Cache.

Consequently developers are strongly recommended to avoid writing re-entrant implementations of these interfaces, as those implementations may not be portable.

3. A Simple Example

This simple example creates a default CacheManager, configures a cache on it called “simpleCache” with a key type of `String` and a value type of `Integer` and an expiry of one hour and then performs some cache operations.

```
 //resolve a cache manager
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();

//configure the cache
MutableConfiguration<String, Integer> config =
     new MutableConfiguration<>()
    .setTypes(String.class, Integer.class)
    .setExpiryPolicyFactory(AccessedExpiryPolicy.factoryOf(ONE_HOUR))
    .setStatisticsEnabled(true);

//create the cache
Cache<String, Integer> cache = cacheManager.createCache("simpleCache", config);

//cache operations
String key = "key";
Integer value1 = 1;
cache.put("key", value1);
Integer value2 = cache.get(key);
assertEquals(value1, value2);
cache.remove(key);
assertNull(cache.get(key));
```

Where the default `CachingProvider` and default `CacheManager` are being used, there is a static convenience method for getting a `Cache`, `Caching.getCache`:

```
//get the cache
Cache<String, Integer> cache = Caching.getCache("simpleCache",
    String.class, Integer.class);
```

4. CacheManagers

CacheManagers are a core concept of the Java Caching API. It is through CacheManagers that developers interact with caches.

A `CacheManager` provides:

- A means of establishing and configuring uniquely named caches.
- A means of acquiring a cache given its uniquely configured name.
- A means of scoping uniquely named caches; caches of the same name but originating from different Cache Managers are considered different caches.
- A means of closing a cache so that it is no longer managed.
- A means of destroying a cache including all of its contents.
- The `ClassLoader` that caches will use, should they require it, for resolving and loading application classes.
- A means of iterating over the currently managed caches.
- A means to close the `CacheManager` together with all of the currently managed caches.
- A means to enable and disable statistics gathering for caches.
- A means to enable and disable JMX management of caches.
- A means of acquiring `CachingProvider` specific properties defined for the `CacheManager`.
- A means of querying the capabilities and optional features supported by the `CachingProvider`.

The `CacheManager` interface is defined as follows:

```
/**
 * A {@link CacheManager} provides a means of establishing, configuring,
 * acquiring, closing and destroying uniquely named {@link Cache}s.
 * <p>
 * {@link Cache}s produced and owned by a {@link CacheManager} typically share
 * common infrastructure, for example, a common {@link ClassLoader} and
 * implementation specific {@link Properties}.
 * <p>
 * Implementations of {@link CacheManager} may additionally provide and share
 * external resources between the {@link Cache}s being managed, for example,
 * the content of the managed {@link Cache}s may be stored in the same cluster.
```

```
 * <p>
 * By default {@link CacheManager} instances are typically acquired through the
 * use of a {@link CachingProvider}.  Implementations however may additionally
 * provide other mechanisms to create, acquire, manage and configure
 * {@link CacheManager}s, including:
 * <ul>
 * <li>making use of {@link java.util.ServiceLoader}s,</li>
 * <li>permitting the use of the <code>new</code> operator to create a
 * concrete implementation, </li>
 * <li>providing the construction through the use of one or more
 * builders, and</li>
 * <li>through the use of dependency injection.</li>
 * </ul>
 * <p>
 * The default {@link CacheManager} however can always be acquired using the
 * default configured {@link CachingProvider} obtained by the {@link Caching}
 * class.  For example:
 * <pre><code>
 * CachingProvider provider = Caching.getCachingProvider();
 * CacheManager manager = provider.getCacheManager();
 * </code></pre>
 * <p>
 * Within a Java process {@link CacheManager}s and the {@link Cache}s they
 * manage are scoped and uniquely identified by a {@link URI},  the meaning of
 * which is implementation specific. To obtain the default {@link URI},
 * {@link ClassLoader} and {@link Properties} for an implementation, consult the
 * {@link CachingProvider} class.
 *
 * @author Greg Luck
 * @author Yannis Cosmadopoulos
 * @author Brian Oliver
 * @see Caching
 * @see CachingProvider
 * @see Cache
 * @since 1.0
 */
public interface CacheManager extends Closeable {

  /**
   * Get the {@link CachingProvider} that created and is responsible for
   * the {@link CacheManager}.
   *
   * @return the CachingProvider or <code>null</code> if the {@link CacheManager}
   *         was created without using a {@link CachingProvider}
   */
  CachingProvider getCachingProvider();

  /**
   * Get the URI of the {@link CacheManager}.
   *
   * @return the URI of the {@link CacheManager}
```

```
 */
URI getURI();

/**
 * Get the {@link ClassLoader} used by the {@link CacheManager}.
 *
 * @return  the {@link ClassLoader} used by the {@link CacheManager}
 */
ClassLoader getClassLoader();

/**
 * Get the {@link Properties} that were used to create this
 * {@link CacheManager}.
 * <p>
 * Implementations are not required to re-configure the
 * {@link CacheManager} should modifications to the returned
 * {@link Properties} be made.
 *
 * @return the Properties used to create the {@link CacheManager}
 */
Properties getProperties();

/**
 * Creates a named {@link Cache} at runtime.
 * <p>
 * If a {@link Cache} with the specified name is known to the {@link
 * CacheManager}, a CacheException is thrown.
 * <p>
 * If a {@link Cache} with the specified name is unknown the {@link
 * CacheManager}, one is created according to the provided {@link Configuration}
 * after which it becomes managed by the {@link CacheManager}.
 * <p>
 * Prior to a {@link Cache} being created, the provided {@link Configuration}s is
 * validated within the context of the {@link CacheManager} properties and
 * implementation.
 * <p>
 * Implementers should be aware that the {@link Configuration} may be used to
 * configure other {@link Cache}s.
 * <p>
 * There's no requirement on the part of a developer to call this method for
 * each {@link Cache} an application may use.  Implementations may support
 * the use of declarative mechanisms to pre-configure {@link Cache}s, thus
 * removing the requirement to configure them in an application.  In such
 * circumstances a developer may simply call either the
 * {@link #getCache(String)} or {@link #getCache(String, Class, Class)}
 * methods to acquire a previously established or pre-configured {@link Cache}.
 *
 * @param cacheName     the name of the {@link Cache}
 * @param configuration a {@link Configuration} for the {@link Cache}
 * @throws IllegalStateException         if the {@link CacheManager}
 *                                       {@link #isClosed()}
```

```
     * @throws CacheException                if there was an error configuring the
     *                                       {@link Cache}, which includes trying
     *                                       to create a cache that already exists.
     * @throws IllegalArgumentException      if the configuration is invalid
     * @throws UnsupportedOperationException if the configuration specifies
     *                                       an unsupported feature
     * @throws NullPointerException          if the cache configuration or name
     *                                       is null
     * @throws SecurityException            when the operation could not be performed
     *                                       due to the current security settings
     */
    <K, V, C extends Configuration<K, V>> Cache<K, V> createCache(String cacheName,
                                                                  C configuration)
        throws IllegalArgumentException;


    /**
     * Looks up a managed {@link Cache} given its name.
     * <p>
     * This method must be used for {@link Cache}s that were configured with
     * runtime key and value types. Use {@link #getCache(String)} for
     * {@link Cache}s where these were not specified.
     * <p>
     * Implementations must ensure that the key and value types are the same as
     * those configured for the {@link Cache} prior to returning from this method.
     * <p>
     * Implementations may further perform type checking on mutative cache operations
     * and throw a {@link ClassCastException} if these checks fail.
     * <p>
     * Implementations that support declarative mechanisms for pre-configuring
     * {@link Cache}s may return a pre-configured {@link Cache} instead of
     * <code>null</code>.
     *
     * @param cacheName the name of the managed {@link Cache} to acquire
     * @param keyType   the expected {@link Class} of the key
     * @param valueType the expected {@link Class} of the value
     * @return the Cache or null if it does exist or can't be pre-configured
     * @throws IllegalStateException    if the {@link CacheManager}
     *                                  is {@link #isClosed()}
     * @throws IllegalArgumentException if the specified key and/or value types are
     *                                  incompatible with the configured cache.
     * @throws SecurityException        when the operation could not be performed
     *                                  due to the current security settings
     */
    <K, V> Cache<K, V> getCache(String cacheName, Class<K> keyType,
                                Class<V> valueType);

    /**
     * Looks up a managed {@link Cache} given its name.
     * <p>
     * This method may only be used to acquire {@link Cache}s that were
```

```
     * configured without runtime key and value types, or were configured
     * to use Object.class key and value types.
     * <p>
     * Use the {@link #getCache(String, Class, Class)} method to acquire
     * {@link Cache}s that were configured with specific runtime types.
     * <p>
     * Implementations must check if key and value types were configured
     * for the requested {@link Cache}. If either the keyType or valueType of the
     * configured {@link Cache} were specified (other than <code>Object.class</code>)
     * an {@link IllegalArgumentException} will be thrown.
     * <p>
     * Implementations that support declarative mechanisms for pre-configuring
     * {@link Cache}s may return a pre-configured {@link Cache} instead of
     * <code>null</code>.
     *
     * @param cacheName the name of the cache to look for
     * @return the Cache or null if it does exist or can't be pre-configured
     * @throws IllegalStateException    if the CacheManager is {@link #isClosed()}
     * @throws IllegalArgumentException if the {@link Cache} was configured with
     *                                  specific types, this method cannot be used
     * @throws SecurityException        when the operation could not be performed
     *                                  due to the current security settings
     * @see #getCache(String, Class, Class)
     */
    <K, V> Cache<K, V> getCache(String cacheName);

    /**
     * Obtains an {@link Iterable} over the names of {@link Cache}s managed by the
     * {@link CacheManager}.
     * <p>
     * {@link java.util.Iterator}s returned by the {@link Iterable} are immutable.
     * Any modification of the {@link java.util.Iterator}, including remove, will
     * raise an {@link IllegalStateException}.  If the {@link Cache}s managed by
     * the {@link CacheManager} change, the {@link Iterable} and
     * associated {@link java.util.Iterator}s are not affected.
     * <p>
     * {@link java.util.Iterator}s returned by the {@link Iterable} may not provide
     * all of the {@link Cache}s managed by the {@link CacheManager}.  For example:
     * Internally defined or platform specific {@link Cache}s that may be accessible
     * by a call to {@link #getCache(String)} or {@link #getCache(String, Class,
     * Class)} may not be present in an iteration.
     *
     * @return an {@link Iterable} over the names of managed {@link Cache}s.
     * @throws IllegalStateException if the {@link CacheManager}
     *                               is {@link #isClosed()}
     * @throws SecurityException     when the operation could not be performed
     *                               due to the current security settings
     */
    Iterable<String> getCacheNames();

    /**
```

```
 * Destroys a specifically named and managed {@link Cache}.  Once destroyed
 * a new {@link Cache} of the same name but with a different {@link
 * Configuration} may be configured.
 * <p>
 * This is equivalent to the following sequence of method calls:
 * <ol>
 * <li>{@link Cache#clear()}</li>
 * <li>{@link Cache#close()}</li>
 * </ol>
 * followed by allowing the name of the {@link Cache} to be used for other
 * {@link Cache} configurations.
 * <p>
 * From the time this method is called, the specified {@link Cache} is not
 * available for operational use. An attempt to call an operational method on
 * the {@link Cache} will throw an {@link IllegalStateException}.
 *
 * @param cacheName the cache to destroy
 * @throws IllegalStateException if the {@link CacheManager}
 *                               {@link #isClosed()}
 * @throws NullPointerException  if cacheName is null
 * @throws SecurityException     when the operation could not be performed
 *                               due to the current security settings
 */
void destroyCache(String cacheName);

/**
 * Controls whether management is enabled. If enabled the {@link CacheMXBean}
 * for each cache is registered in the platform MBean server. The platform
 * MBeanServer is obtained using
 * {@link ManagementFactory#getPlatformMBeanServer()}.
 * <p>
 * Management information includes the name and configuration information for
 * the cache.
 * <p>
 * Each cache's management object must be registered with an ObjectName that
 * is unique and has the following type and attributes:
 * <p>
 * Type:
 * <code>javax.cache:type=Cache</code>
 * <p>
 * Required Attributes:
 * <ul>
 * <li>CacheManager the name of the CacheManager
 * <li>Cache the name of the Cache
 * </ul>
 *
 * @param cacheName the name of the cache to register
 * @param enabled   true to enable management, false to disable.
 * @throws IllegalStateException if the {@link CacheManager} or
 *                               {@link Cache} {@link #isClosed()}
 * @throws SecurityException     when the operation could not be performed
 *                               due to the current security settings
```

```
 */
void enableManagement(String cacheName, boolean enabled);

/**
 * Enables or disables statistics gathering for a managed {@link Cache} at
 * runtime.
 * <p>
 * Each cache's statistics object must be registered with an ObjectName that
 * is unique and has the following type and attributes:
 * <p>
 * Type:
 * <code>javax.cache:type=CacheStatistics</code>
 * <p>
 * Required Attributes:
 * <ul>
 * <li>CacheManager the name of the CacheManager
 * <li>Cache the name of the Cache
 * </ul>
 *
 * @param cacheName the name of the cache to register
 * @param enabled   true to enable statistics, false to disable.
 * @throws IllegalStateException if the {@link CacheManager} or
 *                               {@link Cache} {@link #isClosed()}
 * @throws NullPointerException  if cacheName is null
 * @throws SecurityException when the operation could not be performed
 *                           due to the current security settings
 */
void enableStatistics(String cacheName, boolean enabled);

/**
 * Closes the {@link CacheManager}.
 * <p>
 * For each {@link Cache} managed by the {@link CacheManager}, the
 * {@link Cache#close()} method will be invoked, in no guaranteed order.
 * <p>
 * If a {@link Cache#close()} call throws an exception, the exception will be
 * ignored.
 * <p>
 * After executing this method, the {@link #isClosed()} method will return
 * <code>true</code>.
 * <p>
 * All attempts to close a previously closed {@link CacheManager} will be
 * ignored.
 *
 * @throws SecurityException when the operation could not be performed due to the
 *         current security settings
 */
void close();

/**
 * Determines whether the {@link CacheManager} instance has been closed. A
```

```
   * {@link CacheManager} is considered closed if;
   * <ol>
   * <li>the {@link #close()} method has been called</li>
   * <li>the associated {@link #getCachingProvider()} has been closed, or</li>
   * <li>the {@link CacheManager} has been closed using the associated
   * {@link #getCachingProvider()}</li>
   * </ol>
   * <p>
   * This method generally cannot be called to determine whether the
   * {@link CacheManager} is valid or invalid. A typical client can determine
   * that a {@link CacheManager} is invalid by catching any exceptions that
   * might be thrown when an operation is attempted.
   *
   * @return true if this {@link CacheManager} instance is closed; false if it
   *         is still open
   */
  boolean isClosed();

  /**
   * Provides a standard mechanism to access the underlying concrete caching
   * implementation to provide access to further, proprietary features.
   * <p>
   * If the provider's implementation does not support the specified class,
   * the {@link IllegalArgumentException} is thrown.
   *
   * @param clazz the proprietary class or interface of the underlying concrete
   *              {@link CacheManager}. It is this type that is returned.
   * @return an instance of the underlying concrete {@link CacheManager}
   * @throws IllegalArgumentException if the caching provider doesn't support the
   *                                  specified class.
   * @throws SecurityException        when the operation could not be performed
   *                                  due to the current security settings
   */
  <T> T unwrap(java.lang.Class<T> clazz);
}
```

4.1.1. Acquiring a default CacheManager

To ease adoption of the Java Caching API developers may acquire a default `CacheManager` from a default `CachingProvider` by using the `Caching` helper class. For example:

```
//acquire the default CachingProvider
CachingProvider provider = Caching.getCachingProvider();

//acquire the default CacheManager
CacheManager manager = provider.getCacheManager();
```

To acquire non-default or alternative configurations of `CacheManager`s, for example with custom `ClassLoader`s or caching implementor properties, developers should use one of the overloaded `CachingProvider` `getCacheManager` methods.

How to configure a CachingProvider is covered in the section on CachingProviders.

4.1.2. Configuring Caches

There are two approaches for configuring caches with `CacheManager`s:

- `CacheManager`s must allow applications to programmatically configure caches at runtime through the `<K, V, C extends Configuration<K, V>> Cache<K, V> createCache(String cacheName, C configuration)`method.
- `CacheManager`s may optionally provide mechanisms to declaratively configure caches for applications thus avoiding the need for applications to use the `createCache` method.

The mechanism(s) a `CacheManager` may allow the declarative definition of caches for an application is implementation dependent. One approach is to have a XML configuration file that configures a `CacheManager` and the `Cache`s in it.

Interfaces and Classes related to cache configuration are defined in the `javax.cache.configuration` package. A minimal configuration interface, `Configuration`, and a complete one, `CompleteConfiguration`, are provided.

While a minimal configuration interface is provided by this specification, only implementations that support the complete use of the CompleteConfiguration interface will be compliant to this specification.

The `javax.cache.configuration.Configuration` interface is defined as follows:

```
/**
 * A basic read-only representation of a {@link Cache} configuration.
 * <p>
 * The properties provided by instances of this interface are used by
 * {@link CacheManager}s to configure {@link Cache}s.
 * <p>
 * Implementations of this interface must override {@link Object#hashCode()} and
 * {@link Object#equals(Object)} as {@link Configuration}s are often compared at
 * runtime.
 *
 * @param <K> the type of keys maintained the cache
 * @param <V> the type of cached values
 * @author Greg Luck
 * @author Brian Oliver
 * @since 1.0
 */
public interface Configuration<K, V> extends Serializable {

  /**
```

```
   * Determines the required type of keys for {@link Cache}s configured
   * with this {@link Configuration}.
   *
   * @return the key type or <code>Object.class</code> if the type is undefined
   */
  Class<K> getKeyType();

  /**
   * Determines the required type of values for {@link Cache}s configured
   * with this {@link Configuration}.
   *
   * @return the value type or <code>Object.class</code> if the type is undefined
   */
  Class<V> getValueType();

  /**
   * Whether storeByValue (true) or storeByReference (false).
   * When true, both keys and values are stored by value.
   * <p>
   * When false, both keys and values are stored by reference.
   * Caches stored by reference are capable of mutation by any threads holding
   * the reference. The effects are:
   * <ul>
   * <li>if the key is mutated, then the key may not be retrievable or
   * removable</li>
   * <li>if the value is mutated, then all threads in the JVM can potentially
   * observe those mutations, subject to the normal Java Memory Model rules.</li>
   * </ul>
   * Storage by reference only applies to the local heap. If an entry is moved off
   * heap it will need to be transformed into a representation. Any mutations that
   * occur after transformation may not be reflected in the cache.
   * <p>
   * When a cache is storeByValue, any mutation to the key or value does not
   * affect the key of value stored in the cache.
   * <p>
   * The default value is <code>true</code>.
   *
   * @return true if the cache is store by value
   */
  boolean isStoreByValue();
}
```

The `javax.cache.configuration.CompleteConfiguration` interface is defined as follows:

```
/**
 * A read-only representation of the complete JCache {@link javax.cache.Cache}
 * configuration.
 * <p>
 * The properties provided by instances of this interface are used by
 * {@link javax.cache.CacheManager}s to configure {@link javax.cache.Cache}s.
 * <p>
```

```
 * Implementations of this interface must override {@link Object#hashCode()} and
 * {@link Object#equals(Object)} as
 * {@link javax.cache.configuration.CompleteConfiguration}s are often compared at
 * runtime.
 *
 * @param <K> the type of keys maintained the cache
 * @param <V> the type of cached values
 * @author Greg Luck
 * @author Yannis Cosmadopoulos
 * @author Brian Oliver
 * @since 1.0
 */
public interface CompleteConfiguration<K, V> extends Configuration<K, V>,
    Serializable {

  /**
   * Determines if a {@link javax.cache.Cache} should operate in read-through mode.
   * <p>
   * When in "read-through" mode, cache misses that occur due to cache entries
   * not existing as a result of performing a "get" will appropriately
   * cause the configured {@link javax.cache.integration.CacheLoader} to be
   * invoked.
   * <p>
   * The default value is <code>false</code>.
   *
   * @return <code>true</code> when a {@link javax.cache.Cache} is in
   * "read-through" mode.
   * @see #getCacheLoaderFactory()
   */
  boolean isReadThrough();

  /**
   * Determines if a {@link javax.cache.Cache} should operate in write-through
   * mode.
   * <p>
   * When in "write-through" mode, cache updates that occur as a result of
   * performing "put" operations called via one of
   * {@link javax.cache.Cache#put(Object, Object)},
   * {@link javax.cache.Cache#getAndRemove(Object)},
   * {@link javax.cache.Cache#removeAll()},
   * {@link javax.cache.Cache#getAndPut(Object, Object)}
   * {@link javax.cache.Cache#getAndRemove(Object)},
   * {@link javax.cache.Cache#getAndReplace(Object,
   * Object)}, {@link javax.cache.Cache#invoke(Object,
   * javax.cache.processor.EntryProcessor,
   * Object...)}, {@link javax.cache.Cache#invokeAll(java.util.Set,
   * javax.cache.processor.EntryProcessor, Object...)} will appropriately cause
   * the configured {@link javax.cache.integration.CacheWriter} to be invoked.
   * <p>
   * The default value is <code>false</code>.
   *
```

```
 * @return <code>true</code> when a {@link javax.cache.Cache} is in
 *         "write-through" mode.
 * @see #getCacheWriterFactory()
 */
boolean isWriteThrough();

/**
 * Checks whether statistics collection is enabled in this cache.
 * <p>
 * The default value is <code>false</code>.
 *
 * @return true if statistics collection is enabled
 */
boolean isStatisticsEnabled();

/**
 * Checks whether management is enabled on this cache.
 * <p>
 * The default value is <code>false</code>.
 *
 * @return true if management is enabled
 */
boolean isManagementEnabled();

/**
 * Obtains the {@link javax.cache.configuration.CacheEntryListenerConfiguration}s
 * for {@link javax.cache.event.CacheEntryListener}s to be configured on a
 * {@link javax.cache.Cache}.
 *
 * @return an {@link Iterable} over the
 * {@link javax.cache.configuration.CacheEntryListenerConfiguration}s
 */
Iterable<CacheEntryListenerConfiguration<K,
    V>> getCacheEntryListenerConfigurations();

/**
 * Gets the {@link javax.cache.configuration.Factory} for the
 * {@link javax.cache.integration.CacheLoader}, if any.
 * <p>
 * A CacheLoader should be configured for "Read Through" caches to load values
 * when a cache miss occurs using either the
 * {@link javax.cache.Cache#get(Object)} and/or
 * {@link javax.cache.Cache#getAll(java.util.Set)} methods.
 * <p>
 * The default value is <code>null</code>.
 *
 * @return the {@link javax.cache.configuration.Factory} for the
 * {@link javax.cache.integration.CacheLoader} or null if none has been set.
 */
Factory<CacheLoader<K, V>> getCacheLoaderFactory();
```

```
  /**
   * Gets the {@link javax.cache.configuration.Factory} for the
   * {@link javax.cache.integration.CacheWriter}, if any.
   * <p>
   * The default value is <code>null</code>.
   *
   * @return the {@link javax.cache.configuration.Factory} for the
   * {@link javax.cache.integration.CacheWriter} or null if none has been set.
   */
  Factory<CacheWriter<? super K, ? super V>> getCacheWriterFactory();

  /**
   * Gets the {@link javax.cache.configuration.Factory} for the
   * {@link javax.cache.expiry.ExpiryPolicy} to be used for caches.
   * <p>
   * The default value is a {@link javax.cache.configuration.Factory} that will
   * produce a {@link javax.cache.expiry.EternalExpiryPolicy} instance.
   *
   * @return the {@link javax.cache.configuration.Factory} for
   * {@link javax.cache.expiry.ExpiryPolicy} (must not be <code>null</code>)
   */
  Factory<ExpiryPolicy> getExpiryPolicyFactory();

}
```

To ease configuration of caches the Java Caching API provides an implementation of the `javax.cache.configuration.CompleteConfiguration` interface called `javax.cache.configuration.MutableConfiguration`.

Caching implementations may choose to provide additional implementations of the `Configuration` interface in order to provide implementation specific configuration.

To simplify programmatic configuration when using the `MutableConfiguration` class all setter methods return the `MutableConfiguration` instance thus allowing the class to be used in a fluent manner.

`CacheManager`s have the responsibility to validate `Cache` configurations that are provided by applications. Should a `Cache` configuration be invalid for a `CacheManager`, attempting to create the `Cache` will throw an `IllegalArgumentException`.

Commonly used constructors and setter methods of the `MutableConfiguration` class are defined as follows:

```
  /**
   * Constructs a default {@link MutableConfiguration}.
   */
  public MutableConfiguration()
```

```
/**
 * Constructs a {@link MutableConfiguration} based on another
 * {@link CompleteConfiguration}.
 *
 * @param configuration the {@link CompleteConfiguration}
 */
public MutableConfiguration(CompleteConfiguration<K, V> configuration)

/**
 * Sets the expected type of keys and values for a {@link Cache}
 * configured with this {@link Configuration}. Setting both to
 * <code>Object.class</code> means type-safety checks are not required.
 * <p>
 * This is used by {@link CacheManager} to ensure that the key and value
 * types are the same as those configured for the {@link Cache} prior to
 * returning a requested cache from this method.
 * <p>
 * Implementations may further perform type checking on mutative cache operations
 * and throw a {@link ClassCastException} if these checks fail.
 *
 * @param keyType   the expected key type
 * @param valueType the expected value type
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 * @throws NullPointerException should the key or value type be null
 * @see CacheManager#getCache(String, Class, Class)
 */
public MutableConfiguration<K, V> setTypes(Class<K> keyType, Class<V> valueType)


/**
 * Add a configuration for a {@link CacheEntryListener}.
 *
 * @param cacheEntryListenerConfiguration the
 *  {@link CacheEntryListenerConfiguration}
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 * @throws IllegalArgumentException is the same CacheEntryListenerConfiguration
 * is used more than once
 */
public MutableConfiguration<K, V> addCacheEntryListenerConfiguration(
    CacheEntryListenerConfiguration<K, V> cacheEntryListenerConfiguration)


/**
 * Remove a configuration for a {@link CacheEntryListener}.
 *
 * @param cacheEntryListenerConfiguration  the
 *     {@link CacheEntryListenerConfiguration} to remove
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> removeCacheEntryListenerConfiguration(
    CacheEntryListenerConfiguration<K, V> cacheEntryListenerConfiguration)
```

```
/**
 * Set the {@link CacheLoader} factory.
 *
 * @param factory the {@link CacheLoader} {@link Factory}
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setCacheLoaderFactory(Factory<? extends
    CacheLoader<K, V>> factory)

/**
 * Set the {@link CacheWriter} factory.
 *
 * @param factory the {@link CacheWriter} {@link Factory}
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setCacheWriterFactory(Factory<? extends
    CacheWriter<? super K, ? super V>> factory)

/**
 * Set the {@link Factory} for the {@link ExpiryPolicy}.  If <code>null</code>
 * is specified the default {@link ExpiryPolicy} is used.
 *
 * @param factory the {@link ExpiryPolicy} {@link Factory}
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setExpiryPolicyFactory(Factory<? extends
    ExpiryPolicy> factory)

/**
 * Set if read-through caching should be used.
 * <p>
 * It is an invalid configuration to set this to true without specifying a
 * {@link CacheLoader} {@link Factory}.
 *
 * @param isReadThrough <code>true</code> if read-through is required
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setReadThrough(boolean isReadThrough)


/**
 * Set if write-through caching should be used.
 * <p>
 * It is an invalid configuration to set this to true without specifying a
 * {@link CacheWriter} {@link Factory}.
 *
 * @param isWriteThrough <code>true</code> if write-through is required
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
```

```
public MutableConfiguration<K, V> setWriteThrough(boolean isWriteThrough)


/**
 * Set if a configured cache should use store-by-value or store-by-reference
 * semantics.
 *
 * @param isStoreByValue <code>true</code> if store-by-value is required,
 *                       <code>false</code> for store-by-reference
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setStoreByValue(boolean isStoreByValue)

/**
 * Sets whether statistics gathering is enabled on a cache.
 * <p>
 * Statistics may be enabled or disabled at runtime via
 * {@link CacheManager#enableStatistics(String, boolean)}.
 *
 * @param enabled true to enable statistics, false to disable.
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setStatisticsEnabled(boolean enabled)

/**
 * Sets whether management is enabled on a cache.
 * <p>
 * Management may be enabled or disabled at runtime via
 * {@link CacheManager#enableManagement(String, boolean)}.
 *
 * @param enabled true to enable statistics, false to disable.
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 */
public MutableConfiguration<K, V> setManagementEnabled(boolean enabled)
```

Each of the configuration options provided by the `MutableConfiguration` class and thus the `javax.cache.configuration.Configuration` interface are discussed in depth in later sections of this document.

The following defaults are provided by a new instance of the `MutableConfiguration` class.

Configuration Option	Type	Default Value(s)
Key Type	`Class<?>`	`Object.class`
Value Type	`Class<?>`	`Object.class`

Cache Loader Factory	`Factory<CacheLoader<K, V>>`	`null`
Cache Writer Factory	`Factory<CacheWriter<? super K, ? super V>>`	`null`
Expiry Policy Factory	`Factory<ExpiryPolicy<K>>`	a factory producing an `EternalExpiryPolicy`
Read Through Enabled	`boolean`	`false`
Write Through Enabled	`boolean`	`false`
Cache Entry Listener Configuration	`Iterable<CacheEntryListenerConfiguration<? super K, ? super V>>`	an empty iteration
Statistics Enabled	`boolean`	`false`
Management Enabled	`boolean`	`false`

4.1.3. Cache Names and Cache Scoping

`Cache`s are identified by their uniquely configured name in the scope of the `CacheManager` that was used to create or initially access them.

While Cache names are represented as Java `String`s, there are some restrictions and recommended naming conventions for portability. These are as follows:

- Cache Names starting with `java.` or `javax.`should not be used as these name spaces may be used for internal platform Caches.
- Cache Names should not contain forward slashes (`/`) or full-colons (`:`) as they may be used within Java EE environments for JNDI-based lookups.
- Cache Names may use Unicode characters.

While not a requirement, applications may choose to use the fully-qualified-class-name of the value class for a Cache as the Cache name. For example: A Cache that stores Orders might be called "com.mycompany.Orders".

4.1.4. Acquiring Caches

There are two approaches for acquiring caches with `CacheManager`s:

- When a type-safe `Cache` is required, which is one that attempts to ensure that the correct and expected types of cache entries are used, an application should use the following `CacheManager` method:

```
<K, V> Cache<K, V> getCache(String cacheName,
                            Class<K> keyType,
                            Class<V> valueType)
```

- When an application is explicitly taking responsibility for cache entry type-safety through the use of rawtypes, the following `CacheManager` method should be used:

```
<K, V> Cache<K, V> Cache getCache(String cacheName);
```

For more information on Cache type-safety see the section on Cache Type-Safety.

A simple example of how to acquire a Cache from a CacheManager:

```
Cache<String, Integer> cache = cacheManager.getCache(
    "simpleCache", String.class, Integer.class);
```

4.1.5. Cache and CacheManager Lifecycle

All Cache and CacheManager instances operate in one of two possible states; opened or closed. When open, instances may be used operationally to make requests. For example; creating, updating, removing an entry or configuring, acquiring, closing, removing a cache and so on. When closed, any operational use of these instances will throw an `IllegalStateException`.

4.1.6. Closing Caches

Closing a `Cache` via a call to the `Cache.close()` method signals to the CacheManager that produced or owns the `Cache` that it should no longer be managed. At this point in time the `CacheManager`:

- must close and release all resources being coordinated on behalf of the `Cache` by the `CacheManager`. This includes calling the `close` method on configured `CacheLoader`, `CacheWriter`, registered `CacheEntryListener`s and `ExpiryPolicy` instances that implement the `java.io.Closeable` interface,

- prevent events being delivered to configured `CacheEntryListener`s registered on the `Cache`,

- not return the name of the `Cache` when the `CacheManager getCacheNames()` method is called.

Once closed any attempt to use an operational method on a `Cache` will throw an `IllegalStateException`. The operational methods on `Cache` are:

- `clear`
- `containsKey`
- `deregisterCacheEntryListener`
- `get`
- `getAll`
- `getAndPut`
- `getAndRemove`
- `getAndReplace`
- `invoke`
- `invokeAll`
- `iterator`

- `loadAll`
- `put`
- `putAll`
- `putIfAbsent`
- `registerCacheEntryListener`
- `remove`
- `removeAll`
- `replace`

Closing a Cache does not necessarily destroy the contents of a Cache. It simply signals to the owning CacheManager that the Cache is no longer required by the application and that future uses of a specific Cache instance should not be permitted. Depending on the implementation and Cache topology, e.g., a storage-backed or distributed caches, the contents of a closed Cache may still be available and accessible by other applications or in fact via the Cache Manager that previously owned the Cache if an application calls `getCache` at some point in the future.

4.1.7. Destroying Caches

To destroy a Cache, release it from being managed and drop all of the cache entries, thus allowing a new cache, with the same name but possibly a different configuration to be created, the `CacheManager destroyCache` method should be called.

```
/**
 * Destroys a specifically named and managed {@link Cache}.  Once destroyed
 * a new {@link Cache} of the same name but with a different
 * {@link Configuration} may be configured.
 * <p/>
 * This is equivalent to the following sequence of method calls:
 * <ol>
 *   <li>{@link javax.cache.Cache#clear()}</li>
 *   <li>{@link javax.cache.Cache#close()}</li>
 * </ol>
 * followed by allowing the name of the {@link Cache} to be used for other
 * {@link Cache} configurations.
 * <p/>
 * From the time this method is called, the specified {@link Cache} is not
 * available for operational use. An attempt to call an operational method on
 * the {@link Cache} will throw an {@link IllegalStateException}.
 *
 * @param cacheName the cache name
 * @throws IllegalStateException if the {@link Cache} is {@link #isClosed()}
 * @throws NullPointerException  if cacheName is null
 */
void destroyCache(String cacheName);
```

Once destroyed:

- any attempt to use an operational method on instances of the Cache will throw an `IllegalStateException`.

- the destroyed `Cache`'s name may be reused in a new cache by calling the `CacheManager.create` method, with the same or a different configuration.

Once destroyed a `Cache` is no longer available via a `CacheManager`. Destroying a `Cache` ensures that it is closed and all of the associated entries are no longer available by any application, both immediately and in the future, regardless of implementation or topology.

4.1.8. Closing CacheManagers

Closing a `CacheManager` via a call to the `CacheManager.close()` method or via the `CachingProvider.close(...)` methods has the effect of instructing a CacheManager to:

- close all of the Caches that it is currently managing, and
- release all resources that are currently being used to manage the Caches.
- Once closed any attempt to use an operational method on a closed CacheManager or any of the Caches it was managing will throw an `IllegalStateException`. The operational methods on CacheManager are:
    - `createCache`
    - `destroyCache`
    - `enableManagement`
    - `enableStatistics`
    - `getCache`
    - `getCacheNames`

After closing a CacheManager, another instance, possibly representing the previously managed Caches, may be acquired using the CachingProvider that originally produced the CacheManager. This is covered in the section on CachingProviders.

4.1.9. ClassLoading

All Caches share the same ClassLoader that was configured for the CacheManager from which they were acquired when the CacheManager was created.

To configure and acquire Caches that use different ClassLoaders, individual CacheManagers must be established to do so. For information on how to configure CacheManagers, consult the section on CachingProviders.

5. Caches

The primary artifact developers use to interact with a Cache is the `javax.cache.Cache` interface.

The `javax.cache.Cache` interface provides Map-like methods to enable access, update and remove access to Cache Entries.

The `javax.cache.Cache` interface is defined as follows:

```
/**
 * A {@link Cache} is a Map-like data structure that provides temporary storage
 * of application data.
 * <p>
 * Like {@link Map}s, {@link Cache}s
 * <ol>
 * <li>store key-value pairs, each referred to as an {@link Entry}</li>
 * <li>allow use of Java Generics to improve application type-safety</li>
 * <li>are {@link Iterable}</li>
 * </ol>
 * <p>
 * Unlike {@link Map}s, {@link Cache}s
 * <ol>
 * <li>do not allow null keys or values.  Attempts to use <code>null</code>
 * will result in a {@link NullPointerException}</li>
 * <li>provide the ability to read values from a
 * {@link CacheLoader} (read-through-caching)
 * when a value being requested is not in a cache</li>
 * <li>provide the ability to write values to a
 * {@link CacheWriter} (write-through-caching)
 * when a value being created/updated/removed from a cache</li>
 * <li>provide the ability to observe cache entry changes</li>
 * <li>may capture and measure operational statistics</li>
 * </ol>
 * <p>
 * A simple example of how to use a cache is:
 * <pre><code>
 * String cacheName = "sampleCache";
 * CachingProvider provider = Caching.getCachingProvider();
 * CacheManager manager = provider.getCacheManager();
 * Cache<Integer, Date> cache = manager.getCache(cacheName, Integer.class,
 *                                               Date.class);
 * Date value1 = new Date();
 * Integer key = 1;
 * cache.put(key, value1);
 * Date value2 = cache.get(key);
 * </code></pre>
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Greg Luck
 * @author Yannis Cosmadopoulos
 * @author Brian Oliver
 * @since 1.0
 */
public interface Cache<K, V> extends Iterable<Cache.Entry<K, V>>, Closeable {
  /**
```

```
 * Gets an entry from the cache.
 * <p>
 * If the cache is configured to use read-through, and get would return null
 * because the entry is missing from the cache, the Cache's {@link CacheLoader}
 * is called in an attempt to load the entry.
 *
 * @param key the key whose associated value is to be returned
 * @return the element, or null, if it does not exist.
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws NullPointerException  if the key is null
 * @throws CacheException        if there is a problem fetching the value
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 */
V get(K key);

/**
 * Gets a collection of entries from the {@link Cache}, returning them as
 * {@link Map} of the values associated with the set of keys requested.
 * <p>
 * If the cache is configured read-through, and a get for a key would
 * return null because an entry is missing from the cache, the Cache's
 * {@link CacheLoader} is called in an attempt to load the entry. If an
 * entry cannot be loaded for a given key, the key will not be present in
 * the returned Map.
 *
 * @param keys The keys whose associated values are to be returned.
 * @return A map of entries that were found for the given keys. Keys not found
 *         in the cache are not in the returned map.
 * @throws NullPointerException  if keys is null or if keys contains a null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem fetching the values
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 */
Map<K, V> getAll(Set<? extends K> keys);

/**
 * Determines if the {@link Cache} contains an entry for the specified key.
 * <p>
 * More formally, returns <tt>true</tt> if and only if this cache contains a
 * mapping for a key <tt>k</tt> such that <tt>key.equals(k)</tt>.
 * (There can be at most one such mapping.)
 *
 * @param key key whose presence in this cache is to be tested.
 * @return <tt>true</tt> if this map contains a mapping for the specified key
 * @throws NullPointerException  if key is null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        it there is a problem checking the mapping
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see java.util.Map#containsKey(Object)
 */
```

```
boolean containsKey(K key);

/**
 * Asynchronously loads the specified entries into the cache using the
 * configured {@link CacheLoader} for the given keys.
 * <p>
 * If an entry for a key already exists in the Cache, a value will be loaded
 * if and only if <code>replaceExistingValues</code> is true.   If no loader
 * is configured for the cache, no objects will be loaded.  If a problem is
 * encountered during the retrieving or loading of the objects,
 * an exception is provided to the {@link CompletionListener}.  Once the
 * operation has completed, the specified CompletionListener is notified.
 * <p>
 * Implementations may choose to load multiple keys from the provided
 * {@link Set} in parallel.  Iteration however must not occur in parallel,
 * thus allow for non-thread-safe {@link Set}s to be used.
 * <p>
 * The thread on which the completion listener is called is implementation
 * dependent. An implementation may also choose to serialize calls to
 * different CompletionListeners rather than use a thread per
 * CompletionListener.
 *
 * @param keys                  the keys to load
 * @param replaceExistingValues when true existing values in the Cache will
 *                              be replaced by those loaded from a CacheLoader
 * @param completionListener    the CompletionListener (may be null)
 * @throws NullPointerException  if keys is null or if keys contains a null.
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        thrown if there is a problem performing the
 *                               load. This may also be thrown on calling if
 *                               there are insufficient threads available to
 *                               perform the load.
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 */
void loadAll(Set<? extends K> keys, boolean replaceExistingValues,
             CompletionListener completionListener);

/**
 * Associates the specified value with the specified key in the cache.
 * <p>
 * If the {@link Cache} previously contained a mapping for the key, the old
 * value is replaced by the specified value.  (A cache <tt>c</tt> is said to
 * contain a mapping for a key <tt>k</tt> if and only if {@link
 * #containsKey(Object) c.containsKey(k)} would return <tt>true</tt>.)
 *
 * @param key   key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @throws NullPointerException  if key is null or if value is null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem doing the put
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see java.util.Map#put(Object, Object)
 * @see #getAndPut(Object, Object)
```

```
     * @see #getAndReplace(Object, Object)
     * @see CacheWriter#write
     */
    void put(K key, V value);

    /**
     * Associates the specified value with the specified key in this cache,
     * returning an existing value if one existed.
     * <p>
     * If the cache previously contained a mapping for
     * the key, the old value is replaced by the specified value.  (A cache
     * <tt>c</tt> is said to contain a mapping for a key <tt>k</tt> if and only
     * if {@link #containsKey(Object) c.containsKey(k)} would return
     * <tt>true</tt>.)
     * <p>
     * The previous value is returned, or null if there was no value associated
     * with the key previously.
     *
     * @param key   key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the value associated with the key at the start of the operation or
     *         null if none was associated
     * @throws NullPointerException  if key is null or if value is null
     * @throws IllegalStateException if the cache is {@link #isClosed()}
     * @throws CacheException        if there is a problem doing the put
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @see #put(Object, Object)
     * @see #getAndReplace(Object, Object)
     * @see CacheWriter#write
     */
    V getAndPut(K key, V value);

    /**
     * Copies all of the entries from the specified map to the {@link Cache}.
     * <p>
     * The effect of this call is equivalent to that of calling
     * {@link #put(Object, Object) put(k, v)} on this cache once for each mapping
     * from key <tt>k</tt> to value <tt>v</tt> in the specified map.
     * <p>
     * The order in which the individual puts occur is undefined.
     * <p>
     * The behavior of this operation is undefined if entries in the cache
     * corresponding to entries in the map are modified or removed while this
     * operation is in progress. or if map is modified while the operation is in
     * progress.
     * <p>
     * In Default Consistency mode, individual puts occur atomically but not
     * the entire putAll.  Listeners may observe individual updates.
     *
     * @param map mappings to be stored in this cache
     * @throws NullPointerException  if map is null or if map contains null keys
     *                               or values.
     * @throws IllegalStateException if the cache is {@link #isClosed()}
     * @throws CacheException        if there is a problem doing the put.
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
```

```
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @see CacheWriter#writeAll
     */
    void putAll(java.util.Map<? extends K, ? extends V> map);

    /**
     * Atomically associates the specified key with the given value if it is
     * not already associated with a value.
     * <p>
     * This is equivalent to:
     * <pre><code>
     * if (!cache.containsKey(key)) {}
     *   cache.put(key, value);
     *   return true;
     * } else {
     *   return false;
     * }
     * </code></pre>
     * except that the action is performed atomically.
     *
     * @param key   key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return true if a value was set.
     * @throws NullPointerException  if key is null or value is null
     * @throws IllegalStateException if the cache is {@link #isClosed()}
     * @throws CacheException        if there is a problem doing the put
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @see CacheWriter#write
     */
    boolean putIfAbsent(K key, V value);

    /**
     * Removes the mapping for a key from this cache if it is present.
     * <p>
     * More formally, if this cache contains a mapping from key <tt>k</tt> to
     * value <tt>v</tt> such that
     * <code>(key==null ?  k==null : key.equals(k))</code>, that mapping is removed.
     * (The cache can contain at most one such mapping.)
     *
     * <p>Returns <tt>true</tt> if this cache previously associated the key,
     * or <tt>false</tt> if the cache contained no mapping for the key.
     * <p>
     * The cache will not contain a mapping for the specified key once the
     * call returns.
     *
     * @param key key whose mapping is to be removed from the cache
     * @return returns false if there was no matching key
     * @throws NullPointerException  if key is null
     * @throws IllegalStateException if the cache is {@link #isClosed()}
     * @throws CacheException        if there is a problem doing the remove
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @see CacheWriter#delete
```

```
 */
boolean remove(K key);

/**
 * Atomically removes the mapping for a key only if currently mapped to the
 * given value.
 * <p>
 * This is equivalent to:
 * <pre><code>
 * if (cache.containsKey(key) &amp;&amp; equals(cache.get(key), oldValue) {
 *   cache.remove(key);
 *   return true;
 * } else {
 *   return false;
 * }
 * </code></pre>
 * except that the action is performed atomically.
 *
 * @param key      key whose mapping is to be removed from the cache
 * @param oldValue value expected to be associated with the specified key
 * @return returns false if there was no matching key
 * @throws NullPointerException  if key is null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem doing the remove
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see CacheWriter#delete
 */
boolean remove(K key, V oldValue);

/**
 * Atomically removes the entry for a key only if currently mapped to some
 * value.
 * <p>
 * This is equivalent to:
 * <pre><code>
 * if (cache.containsKey(key)) {
 *   V oldValue = cache.get(key);
 *   cache.remove(key);
 *   return oldValue;
 * } else {
 *   return null;
 * }
 * </code></pre>
 * except that the action is performed atomically.
 *
 * @param key key with which the specified value is associated
 * @return the value if one existed or null if no mapping existed for this key
 * @throws NullPointerException  if the specified key or value is null.
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the remove
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see CacheWriter#delete
 */
```

```
V getAndRemove(K key);

/**
 * Atomically replaces the entry for a key only if currently mapped to a
 * given value.
 * <p>
 * This is equivalent to:
 * <pre><code>
 * if (cache.containsKey(key) &amp;&amp; equals(cache.get(key), oldValue)) {
 *  cache.put(key, newValue);
 * return true;
 * } else {
 *  return false;
 * }
 * </code></pre>
 * except that the action is performed atomically.
 *
 * @param key      key with which the specified value is associated
 * @param oldValue value expected to be associated with the specified key
 * @param newValue value to be associated with the specified key
 * @return <tt>true</tt> if the value was replaced
 * @throws NullPointerException  if key is null or if the values are null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the replace
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see CacheWriter#write
 */
boolean replace(K key, V oldValue, V newValue);

/**
 * Atomically replaces the entry for a key only if currently mapped to some
 * value.
 * <p>
 * This is equivalent to
 * <pre><code>
 * if (cache.containsKey(key)) {
 *   cache.put(key, value);
 *   return true;
 * } else {
 *   return false;
 * }</code></pre>
 * except that the action is performed atomically.
 *
 * @param key  the key with which the specified value is associated
 * @param value the value to be associated with the specified key
 * @return <tt>true</tt> if the value was replaced
 * @throws NullPointerException  if key is null or if value is null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the replace
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see #getAndReplace(Object, Object)
 * @see CacheWriter#write
 */
```

```
boolean replace(K key, V value);

/**
 * Atomically replaces the value for a given key if and only if there is a
 * value currently mapped by the key.
 * <p>
 * This is equivalent to
 * <pre><code>
 * if (cache.containsKey(key)) {
 *   V oldValue = cache.get(key);
 *   cache.put(key, value);
 *   return oldValue;
 * } else {
 *   return null;
 * }
 * </code></pre>
 * except that the action is performed atomically.
 *
 * @param key   key with which the specified value is associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with the specified key, or
 *         <tt>null</tt> if there was no mapping for the key.
 * @throws NullPointerException  if key is null or if value is null
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the replace
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see java.util.concurrent.ConcurrentMap#replace(Object, Object)
 * @see CacheWriter#write
 */
V getAndReplace(K key, V value);

/**
 * Removes entries for the specified keys.
 * <p>
 * The order in which the individual entries are removed is undefined.
 * <p>
 * For every entry in the key set, the following are called:
 * <ul>
 *   <li>any registered {@link CacheEntryRemovedListener}s</li>
 *   <li>if the cache is a write-through cache, the {@link CacheWriter}</li>
 * </ul>
 * If the key set is empty, the {@link CacheWriter} is not called.
 *
 * @param keys the keys to remove
 * @throws NullPointerException  if keys is null or if it contains a null key
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the remove
 * @throws ClassCastException    if the implementation is configured to perform
 *                               runtime-type-checking, and the key or value
 *                               types are incompatible with those that have been
 *                               configured for the {@link Cache}
 * @see CacheWriter#deleteAll
 */
void removeAll(Set<? extends K> keys);

/**
```

```
 * Removes all of the mappings from this cache.
 * <p>
 * The order that the individual entries are removed is undefined.
 * <p>
 * For every mapping that exists the following are called:
 * <ul>
 *   <li>any registered {@link CacheEntryRemovedListener}s</li>
 *   <li>if the cache is a write-through cache, the {@link CacheWriter}</li>
 * </ul>
 * If the cache is empty, the {@link CacheWriter} is not called.
 * <p>
 * This is potentially an expensive operation as listeners are invoked.
 * Use {@link #clear()} to avoid this.
 *
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the remove
 * @see #clear()
 * @see CacheWriter#deleteAll
 */
void removeAll();

/**
 * Clears the contents of the cache, without notifying listeners or
 * {@link CacheWriter}s.
 *
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 * @throws CacheException        if there is a problem during the clear
 */
void clear();

/**
 * Provides a standard way to access the configuration of a cache using
 * JCache configuration or additional proprietary configuration.
 * <p>
 * The returned value must be immutable.
 * <p>
 * If the provider's implementation does not support the specified class,
 * the {@link IllegalArgumentException} is thrown.
 *
 * @param clazz the configuration interface or class to return. This includes
 *              {@link Configuration}.class and
 *              {@link javax.cache.configuration.CompleteConfiguration}s.
 * @return the requested implementation of {@link Configuration}
 * @throws IllegalArgumentException if the caching provider doesn't support
 *                                  the specified class.
 */
<C extends Configuration<K, V>> C getConfiguration(Class<C> clazz);

/**
 * Invokes an {@link EntryProcessor} against the {@link Entry} specified by
 * the provided key. If an {@link Entry} does not exist for the specified key,
 * an attempt is made to load it (if a loader is configured) or a surrogate
 * {@link Entry}, consisting of the key with a null value is used instead.
 * <p>
 *
 * @param key            the key to the entry
 * @param entryProcessor the {@link EntryProcessor} to invoke
 * @param arguments      additional arguments to pass to the
 *                       {@link EntryProcessor}
```

```
     * @return the result of the processing, if any, defined by the
     *         {@link EntryProcessor} implementation
     * @throws NullPointerException    if key or {@link EntryProcessor} is null
     * @throws IllegalStateException   if the cache is {@link #isClosed()}
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @throws EntryProcessorException if an exception is thrown by the {@link
     *                                 EntryProcessor}, a Caching Implementation
     *                                 must wrap any {@link Exception} thrown
     *                                 wrapped in an {@link EntryProcessorException}.
     * @see EntryProcessor
     */
    <T> T invoke(K key,
                 EntryProcessor<K, V, T> entryProcessor,
                 Object... arguments) throws EntryProcessorException;

    /**
     * Invokes an {@link EntryProcessor} against the set of {@link Entry}s
     * specified by the set of keys.
     * <p>
     * If an {@link Entry} does not exist for the specified key, an attempt is made
     * to load it (if a loader is configured) or a surrogate {@link Entry},
     * consisting of the key and a value of null is provided.
     * <p>
     * The order that the entries for the keys are processed is undefined.
     * Implementations may choose to process the entries in any order, including
     * concurrently.  Furthermore there is no guarantee implementations will
     * use the same {@link EntryProcessor} instance to process each entry, as
     * the case may be in a non-local cache topology.
     * <p>
     * The result of executing the {@link EntryProcessor} is returned as a
     * {@link Map} of {@link EntryProcessorResult}s, one result per key.  Should the
     * {@link EntryProcessor} or Caching implementation throw an exception, the
     * exception is wrapped and re-thrown when a call to
     * {@link javax.cache.processor.EntryProcessorResult#get()} is made.
     *
     * @param keys           the set of keys for entries to process
     * @param entryProcessor the {@link EntryProcessor} to invoke
     * @param arguments      additional arguments to pass to the
     *                       {@link EntryProcessor}
     * @return the map of {@link EntryProcessorResult}s of the processing per key,
     * if any, defined by the {@link EntryProcessor} implementation.  No mappings
     * will be returned for {@link EntryProcessor}s that return a
     * <code>null</code> value for a key.
     * @throws NullPointerException    if keys or {@link EntryProcessor} are null
     * @throws IllegalStateException   if the cache is {@link #isClosed()}
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @see EntryProcessor
     */
    <T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys,
                                                  EntryProcessor<K, V, T>
                                                    entryProcessor,
                                                  Object... arguments);
```

```
/**
 * Return the name of the cache.
 *
 * @return the name of the cache.
 */
String getName();

/**
 * Gets the {@link CacheManager} that owns and manages the {@link Cache}.
 *
 * @return the manager or <code>null</code> if the {@link Cache} is not
 *         managed
 */
CacheManager getCacheManager();

/**
 * Closing a {@link Cache} signals to the {@link CacheManager} that produced or
 * owns the {@link Cache} that it should no longer be managed. At this
 * point in time the {@link CacheManager}:
 * <ul>
 * <li>must close and release all resources being coordinated on behalf of the
 * Cache by the {@link CacheManager}. This includes calling the <code>close
 * </code> method on configured {@link CacheLoader},
 * {@link CacheWriter}, registered {@link CacheEntryListener}s and
 * {@link ExpiryPolicy} instances that implement the java.io.Closeable
 * interface.
 * <li>prevent events being delivered to configured {@link CacheEntryListener}s
 * registered on the {@link Cache}
 * </li>
 * <li>not return the name of the Cache when the CacheManager getCacheNames()
 * method is called</li>
 * </ul>
 * Once closed any attempt to use an operational method on a Cache will throw an
 * {@link IllegalStateException}.
 *
 * @throws SecurityException when the operation could not be performed
 *                           due to the current security settings
 */
void close();

/**
 * Determines whether this Cache instance has been closed. A Cache is
 * considered closed if;
 * <ol>
 * <li>the {@link #close()} method has been called</li>
 * <li>the associated {@link #getCacheManager()} has been closed, or</li>
 * <li>the Cache has been removed from the associated
 * {@link #getCacheManager()}</li>
 * </ol>
 * <p>
 * This method generally cannot be called to determine whether a Cache instance
 * is valid or invalid. A typical client can determine that a Cache is invalid
 * by catching any exceptions that might be thrown when an operation is
 * attempted.
 *
 * @return true if this Cache instance is closed; false if it is still open
 */
boolean isClosed();
```

```
/**
 * Provides a standard way to access the underlying concrete caching
 * implementation to provide access to further, proprietary features.
 * <p>
 * If the provider's implementation does not support the specified class,
 * the {@link IllegalArgumentException} is thrown.
 *
 * @param clazz the proprietary class or interface of the underlying concrete
 *              cache. It is this type that is returned.
 * @return an instance of the underlying concrete cache
 * @throws IllegalArgumentException if the caching provider doesn't support
 *                                  the specified class.
 * @throws SecurityException        when the operation could not be performed
 *                                  due to the current security settings
 */
<T> T unwrap(java.lang.Class<T> clazz);

/**
 * Registers a {@link CacheEntryListener}. The supplied
 * {@link CacheEntryListenerConfiguration} is used to instantiate a listener
 * and apply it to those events specified in the configuration.
 *
 * @param cacheEntryListenerConfiguration
 *         a factory and related configuration
 *         for creating the listener
 * @throws IllegalArgumentException is the same CacheEntryListenerConfiguration
 *                                  is used more than once
 * @throws IllegalStateException    if the cache is {@link #isClosed()}
 * @see CacheEntryListener
 */
void registerCacheEntryListener(
    CacheEntryListenerConfiguration<K, V> cacheEntryListenerConfiguration);

/**
 * Deregisters a listener, using the
 * {@link CacheEntryListenerConfiguration} that was used to register it.
 * <p>
 * Both listeners registered at configuration time,
 * and those created at runtime with {@link #registerCacheEntryListener} can
 * be deregistered.
 *
 * @param cacheEntryListenerConfiguration
 *         the factory and related configuration
 *         that was used to create the
 *         listener
 * @throws IllegalStateException if the cache is {@link #isClosed()}
 */
void deregisterCacheEntryListener(CacheEntryListenerConfiguration<K, V>
                                      cacheEntryListenerConfiguration);

/**
 * {@inheritDoc}
 * <p>
 * The ordering of iteration over entries is undefined.
 * <p>
 * During iteration, any entries that are a). read will have their appropriate
 * CacheEntryReadListeners notified and b). removed will have their appropriate
 * CacheEntryRemoveListeners notified.
 * <p>
```

```
     * {@link java.util.Iterator#next()} may return null if the entry is no
     * longer present, has expired or has been evicted.
     *
     * @throws IllegalStateException if the cache is {@link #isClosed()}
     */
    Iterator<Cache.Entry<K, V>> iterator();

    /**
     * A cache entry (key-value pair).
     */
    interface Entry<K, V> {

      /**
       * Returns the key corresponding to this entry.
       *
       * @return the key corresponding to this entry
       */
      K getKey();

      /**
       * Returns the value stored in the cache when this entry was created.
       *
       * @return the value corresponding to this entry
       */
      V getValue();

      /**
       * Provides a standard way to access the underlying concrete cache entry
       * implementation in order to provide access to further, proprietary features.
       * <p>
       * If the provider's implementation does not support the specified class,
       * the {@link IllegalArgumentException} is thrown.
       *
       * @param clazz the proprietary class or interface of the underlying
       *              concrete cache. It is this type that is returned.
       * @return an instance of the underlying concrete cache
       * @throws IllegalArgumentException if the caching provider doesn't support
       *                                  the specified class.
       */
      <T> T unwrap(Class<T> clazz);
    }
}
```

5.1. Cache Type-Safety

The Java Caching API makes extensive use of Java Generics, as defined by JSR-14, to enable the development of compile-time type-safe applications when adopting Caching.

While available, compile-time type-safety does not guarantee runtime-type correctness for applications adopting caching. For some cache topologies, specifically those that store or communicate entries across Java process boundaries, Java runtime-type-information erasure and the inability to acquire and transfer generic type information may mean application types are unable to ensure type-safety of Cache operations in such environments. Care should always be taken to ensure Caches are configured using the appropriate key and value types so that implementations may perform type checking as necessary or required.

5.2. Compile-time Type-Safety

Compile-time type-safety is provided by declaring a `Cache` with the required generic types.

Example 1

In the following example a `Cache` is declared to have a key of type `String` and a value of type `Integer`. Compile time errors will be generated when incompatible values are specified when interacting with this cache.

```
Configuration config = new MutableConfiguration();

//create the cache
cacheManager.createCache(cacheName, config);

//... then later to get the cache
Cache<String, Integer> cache = cacheManager.getCache(cacheName);

//use the cache
String key = "key";
Integer value1 = 1;
cache.put("key", value1);
Integer value2 = cache.get(key);

//the following will not compile - incorrect types specified
//cache.put(2, "some value);
```

While it is possible to circumvent compile-time type-safety checking by declaring a `Cache` using raw types (not specifying generic type parameters), it is not a recommended practise as it permits simple programming errors to occur.

Example 2

In the following example a Cache is declared as a raw type. In this situation no compile-time type-checking can performed (although type warnings may be generated).

```
Configuration config = new MutableConfiguration();
cacheManager.createCache(cacheName, config);

//... then later to get the cache without type information
Cache cache = cacheManager.getCache(cacheName);
String key = "key";
Integer value1 = 1;
cache.put("key", value1);

cache.put(value1, "key1");  //not intended but will still compile and execute!
Integer value2 = (Integer) cache.get(key);
assertEquals(value1, value2);
```

5.3. Runtime Type-Safety

In addition to compile-type type-safety, developers may enable runtime type-safety through configuring a `Cache` with specific key and value types. For example, the `MutableConfiguration` class provides the following method to define the required key and value types for a `Cache`.

```
/**
 * Sets the expected type of keys and values for a {@link Cache}
 * configured with this {@link Configuration}. Setting both to
 * <code>Object.class</code> means type-safety checks are not required.
 * <p/>
 * This is used by {@link CacheManager} to ensure that the key and value
 * types are the same as those configured for the {@link Cache} prior to
 * returning a requested cache from this method.
 * <p/>
 * Implementations may further perform type checking on mutative cache operations
 * and throw a {@link ClassCastException} if these checks fail.
 *
 * @param keyType   the expected key type
 * @param valueType the expected value type
 * @return the {@link MutableConfiguration} to permit fluent-style method calls
 * @throws NullPointerException should the key or value type be null
 * @see CacheManager#getCache(String, Class, Class)
 */
public MutableConfiguration<K, V> setTypes(Class<K> keyType, Class<V> valueType)
```

When a `Configuration` defines key and value types, a `Cache` returned by `CacheManager.getCache` must enforce that the requested key and value types are the same as those configured. To request a `Cache` with specific key and value types, the following `CacheManager` method must be used.

```
<K, V> Cache<K, V> getCache(String cacheName,
                            Class<K> keyType,
                            Class<V> valueType);
```

When using the above method, implementations must ensure that the returned Caches have been explicitly configured with types as specified when calling the method. This provides developers with an increased level of safety beyond that of simply using Generics.

Implementations may also perform key and value type checking at runtime for mutative Cache operations.

When a `Configuration` does not define required key and value types, or they are both defined as `Object.class`, an implementation is not required to perform runtime type-checking when requesting a `Cache`. To request a `Cache` without checking use:

```
<K, V> Cache<K, V> Cache getCache(String cacheName);
```

Attempting to use getCache without providing type parameters when a Cache has been configured with specific types or using getCache with specific type parameters when a Cache has been configured without specific types cause an `IllegalArgumentException` to be thrown.

Example 2

In this example a cache is configured to have a `String` key type and an `Integer` value type. The implementation then ensures that the declared types match the configured cache or an `IllegalArgumentException` is thrown.

```
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();

MutableConfiguration<String, Integer> config = new
    MutableConfiguration<String, Integer>();
config.setTypes(String.class, Integer.class);
cacheManager.createCache("simpleCache", config);

//... then later to get the cache without type information
Cache<String, Integer> simpleCache = cacheManager.getCache("simpleCache",
    String.class, Integer.class);

simpleCache.put("key1", 3);
Integer value2 = simpleCache.get("key1");
```

While the Java Caching API provides mechanisms for both compile and runtime-type safety, type checking is only applicable to reifiable types of keys and values, including all generic collection types. For example a value of type `List<MyClass>` is not reifiable at runtime and thus may only be compared with the type `List.class`.

6. Expiry Policies

If an entry is expired, it is not returned from a cache. If no expiry policy has been configured for a cache, it defaults to the `Eternal` expiry policy, where cache entries do not expire.

While the Eternal Expiry Policy does not allow entries to expire, implementations may evict entries when required.

Expiry policies may be set at configuration time by providing an `ExpiryPolicy` implementation, See below for the definition.

```
/**
 * Defines functions to determine when cache entries will expire based on
 * creation, access and modification operations.
 * <p>
 * Each of the functions return a new {@link Duration} that specifies the
 * amount of time that must pass before a cache entry is considered expired.
 * {@link Duration} has constants defined for useful durations.
 *
 * @author Brian Oliver
 * @author Greg Luck
 * @since 1.0
 * @see Duration
 */
public interface ExpiryPolicy {

  /**
   * Gets the {@link Duration} before a newly created Cache.Entry is considered
   * expired.
   * <p>
   * This method is called by a caching implementation after a Cache.Entry is
   * created, but before a Cache.Entry is added to a cache, to determine the
   * {@link Duration} before an entry expires.  If a {@link Duration#ZERO}
   * is returned the new Cache.Entry is considered to be already expired and
   * will not be added to the Cache.
   * <p>
   * Should an exception occur while determining the Duration, an implementation
   * specific default {@link Duration} will be used.
   *
   * @return the new {@link Duration} before a created entry expires
   */
  Duration getExpiryForCreation();

  /**
   * Gets the {@link Duration} before an accessed Cache.Entry is
   * considered expired.
   * <p>
   * This method is called by a caching implementation after a Cache.Entry is
   * accessed to determine the {@link Duration} before an entry expires.  If a
   * {@link Duration#ZERO} is returned a Cache.Entry will be
```

```
   * considered immediately expired.  Returning <code>null</code> will result
   * in no change to the previously understood expiry {@link Duration}.
   * <p>
   * Should an exception occur while determining the Duration, an implementation
   * specific default Duration will be used.
   *
   * @return the new {@link Duration} before an accessed entry expires
   */
  Duration  getExpiryForAccess();

  /**
   * Gets the {@link Duration} before an updated Cache.Entry is considered
   * expired.
   * <p>
   * This method is called by the caching implementation after a Cache.Entry is
   * updated to determine the {@link Duration} before the updated entry expires.
   * If a {@link Duration#ZERO} is returned a Cache.Entry is considered
   * immediately expired.  Returning <code>null</code> will result in no change
   * to the previously understood expiry {@link Duration}.
   * <p>
   * Should an exception occur while determining the Duration, an implementation
   * specific default Duration will be used.
   *
   * @return the new {@link Duration} before an updated entry expires
   */
  Duration getExpiryForUpdate();
}
```

Cache entries are expired a set time after certain cache operations are performed, the time defined using the `Duration` class. Duration is a pair made up of a `java.util.concurrent.TimeUnit` and a `long durationAmount`. The minimum allowed `TimeUnit` is `TimeUnit.MILLISECONDS`.

The expiry duration for a Cache entry depends on the configured expiry policy and the cache operation performed. The following `ExpiryPolicy` methods are defined to determine suitable durations based on cache operations:

- `getExpiryForCreation()` - the duration for an entry when it is created
- `getExpiryForAccess()` - the new duration for an entry when it is accessed
- `getExpiryForUpdate()` - the new duration for an entry when it is updated

When a Cache implementation calls these methods the `ExpiryPolicy` will return one of the following:

- A duration defining the required entry expiry duration
- `Duration.ZERO` indicating an entry is now considered expired

In addition `getExpiryForUpdate()` and `getExpiryForAccess()` may also return null, indicating the caching implementation should leave the expiry duration of an entry unchanged for these operations.

In addition to the `Duration#ZERO` constant, constants are also defined for 1 day, 1 hour, 30 minutes, 20 minutes, 10 minutes, 5 minutes and 1 minute.

The following table details how each of the cache methods interact with a configured `ExpiryPolicy`.

Method	`ExpiryPolicy.getExpiryForCreation` called?	`ExpiryPolicy.getExpiryFor Access` called?	`ExpiryPolicy.getExpiryFor Update` called?
`boolean containsKey(K key)`	No	No	No
`V get(K key)`	No (unless read-though caused a load)	Yes	No
`Map<K,V> getAll(Collection<? extends K> keys)`	No (unless read-though caused a load)	Yes	No
`V getAndPut(K key, V value)`	Yes (when the key is not associated with an existing value)	No	Yes (when the key is associated with an existing value)
`V getAndRemove(K key)`	No	No	No
`V getAndReplace(K key, V value)`	No	No	Yes (when the key is associated with an existing value)
`CacheManager getCacheManager()`	No	No	No
`CacheConfiguration getConfiguration()`	No	No	No
`String getName()`	No	No	No

`Iterator<Cache.Entry<K, V>> iterator()`	No	Yes (when an entry is visited by an iterator)	No
`void loadAll(Set<? extends K> keys, boolean replaceExistingValues, CompletionListener listener)`	Yes (when a key is not associated with a loaded value)	No	Yes (when a key is associated with a loaded value and the value should be replaced)
`void put(K key, V value)`	Yes (when the key is not associated with an existing value)	No	Yes (when the key is associated with an existing value)
`void putAll(Map<? extends K,? extends V> map)`	Yes (when the key is not associated with an existing value)	No	Yes (when the key is associated with an existing value)
`boolean putIfAbsent(K key, V value)`	Yes (when the key is not associated with an existing value)	No	No
`boolean remove(K key)`	No	No	No
`boolean remove(K key, V oldValue)`	No	Yes (when the old value does not match the existing value)	No
`void removeAll()`	No	No	No
`void removeAll(Set<? extends K> keys)`	No	No	No

`<T> T invoke(K key, EntryProcessor<K, V, T> entryProcessor, Object... arguments)entryProcessor);`	Yes (for the following cases: (1) setValue called and entry did not exist for key before invoke was called. (2) if read-through enabled and getValue() is called and causes a new entry to be loaded for key)	Yes (when getValue was called and no other mutations occurred during entry processor execution. note: Create, modify or remove take precedence over Access)	Yes (when setValue was called and the entry already existed before entry processor was called)
`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	Yes (for the following cases: (1) setValue called and an entry did not exist for key before invoke was called. (2) if read-through enabled and getValue() is called and causes a new entry to be loaded for key)	Yes (when getValue was called and no other mutations occurred during entry processor execution. note: Create, modify or remove take precedence over Access)	Yes (when setValue was called and an entry already existed before entry processor was called)
`boolean replace(K key, V value)`	No	No	Yes (when the key is associated with an

			existing value)
`boolean replace(K key, V oldValue, V newValue)`	No	Yes (when value is not replaced)	Yes (when value is replaced)
`<T> T unwrap(Class<T> cls)`	No	No	No

Five expiry policy class implementations are defined and included with the specification in the package `javax.cache.expiry`:

1. `CreatedExpiryPolicy` - expire a set time after creation.
2. `ModifiedExpiryPolicy` - expire a set time after creation. Refresh expiry when an entry is updated.
3. `AccessedExpiryPolicy` - expire a set time after creation. Refresh expiry when an entry is accessed (a read operation of some kind)
4. `TouchedExpiryPolicy` - expire a set time after creation. Refresh expiry when an entry is updated or accessed
5. `EternalExpiryPolicy` - never expire. This is the default.

7. Integration

Convenience methods have been created to ease integration with external resources. These are in the `javax.cache.integration` package.

Two interfaces CacheLoader and CacheWriter which are defined as follows:

```
/**
 * Used when a cache is read-through or when loading data into a cache via the
 * {@link javax.cache.Cache#loadAll(java.util.Set, boolean,
 * CompletionListener)} method.
 * <p/>
 *
 * @param <K> the type of keys handled by this loader
 * @param <V> the type of values generated by this loader
 * @author Greg Luck
 * @author Yannis Cosmadopoulos
 * @see Configuration#isReadThrough()
 * @see CacheWriter
 * @since 1.0
 */
public interface CacheLoader<K, V> {

  /**
   * Loads an object. Application developers should implement this
   * method to customize the loading of a value for a cache entry. This method
   * is called by a cache when a requested entry is not in the cache. If
   * the object can't be loaded <code>null</code> should be returned.
   *
   * @param key the key identifying the object being loaded
   * @return The value for the entry that is to be stored in the cache or
   *         <code>null</code> if the object can't be loaded
   * @throws CacheLoaderException if there is problem executing the loader.
   */
  V load(K key) throws CacheLoaderException;



  /**
   * Loads multiple objects. Application developers should implement this
   * method to customize the loading of cache entries. This method is called
   * when the requested object is not in the cache. If an object can't be loaded,
   * it is not returned in the resulting map.
   *
   * @param keys keys identifying the values to be loaded
   * @return A map of key, values to be stored in the cache.
   * @throws CacheLoaderException if there is problem executing the loader.
   */
```

```
  Map<K, V> loadAll(Iterable<? extends K> keys) throws CacheLoaderException;
}
```

```
/**
 * A CacheWriter is used for write-through to an external resource.
 * <p/>
 * Under Default Consistency, the non-batch writer methods are atomic with respect
 * to the corresponding cache operation.
 * <p/>
 * For batch methods under Default Consistency, the entire cache operation
 * is not required to be atomic in {@link Cache} and is therefore not required to
 * be atomic in the writer. As individual writer operations can fail, cache
 * operations are not required to occur until after the writer batch method has
 * returned or, in the case of partial success, thrown an exception. In the case
 * of
 * partial success, the collection of entries must contain only those entries
 * which failed.
 * <p/>
 * The entry passed into {@link #write(Cache.Entry)} is independent
 * of the cache mapping for that key, meaning that if the value changes in the
 * cache or is removed it does not change the entry.
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 * @author Greg Luck
 * @author Brian Oliver
 * @see CacheLoader
 * @since 1.0
 */
public interface CacheWriter<K, V> {


  /**
   * Write the specified value under the specified key to the external resource.
   * <p/>
   * This method is intended to support both key/value creation and value update
   * for a specific key.
   *
   * @param entry the entry to be written
   * @throws CacheWriterException if the write fails. If thrown the
   *                              cache mutation will not occur.
   */
  void write(Cache.Entry<? extends K, ? extends V> entry) throws
CacheWriterException;

  /**
   * Write the specified entries to the external resource. This method is intended
   * to support both insert and update.
```

```
     * <p/>
     * The order in which individual writes occur is undefined, as
     * {@link Cache#putAll(java.util.Map)} also has undefined ordering.
     * <p/>
     * If this operation fails (by throwing an exception) after a partial success,
     * the writer must remove any successfully written entries from the entries
     * collection so that the caching implementation knows what succeeded and can
     * mutate the cache.
     *
     * @param entries a mutable collection to write. Upon invocation, it contains
     *                the entries to write for write-through. Upon return the
     *                collection must only contain entries that were not
     *                successfully written. (see partial success above)
     * @throws CacheWriterException if one or more of the writes fail. If
     *                              thrown cache mutations will occur for
     *                              entries which succeeded.
     */
    void writeAll(Collection<Cache.Entry<? extends K, ? extends V>> entries) throws
        CacheWriterException;


    /**
     * Delete the cache entry from the external resource.
     * <p/>
     * Expiry of a cache entry is not a delete hence will not cause this method to
     * be invoked.
     * <p/>
     * This method is invoked even if no mapping for the key exists.
     *
     * @param key the key that is used for the delete operation
     * @throws CacheWriterException if delete fails. If thrown the cache delete will
     *                              not occur.
     */
    void delete(Object key) throws CacheWriterException;


    /**
     * Remove data and keys from the external resource for the given collection of
     * keys, if present.
     * <p/>
     * The order in which individual deletes occur is undefined, as
     * {@link Cache#removeAll(java.util.Set)} also has undefined ordering.
     * <p/>
     * If this operation fails (by throwing an exception) after a partial success,
     * the writer must remove any successfully written entries from the entries
     * collection so that the caching implementation knows what succeeded and can
     * mutate the cache.
     * <p/>
     * Expiry of a cache entry is not a delete hence will not cause this method to
     * be invoked.
     * <p/>
```

```
   * This method is only invoked for keys that exist in the cache.
   *
   * @param keys a mutable collection of keys for entries to delete. Upon
   *             invocation, it contains the keys to delete for write-through.
   *             Upon return the collection must only contain the keys that were
   *             not successfully deleted. (see partial success above)
   * @throws CacheWriterException if one or more deletes fail. If thrown
   *                              cache deletes will occur for entries which
   *                              succeeded.
   */
  void deleteAll(Collection<?> keys) throws CacheWriterException;
}
```

These interfaces are used as described below.

7.1. Cache Loading

The Cache loadAll method is used to load values from an external resource and is defined as follows:

```
  /**
   * Asynchronously loads the specified entries into the cache using the
   * configured {@link CacheLoader} for the given keys.
   * <p/>
   * If an entry for a key already exists in the Cache, a value will be loaded
   * if and only if <code>replaceExistingValues</code> is true.   If no loader
   * is configured for the cache, no objects will be loaded.  If a problem is
   * encountered during the retrieving or loading of the objects,
   * an exception is provided to the {@link CompletionListener}.  Once the
   * operation has completed, the specified CompletionListener is notified.
   * <p/>
   * Implementations may choose to load multiple keys from the provided
   * {@link Set} in parallel.  Iteration however must not occur in parallel,
   * thus allow for non-thread-safe {@link Set}s to be used.
   * <p/>
   * The thread on which the completion listener is called is implementation
   * dependent. An implementation may also choose to serialize calls to
   * different CompletionListeners rather than use a thread per
   * CompletionListener.
   *
   * @param keys                  the keys to load
   * @param replaceExistingValues when true existing values in the Cache will
   *                              be replaced by those loaded from a CacheLoader
   * @param completionListener    the CompletionListener (may be null)
   * @throws NullPointerException  if keys is null or if keys contains a null.
   * @throws IllegalStateException if the cache is {@link #isClosed()}
   * @throws CacheException        thrown if there is a problem performing the
   *                               load. This may also be thrown on calling if
   *                               there are insufficient threads available to
   *                               perform the load.
```

```
   * @throws ClassCastException    if the implementation supports and is
   *                               configured to perform runtime-type-checking,
   *                               and any key type is incompatible with that
   *                               which has been configured for the
   *                               {@link Cache}
   */
  void loadAll(Set<? extends K> keys, boolean replaceExistingValues,
               CompletionListener completionListener);
```

For this method to be used a `CacheLoader` must have been set in `Configuration` when the cache was created. A cache is not required to be set to read-through caching mode to use this method.

Loading may take a significant amount of time. For this reason a `CompletionListener` can be passed in which is notified on completion or on exception. It is defined as follows:

```
/**
 * A CompletionListener is implemented by an application when it needs to be
 * notified of the completion of some Cache operation.
 * <p/>
 * When the operation is complete, the Cache provider notifies the application
 * by calling the {@link #onCompletion()} method of the {@link
 * CompletionListener}.
 * <p/>
 * If the operation fails for any reason, the Cache provider calls the
 * {@link #onException(Exception)} method of the {@link CompletionListener}.
 * <p/>
 * To support a Java Future-based approach to synchronously wait for a Cache
 * operation to complete, use a {@link CompletionListenerFuture}.
 * <p/>
 * A CompletionListener will use an implementation specific thread for the call.
 *
 * @author Brian Oliver
 * @see CompletionListenerFuture
 */
public interface CompletionListener {

  /**
   * Notifies the application that the operation completed successfully.
   */
  void onCompletion();

  /**
   * Notifies the application that the operation failed.
   *
   * @param e the Exception that occurred
   */
```

```
  void onException(Exception e);
}
```

There is also a blocking implementation of `CompletionListener`, `CompletionListenerFuture`. It implements both `CompletionListener` and `Future`. If the `onException(Exception e)` method of `CompletionListener` is called, the exception is wrapped in `ExecutionException` and rethrown by the `Future get()` and `get(long timeout, TimeUnit unit)` methods.

7.1.1. Example 1

Using CompletionListenerFuture.

```
HashSet<String> keys = new HashSet<>();
keys.add("23432lkj");
keys.add("4fsdldkj");


//create a completion future to use to wait for loadAll
CompletionListenerFuture future = new CompletionListenerFuture();

//load the values for the set of keys, replacing those that may already
//exist in the cache
cache.loadAll(keys, true, future);

//wait for the cache to load the keys
try {
  future.get();
} catch (InterruptedException e) {
  //future interrupted
  e.printStackTrace();
} catch (ExecutionException e) {
  //throwable was what was sent to onException(Exception e)
  Throwable throwable = e.getCause();
}
```

The `loadAll` method is useful for pre-loading a cache with data from an external resource. It may be required because the application logic assumes the data is there. Another usage is cache warming. Here it will not cause an application error if the data is absent from the cache, but it will affect performance or scalability.

7.2. Read-Through Caching

A read-through cache behaves exactly the same way as a non-read-through cache except that certain accessor methods will invoke the CacheLoader if the entry or entries are missing from the Cache.

Read-Through caching is set at configuration time by calling `setReadThrough(boolean isReadThrough)` on `MutableConfiguration`. A `CacheLoader Factory` must also have been defined. The CacheLoader is used to load entries from an external resource.

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
`<T> T invoke(K key, EntryProcessor<K, V, T> entryProcessor, Object... arguments)entryProcessor);`	Yes, if getValue() called.
`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	Yes, if getValue called.
`Iterator<Cache.Entry<K, V>> iterator()`	No
`void loadAll(Set<? extends K> keys, boolean replaceExistingValues, CompletionListener completionListener)`	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(Set<? extends K> keys)`	No
`boolean replace(K key, V value)`	No
`boolean replace(K key, V oldValue, V newValue)`	No

Read-through caching is a useful idiom for lazily loading a cache. It is also useful in shielding the Cache user from the details of how an external resource is loaded into the cache.

When it is important for some or all cached content to be pre-loaded, use the l`oadAll` method.

7.3. Write-Through Caching

A write-through cache behaves exactly the same way as a non-write-through cache except that certain mutative methods will invoke the CacheWriter.

Write-Through caching is set at configuration time by calling `setWriteThrough(boolean isWriteThrough)` on `MutableConfiguration`. A `CacheWriter Factory` must also have been defined. The CacheWriter is used to write and remove entries from an external resource.

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

Method	**Invoke Write-Through**
`boolean containsKey(K key)`	No
`V get(K key)`	No
`Map<K,V> getAll(Collection<? extends K> keys)`	No
`V getAndPut(K key, V value)`	Yes
`V getAndRemove(K key)`	Yes
`V getAndReplace(K key, V value)`	Yes, if this method returns true
`<T> T invoke(K key, EntryProcessor<K, V, T> entryProcessor, Object... arguments)`	Yes, if setValue() is called.
`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	Yes, if setValue() is called.
`Iterator<Cache.Entry<K, V>> iterator()`	No
`void loadAll(Set<? extends K> keys, boolean replaceExistingValues, CompletionListener completionListener)`	No
`void put(K key, V value)`	Yes
`void putAll(Map<? extends K,? extends V> map)`	Yes, writeAll will be called

`boolean putIfAbsent(K key, V value)`	Yes, if this method returns true
`boolean remove(K key)`	Yes, even if no mapping exists
`boolean remove(K key, V oldValue)`	Yes, if this method returns true
`void removeAll()`	Yes, `deleteAll` will be called for entries in the cache
`void removeAll(Set<? extends K> keys)`	Yes, `removeAll(Set)` will be called even if mappings do not exist on some or all of the keys
`void clear()`	No
`boolean replace(K key, V value)`	Yes, if this method returns true
`boolean replace(K key, V oldValue, V newValue)`	Yes, if this method returns true

Write-through caching is a useful idiom for keeping an external resource updated with cache changes. It shields the Cache user from the details of how an external resource is written to.

8. Cache Entry Listeners

The `javax.cache.event` package contains classes and interfaces for handling events produced by a Cache.

8.1. Events and Event Types

A `CacheEntryEvent` is defined as follows:

```
/**
 * A Cache entry event base class.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Greg Luck
 * @since 1.0
 */
public abstract class CacheEntryEvent<K, V> extends EventObject
    implements Cache.Entry<K, V> {

  private EventType eventType;

  /**
   * Constructs a cache entry event from a given cache as source
   *
   * @param source the cache that originated the event
   */
  public CacheEntryEvent(Cache source, EventType eventType) {
    super(source);
    this.eventType = eventType;
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public final Cache getSource() {
    return (Cache) super.getSource();
  }

  /**
   * Returns the previous value, that existed prior to the
   * modification of the Entry value.
   *
   * @return the previous value or <code>null</code> if there was no previous value
   */
  public abstract V getOldValue();

  /**
   * Whether the old value is available.
   *
```

```
   * @return true if the old value is populated
   */
  public abstract boolean isOldValueAvailable();

  /**
   * Gets the event type of this event
   *
   * @return the event type.
   */
  public final EventType getEventType() {
    return eventType;
  }
}
```

There are four types of event, as enumerated by the EventType enum, defined as follows:

```
/**
 * The type of event received by the listener.
 *
 * @author Greg Luck
 */
public enum EventType {

  /**
   * An event type indicating that the cache entry was created.
   */
  CREATED,

  /**
   * An event type indicating that the cache entry was updated. i.e. a previous
   * mapping existed
   */
  UPDATED,


  /**
   * An event type indicating that the cache entry was removed.
   */
  REMOVED,


  /**
   * An event type indicating that the cache entry has expired.
   */
  EXPIRED

}
```

8.2. CacheEntryListeners

Events are propagated to CacheEntryListeners registered with a Cache through a CacheEntryListenerConfiguration. The CacheEntryListener interface is defined as follows:

```
/**
 * A tagging interface for cache entry listeners.
 * <p>
 * Sub-interfaces exist for the various cache events allowing a listener to be
 * created that implements only those listeners it is interested in.
 * <p>
 * Listeners should be implemented with care. In particular it is important to
 * consider their impact on performance and latency.
 * <p>
 * Listeners:
 * <ul>
 * <li>are fired after the entry is mutated in the cache</li>
 * <li>if synchronous are fired, for a given key, in the order that events
 * occur</li>
 * <li>block the calling thread until the listener returns,
 * where the listener was registered as synchronous</li>
 * <li>that are asynchronous iterate through multiple events with an undefined
 * ordering, except that events on the same key are in the order that the
 * events occur.</li>
 * </ul>
 * Listeners follow the observer pattern. An exception thrown by a
 * listener does not cause the cache operation to fail.
 * <p>
 * Listeners can only throw {@link CacheEntryListenerException}. Caching
 * implementations must catch any other {@link Exception} from a listener, then
 * wrap and rethrow it as a {@link CacheEntryListenerException}.
 * <p>
 * A listener that mutates a cache on the CacheManager may cause a deadlock.
 * Detection and response to deadlocks is implementation specific.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Yannis Cosmadopoulos
 * @author Greg Luck
 * @see CacheEntryCreatedListener
 * @see CacheEntryUpdatedListener
 * @see CacheEntryRemovedListener
 * @see CacheEntryExpiredListener
 * @see EventType
 * @since 1.0
 */
public interface CacheEntryListener<K, V> extends EventListener {

}
```

There are four sub-interfaces, corresponding to each of the `EventType`s, defined as follows:

```
/**
 * Invoked after a cache entry is created, or if a batch call is made, after the
 * entries are created.
 * <p/>
 * If an entry for the key existed prior to the operation it is not invoked,
 * instead {@link CacheEntryUpdatedListener} is invoked.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Yannis Cosmadopoulos
 * @author Greg Luck
 * @see CacheEntryUpdatedListener
 * @since 1.0
 */
public interface CacheEntryCreatedListener<K, V> extends CacheEntryListener<K, V> {

  /**
   * Called after one or more entries have been created.
   *
   * @param events The entries just created.
   * @throws CacheEntryListenerException if there is problem executing the listener
   */
  void onCreated(Iterable<CacheEntryEvent<? extends K, ? extends V>> events)
      throws CacheEntryListenerException;

}

/**
 * Invoked if an existing cache entry is updated, or if a batch call is made,
 * after the entries are updated.
 * <p/>
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Yannis Cosmadopoulos
 * @author Greg Luck
 * @see CacheEntryCreatedListener
 * @since 1.0
 */
public interface CacheEntryUpdatedListener<K, V> extends CacheEntryListener<K, V> {

  /**
   * Called after one or more entries have been updated.
   *
   * @param events The entries just updated.
   * @throws CacheEntryListenerException if there is problem executing the listener
   */
```

```
  void onUpdated(Iterable<CacheEntryEvent<? extends K, ? extends V>> events)
      throws CacheEntryListenerException;
}

/**
 * Invoked if a cache entry is removed, or if a batch call is made, after the
 * entries are removed.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Yannis Cosmadopoulos
 * @author Greg Luck
 * @since 1.0
 */
public interface CacheEntryRemovedListener<K, V> extends CacheEntryListener<K, V> {

  /**
   * Called after one or more entries have been removed. If no entry existed for
   * a key an event is not raised for it.
   *
   * @param events The entries just removed.
   * @throws CacheEntryListenerException if there is problem executing the listener
   */
  void onRemoved(Iterable<CacheEntryEvent<? extends K, ? extends V>> events)
      throws CacheEntryListenerException;


}

/**
 * Invoked if a cache entry or entries are evicted due to expiration.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Greg Luck
 * @since 1.0
 */
public interface CacheEntryExpiredListener<K, V> extends CacheEntryListener<K, V> {

  /**
   * Called after one or more entries have been expired by the cache. This is not
   * necessarily when an entry is expired, but when the cache detects the expiry.
   *
   * @param events The entries just removed.
   * @throws CacheEntryListenerException if there is problem executing the listener
   */
  void onExpired(Iterable<CacheEntryEvent<? extends K, ? extends V>> events)
      throws CacheEntryListenerException;

}
```

The motivation for this design is to allow efficient implementation of distributed listeners.

8.3. Registration of Listeners

Listeners are not assumed to be in-process with a cache. To avoid registration of instances which may not support Serialization, instead `CacheEntryListenerConfiguration`s are used. These may be added to `MutableConfiguration` using `MutableConfiguation.addCacheEntryListenerConfiguration` at configuration time or to a `Cache` using `Cache.registerCacheEntryListener` at runtime.

Listeners may be deregistered at runtime using `Cache.deregisterCacheEntryListener`.

These are defined as shown below:

```
/**
 * Defines the configuration requirements for a
 * {@link javax.cache.event.CacheEntryListener} and a {@link Factory} for its
 * creation.
 *
 * @param <K> the type of keys
 * @param <V> the type of values
 * @author Brian Oliver
 * @author Greg Luck
 */
public interface CacheEntryListenerConfiguration<K, V> {
  /**
   * Obtains the {@link Factory} for the
   * {@link javax.cache.event.CacheEntryListener}.
   *
   * @return the {@link Factory} for the
   *         {@link javax.cache.event.CacheEntryListener}
   */
  Factory<CacheEntryListener<? super K, ? super V>> getCacheEntryListenerFactory();

  /**
   * Determines if the old value should be provided to the
   * {@link CacheEntryListener}.
   *
   * @return <code>true</code> if the old value is required by the
   *         {@link CacheEntryListener}
   */
  boolean isOldValueRequired();

  /**
   * Obtains the {@link Factory} for the
   * {@link javax.cache.event.CacheEntryEventFilter} that
   * should be applied prior to notifying the {@link CacheEntryListener}.
   * When <code>null</code> no filtering is applied and all appropriate events
   * are notified.
   *
   * @return the {@link Factory} for the
```

```
     *          {@link javax.cache.event.CacheEntryEventFilter} or <code>null</code>
     *          if no filtering is required
     */
    Factory<CacheEntryEventFilter<? super K, ? super V>>
    getCacheEntryEventFilterFactory();

    /**
     * Determines if the thread that caused an event to be created should be
     * blocked (not return from the operation causing the event) until the
     * {@link CacheEntryListener} has been notified.
     *
     * @return <code>true</code> if the thread that created the event should block
     */
    boolean isSynchronous();
}
```

For convenience, a `MutableCacheEntryListenerConfiguration` implementation is provided in the `javax.cache.configuration` package.

Multiple `CacheEntryListenerConfiguration`s can be added to a `Configuration`. When the cache is initiated the `CacheEntryListener`s are created using the `Factory` are registered. A cache may have any number of listeners for the same or different `EventType`s. There is no ordering guarantee between listeners for their creation or dispatch of events.

8.4. Invocation of Listeners

Cache Listeners:

- are fired after the entry is mutated in the cache
- if synchronous, are fired, for a given key, in the order in that events occur, blocking the calling thread until the listener returns
- if asynchronous, iterate through multiple events with an undefined ordering, except that events on the same key must be processed in the order in that the events occur.

Listeners follow the observer pattern. An exception thrown by a listener does not cause the cache operation to fail.

A listener that mutates a cache on the CacheManager may cause a deadlock. Detection and response to deadlocks is implementation specific.

A registered listener will be invoked at most once by a caching implementation for each event.

i.e. listeners have once across a cluster semantics, not broadcast and execute in each node semantics.

A listener is not required to be in-process with the originating event.

In a distributed implementation, the listener may be implemented anywhere.

A listener may have a `CacheEntryEventFilter`, as part of its `CacheEntryListenerConfiguration`. These are defined as shown below:

```
/**
 * A function which may be used to check {@link CacheEntryEvent}s prior to being
 * dispatched to {@link CacheEntryListener}s.
 * <p/>
 * A filter must not create side effects.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 * @author Greg Luck
 * @author Brian Oliver
 * @since 1.0
 */
public interface CacheEntryEventFilter<K, V> {

  /**
   * Evaluates specified {@link CacheEntryEvent}.
   *
   * @param event the event that occurred
   * @return true if the evaluation passes, otherwise false.
   *         The effect of returning true is that listener will be invoked
   * @throws CacheEntryListenerException if there is problem executing the listener
   */
  boolean evaluate(CacheEntryEvent<? extends K, ? extends V> event)
      throws CacheEntryListenerException;
}
```

A `CacheEntryEventFilter` is not required to be in-process with the originating event.

In a distributed implementation, the filter may be implemented wherever it gives the best performance advantage.

The table below summarises the listeners that are invoked by each cache operation. Conditions are on the state of the entry before the operation. Expiry is always “No”. The exact timing of expiry is caching implementation specific.

Method	**Created**	**Expired**	**Removed**	**Update**
`boolean containsKey(K key)`	No	No	No	No
`V get(K key)`	Yes, if created by read-through	No	No	No
`Map<K,V> getAll(Collection<? extends K> keys)`	Yes, if created by read-	No	No	No

	through			
`V getAndPut(K key, V value)`	if missing	No	No	if there
`V getAndRemove(K key)`	No	No	if there	No
`V getAndReplace(K key, V value)`	No	No	No	if there
`<T> T invoke(K key, EntryProcessor<K, V> entryProcessor);`	Yes, if setValue() created an entry, or getValue() created an entry by read-through	No	Yes, if remove() was called	Yes, if setValue() updated an entry
`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	Yes, if setValue() created an entry, or getValue() created an entry by read-through	No	Yes, if remove() was called	Yes, if setValue() updated an entry
`Iterator<Cache.Entry<K, V>> iterator()`	No	No	Yes, if remove() was called	No
`void loadAll(Set<? extends K> keys,boolean replaceExistingValues, CompletionListener completionListener);`	if missing	No	No	if there
`void put(K key, V value)`	if missing	No	No	if there
`void putAll(Map<? extends K,? extends V> map)`	if missing	No	No	if there
`boolean putIfAbsent(K key, V value)`	if missing	No	No	No
`boolean remove(K key)`	No	No	if there	No
`boolean remove(K key, V oldValue)`	No	No	if there && equal	No

`void removeAll()`	No	No	if there	No
`void removeAll(Set<? extends K> keys)`	No	No	if there	No
`boolean replace(K key, V value)`	No	No	No	if there
`boolean replace(K key, V oldValue, V newValue)`	No	No	No	if there && equal

9. Entry Processors

A `javax.cache.processor.EntryProcessor` is an invocable function, much like a `java.util.concurrent.Callable`, that applications may use to efficiently perform compound Cache operations, including access, update and removal atomically on a Cache Entry, without requiring explicit locking or transactions.

When invoked using either the `Cache#invoke` or `Cache#invokeAll` methods, an `EntryProcessor` is provided with a `MutableEntry`, that of which allows an application to exclusively have access to the entry.

The `javax.cache.processor.MutableEntry` is defined as follows:

```
/**
 * A mutable representation of a {@link javax.cache.Cache.Entry}.
 *
 * @param <K> the type of key
 * @param <V> the type of value
 *
 * @author Greg Luck
 * @since 1.0
 */
public interface MutableEntry<K, V> extends Cache.Entry<K, V> {

  /**
   * Checks for the existence of the entry in the cache
   *
   * @return true if the entry exists
   */
  boolean exists();

  /**
   * Removes the entry from the Cache.
   * <p>
   * This has the same semantics as calling {@link Cache#remove}.
   */
  void remove();

  /**
   * Sets or replaces the value associated with the key.
   * <p>
   * If {@link #exists} is false and setValue is called
   * then a mapping is added to the cache visible once the EntryProcessor
   * completes. Moreover a second invocation of {@link #exists()}
   * will return true.
   *
   * @param value the value to update the entry with
   * @throws ClassCastException if the implementation supports and is
```

```
   *                              configured to perform runtime-type-checking,
   *                              and value type is incompatible with that
   *                              which has been configured for the
   *                              {@link Cache}
   */
  void setValue(V value);
}
```

For example, an application that may wish to inspect the value of a Cache entry, calculate a new value, update the entry and return some other value atomically, could do so using a custom `EntryProcessor` implementation.

The `javax.cache.processor.EntryProcessor` interface is defined as follows:

```
/**
 * An invocable function that allows applications to perform compound operations
 * on a {@link javax.cache.Cache.Entry} atomically, according the defined
 * consistency of a {@link Cache}.
 * <p>
 * Any {@link javax.cache.Cache.Entry} mutations will not take effect until after
 * the {@link EntryProcessor#process(MutableEntry, Object...)} method has completed
 * execution.
 * <p>
 * If an exception is thrown by an {@link EntryProcessor}, a Caching Implementation
 * must wrap any {@link Exception} thrown wrapped in an {@link
 * EntryProcessorException}. If this occurs no mutations will be made to the
 * {@link javax.cache.Cache.Entry}.
 * <p>
 * Implementations may execute {@link EntryProcessor}s in situ, thus avoiding
 * locking, round-trips and expensive network transfers.
 *
 * <h3>Effect of {@link MutableEntry} operations</h3>
 * {@link javax.cache.Cache.Entry} access, via a call to
 * {@link javax.cache.Cache.Entry#getValue()}, will behave as if
 * {@link Cache#get(Object)} was called for the key.  This includes updating
 * necessary statistics, consulting the configured {@link ExpiryPolicy} and loading
 * from a configured {@link javax.cache.integration.CacheLoader}.
 * <p>
 * {@link javax.cache.Cache.Entry} mutation, via a call to
 * {@link MutableEntry#setValue(Object)}, will behave as if {@link
 * Cache#put(Object, Object)} was called for the key. This includes updating
 * necessary statistics, consulting the configured {@link
 * ExpiryPolicy}, notifying {@link CacheEntryListener}s and writing to a
 * configured {@link CacheWriter}.
 * <p>
 * {@link javax.cache.Cache.Entry} removal, via a call to
 * {@link MutableEntry#remove()}, will behave as if {@link Cache#remove(Object)}
 * was called for the key. This includes updating necessary statistics, notifying
 * {@link CacheEntryListener}s and causing a delete on a configured
 * {@link CacheWriter}.
 * <p>
```

```
* As implementations may choose to execute {@link EntryProcessor}s remotely,
* {@link EntryProcessor}s, together with specified parameters and return
* values, may be required to implement {@link java.io.Serializable}.
*
* <h3>Effect of multiple {@link MutableEntry} operations performed by one {@link
* EntryProcessor}</h3>
* Only the net effect of multiple operations has visibility outside of the Entry
* Processor. The entry is locked by the entry processor for the entire scope
* of the entry processor, so intermediate effects are not visible.
* <h4>Example 1</h4>
* In this example, an {@link EntryProcessor} calls:
* <ol>
* <li>{@link MutableEntry#getValue()}</li>
* <li>{@link MutableEntry#setValue(Object)}</li>
* <li>{@link MutableEntry#getValue()}</li>
* <li>{@link MutableEntry#setValue(Object)}</li>
* </ol>
* This will have the following {@link Cache} effects:
* <br>
* Final value of the cache: last setValue<br>
* Statistics: one get and one put as the second get and the first put are
* internal to the EntryProcessor.<br>
* Listeners: second put will cause either a put or an update depending on whether
* there was an initial value for the entry.<br>
* CacheLoader: Invoked by the first get only if a loader was registered.<br>
* CacheWriter: Invoked by the second put only as the first put was internal to
* the Entry Processor.<br>
* ExpiryPolicy: The first get and the second put only are visible to the
* ExpiryPolicy.<br>
*
* <h4>Example 2</h4>
* In this example, an {@link EntryProcessor} calls:
* <ol>
* <li>{@link MutableEntry#getValue()}</li>
* <li>{@link MutableEntry#remove()}}</li>
* <li>{@link MutableEntry#getValue()}</li>
* <li>{@link MutableEntry#setValue(Object)}</li>
* </ol>
* This will have the following {@link Cache} effects:
* <br>
* Final value of the cache: last setValue<br>
* Statistics: one get and one put as the second get and the first put are
* internal to the EntryProcessor.<br>
* Listeners: second put will cause either a put or an update depending on whether
* there was an initial value for the entry.<br>
* CacheLoader: Invoked by the first get only if a loader was registered.<br>
* CacheWriter: Invoked by the second put only as the first put was internal to
* the Entry Processor.<br>
* ExpiryPolicy: The first get and the second put only are visible to the
* ExpiryPolicy.<br>
*
* <h4>Example 3</h4>
```

```
 * In this example, an {@link EntryProcessor} calls:
 * <ol>
 * <li>{@link MutableEntry#getValue()}</li>
 * <li>{@link MutableEntry#setValue(Object)}}</li>
 * <li>{@link MutableEntry#getValue()}</li>
 * <li>{@link MutableEntry#setValue(Object)}</li>
 * <li>{@link MutableEntry#remove()}</li>
 * </ol>
 * This will have the following {@link Cache} effects:
 * <br>
 * Final value of the cache: last setValue<br>
 * Statistics: one get and one remove as the second get and the two puts are
 * internal to the EntryProcessor.<br>
 * Listeners: remove if there was initial value in the cache, otherwise no
 * listener invoked.
 * <br> CacheLoader: Invoked by the first get only if a loader was
 * registered.
 * <br> CacheWriter: Invoked by the remove only as the two puts are internal to
 * the Entry Processor.<br>
 * ExpiryPolicy: The first get only is visible to the ExpiryPolicy. There is no
 * remove event in ExpiryPolicy.
 *
 * @param <K> the type of keys maintained by this cache
 * @param <V> the type of cached values
 * @param <T> the type of the return value
 * @author Greg Luck
 * @since 1.0
 */
public interface EntryProcessor<K, V, T> {

  /**
   * Process an entry.
   *
   * @param entry     the entry
   * @param arguments a number of arguments to the process.
   * @return the user-defined result of the processing, if any.
   * @throws EntryProcessorException if there is a failure in entry processing.
   */
  T process(MutableEntry<K, V> entry, Object... arguments)
      throws EntryProcessorException;
}
```

To invoke an `EntryProcessor` on a Cache Entry, applications must use the `Cache.invoke` method for invocation against single keys and the `Cache.invokeAll` method for invocation against sets of keys.

```
  /**
   * Invokes an {@link EntryProcessor} against the {@link Entry} specified by
   * the provided key. If an {@link Entry} does not exist for the specified key,
   * an attempt is made to load it (if a loader is configured) or a surrogate
   * {@link Entry}, consisting of the key with a null value is used instead.
```

```
     * <p>
     *
     * @param key            the key to the entry
     * @param entryProcessor the {@link EntryProcessor} to invoke
     * @param arguments      additional arguments to pass to the
     *                       {@link EntryProcessor}
     * @return the result of the processing, if any, defined by the
     *         {@link EntryProcessor} implementation
     * @throws NullPointerException    if key or {@link EntryProcessor} is null
     * @throws IllegalStateException   if the cache is {@link #isClosed()}
     * @throws ClassCastException    if the implementation is configured to perform
     *                               runtime-type-checking, and the key or value
     *                               types are incompatible with those that have been
     *                               configured for the {@link Cache}
     * @throws EntryProcessorException if an exception is thrown by the {@link
     *                                 EntryProcessor}, a Caching Implementation
     *                                 must wrap any {@link Exception} thrown
     *                                 wrapped in an {@link EntryProcessorException}.
     * @see EntryProcessor
     */
    <T> T invoke(K key,
                 EntryProcessor<K, V, T> entryProcessor,
                 Object... arguments) throws EntryProcessorException;

    /**
     * Invokes an {@link EntryProcessor} against the set of {@link Entry}s
     * specified by the set of keys.
     * <p>
     * If an {@link Entry} does not exist for the specified key, an attempt is made
     * to load it (if a loader is configured) or a surrogate {@link Entry},
     * consisting of the key and a value of null is provided.
     * <p>
     * The order that the entries for the keys are processed is undefined.
     * Implementations may choose to process the entries in any order, including
     * concurrently.  Furthermore there is no guarantee implementations will
     * use the same {@link EntryProcessor} instance to process each entry, as
     * the case may be in a non-local cache topology.
     * <p>
     * The result of executing the {@link EntryProcessor} is returned as a
     * {@link Map} of {@link EntryProcessorResult}s, one result per key.  Should the
     * {@link EntryProcessor} or Caching implementation throw an exception, the
     * exception is wrapped and re-thrown when a call to
     * {@link javax.cache.processor.EntryProcessorResult#get()} is made.
     *
     * @param keys           the set of keys for entries to process
     * @param entryProcessor the {@link EntryProcessor} to invoke
     * @param arguments      additional arguments to pass to the
     *                       {@link EntryProcessor}
     * @return the map of {@link EntryProcessorResult}s of the processing per key,
     * if any, defined by the {@link EntryProcessor} implementation.  No mappings
     * will be returned for {@link EntryProcessor}s that return a
     * <code>null</code> value for a key.
```

```
   * @throws NullPointerException    if keys or {@link EntryProcessor} are null
   * @throws IllegalStateException   if the cache is {@link #isClosed()}
   * @throws ClassCastException    if the implementation is configured to perform
   *                               runtime-type-checking, and the key or value
   *                               types are incompatible with those that have been
   *                               configured for the {@link Cache}
   * @see EntryProcessor
   */
  <T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys,
                                                EntryProcessor<K, V, T>
                                                  entryProcessor,
                                                Object... arguments);
```

The following example demonstrates atomically incrementing the value of an entry with an `EntryProcessor`.

```
    CachingProvider provider = Caching.getCachingProvider();
    CacheManager manager = provider.getCacheManager();

    MutableConfiguration<String, Integer> configuration =
      new MutableConfiguration<String, Integer>()
        .setTypes(String.class, Integer.class);

    Cache<String, Integer> cache = manager.createCache("example", configuration);

    String key = "counter";
    cache.put(key, 1);

    int previous = cache.invoke(key, new IncrementProcessor<String>());

    assert previous == 1;
    assert cache.get(key) == 2;
```

With the `IncrementProcessor` Entry Processor being defined as follows:

```
  /**
   * An {@link EntryProcessor} that increments an {@link Integer}.
   *
   * @param <K> the type of keys
   */
  public static class IncrementProcessor<K> implements EntryProcessor<K,
      Integer, Integer>, Serializable {

    /**
     * The serialVersionUID required for {@link java.io.Serializable}.
     */
    public static final long serialVersionUID = 201306211238L;

    /**
```

```
     * {@inheritDoc}
     */
    @Override
    public Integer process(MutableEntry<K, Integer> entry, Object... arguments) {
      if (entry.exists()) {
        Integer current = entry.getValue();
        entry.setValue(current + 1);
        return current;
      } else {
        entry.setValue(0);
        return -1;
      }
    }
  }
```

Implementations that support remote or distributed Caching Topologies may choose to execute Entry Processors in a remote process. In such circumstances implementations may require EntryProcessors, the invocation parameters and return types to implement java.lang.Serializable or be serializable in some manner. Alternatively implementations may choose to simply serialize the EntryProcessor class name, together with the specified invocation parameters and execute the request remotely by instantiating the EntryProcessor class and calling it with the deserialized invocation parameters.

As the outcome of an `EntryProcessor` is atomic, so are the interactions with Cache Loaders, Cache Writers, Cache Entry Listeners and Expiry Policies.

An application will never observe the intermediate events or side-effect for individual calls to `MutableEntry getValue`, `setValue`, `remove` etc while an `EntryProcessor` is being invoked. Rather applications will only observe the "net" result of an operation performed by an `EntryProcessor` on a `Cache.Entry`.

For Example: An `EntryProcessor` that has the following calls:

```
V v1 = entry.getValue();
entry.setValue(v2);
entry.remove();
entry.setValue(v3);
v4 = entry.getValue();
```

Will produce a single `CacheEntryListeneEvent`; an update from v1 to v3.

9.1. Exceptions in EntryProcessors

Should an exception be thrown during the invocation of an `EntryProcessor,` either by the Caching implementation or the `EntryProcessor` itself, the exception must be returned to the calling application wrapped as an `javax.cache.processor.EntryProcessorException.`

When an exception occurs as part of invoking an `EntryProcessor` using the `Cache.invoke` method, the exception is wrapped as an `EntryProcessorException` and re-throw to the calling application.

When an exception occurs as part of invoking an `EntryProcessor` using the `Cache.invokeAll` method, the exception is wrapped and returned as part of an `EntryProcessorResult` to the calling application.

9.2. EntryProcessorResults for Cache.invokeAll

When invoking an `EntryProcessor` against a set of keys using the `Cache.invokeAll` method, the result is a `Map` of `EntryProcessorResults,` one per key (unless an `EntryProcessor` invocation for a key resulted in null in which case no `javax.cache.processor.EntryProcessorResult` is returned for the key in the result Map). To retrieve the value returned by the `EntryProcessor` for a specific key, applications should call the `EntryProcessorResult.get` method for the `EntryProcessorResult` for the key. If the `EntryProcessor` produced an exception for the key, calling `EntryProcessorResult.get` will raise the corresponding wrapped exception.

10. Caching Providers

Caching Providers are a core concept of the Java Caching API. It is through a `CachingProvider` that developers acquire `CacheManager`s, so that they may interact with `Cache`s.

A `CachingProvider` provides a means of:

- acquiring a default `CacheManager` instance
- establishing `CacheManager` instances, uniquely identified by implementation specific `URI`s

  e.g. an implementation might request a CacheManager configured declaratively with an implementation specific configuration file on the classpath

  ```
  cachingProvider.getCacheManager("/sample/ConfigurationFile.xml");
  ```

- scoping and managing `CacheManager` instances by `URI` and `ClassLoader`
- closing and releasing specific or collections of related `CacheManager`s
- querying the capabilities of a `CachingProvider` implementation, including support for optional features.

10.1.1. CacheManager Identity and Configuration

- `CacheManager`s are logically identified by the `URI` that was used to establish them within the context of a `CachingProvider`.
- A `CacheManager` that is created using the default `URI` and default `ClassLoader`, provided by a `CachingProvider` is called the "Default Cache Manager". It is obtained with:

```
CacheManager getCacheManager();
```

While applications often use the default `URI` as defined by a `CachingProvider` as a means of acquiring a `CacheManager`, applications may additionally use implementation specific `URI`s for advanced configuration of `CacheManager`s.

For example an implementation may permit a `URI` to be used as the location of a configuration file, say for pre-configured caches.

```
cachingProvider.getCacheManager("/sample/ConfigurationFile.xml");
```

Two or more `CacheManager`s defined with the same `URI` and the same `ClassLoader` within an application deployment are said to be logically equivalent and must manage the same Caches.

The semantics of caches defined by the `URI` used to acquire a `CacheManager` is implementation dependent.

For example: Two or more applications using the same URI with an implementation supporting distributed caching topologies may logically share the cache content for caches of the same name. In such situations, changes to cache entries in one application will be visible to the other application.

Alternatively two or more applications using the same URI with an implementation that only supports local caching topologies may not be able to share cache content, even when the same cache names are used. In such situations changes to cache entries in one application may not be visible to the other application.

The following table outlines how CacheManager URI may affect visibility of cache entries in caches of the same name using implementations that support only local (non-shared) vs distributed (or shared) cache topologies.

CacheManager URI	**Local (non-shared) Cache Topology**	**Distributed / Shared Cache Topology**
Same	Caches will have the **same configurations**. Caches will have **different entries**.	Caches will have the **same configurations**. Caches will have the **same entries**.
Different	Caches will have **different configurations**. Caches will have **different entries**.	Caches will have **different configurations**. Caches will have **different entries**.

The `CachingProvider` interface is defined as follows:

```
/**
 * Provides mechanisms to create, request and later manage the life-cycle of
 * configured {@link CacheManager}s, identified by {@link URI}s and scoped by
 * {@link ClassLoader}s.
 * <p>
 * The meaning and semantics of the {@link URI} used to identify a
 * {@link CacheManager} is implementation dependent.  For applications to remain
 * implementation independent, they should avoid attempting to create {@link URI}s
 * and instead use those returned by {@link #getDefaultURI()}.
 *
 * @author Brian Oliver
 * @author Greg Luck
 * @since 1.0
 */
public interface CachingProvider extends Closeable {
```

```
/**
 * Requests a {@link CacheManager} configured according to the implementation
 * specific {@link URI} be made available that uses the provided
 * {@link ClassLoader} for loading underlying classes.
 * <p>
 * Multiple calls to this method with the same {@link URI} and
 * {@link ClassLoader} must return the same {@link CacheManager} instance,
 * except if a previously returned {@link CacheManager} has been closed.
 * <p>
 * Properties are used in construction of a {@link CacheManager} and do not form
 * part of the identity of the CacheManager. i.e. if a second call is made to
 * with the same {@link URI} and {@link ClassLoader} but different properties,
 * the {@link CacheManager} created in the first call is returned.
 *
 * @param uri         an implementation specific URI for the
 *                    {@link CacheManager} (null means use
 *                    {@link #getDefaultURI()})
 * @param classLoader the {@link ClassLoader}  to use for the
 *                    {@link CacheManager} (null means use
 *                    {@link #getDefaultClassLoader()})
 * @param properties  the {@link Properties} for the {@link CachingProvider}
 *                    to create the {@link CacheManager} (null means no
 *                    implementation specific Properties are required)
 * @throws CacheException    when a {@link CacheManager} for the
 *                           specified arguments could not be produced
 * @throws SecurityException when the operation could not be performed
 *                           due to the current security settings
 */
CacheManager getCacheManager(URI uri, ClassLoader classLoader,
                           Properties properties);

/**
 * Obtains the default {@link ClassLoader} that will be used by the
 * {@link CachingProvider}.
 *
 * @return the default {@link ClassLoader} used by the {@link CachingProvider}
 */
ClassLoader getDefaultClassLoader();

/**
 * Obtains the default {@link URI} for the {@link CachingProvider}.
 * <p>
 * Use this method to obtain a suitable {@link URI} for the
 * {@link CachingProvider}.
 *
 * @return the default {@link URI} for the {@link CachingProvider}
 */
URI getDefaultURI();

/**
 * Obtains the default {@link Properties} for the {@link CachingProvider}.
```

```
     * <p>
     * Use this method to obtain suitable {@link Properties} for the
     * {@link CachingProvider}.
     *
     * @return the default {@link Properties} for the {@link CachingProvider}
     */
    Properties getDefaultProperties();

    /**
     * Requests a {@link CacheManager} configured according to the implementation
     * specific {@link URI} that uses the provided {@link ClassLoader} for loading
     * underlying classes.
     * <p>
     * Multiple calls to this method with the same {@link URI} and
     * {@link ClassLoader} must return the same {@link CacheManager} instance,
     * except if a previously returned {@link CacheManager} has been closed.
     *
     * @param uri         an implementation specific {@link URI} for the
     *                    {@link CacheManager} (null means
     *                    use {@link #getDefaultURI()})
     * @param classLoader the {@link ClassLoader}  to use for the
     *                    {@link CacheManager} (null means
     *                    use {@link #getDefaultClassLoader()})
     * @throws CacheException    when a {@link CacheManager} for the
     *                           specified arguments could not be produced
     * @throws SecurityException when the operation could not be performed
     *                           due to the current security settings
     */
    CacheManager getCacheManager(URI uri, ClassLoader classLoader);

    /**
     * Requests a {@link CacheManager} configured according to the
     * {@link #getDefaultURI()} and {@link #getDefaultProperties()} be made
     * available that using the {@link #getDefaultClassLoader()} for loading
     * underlying classes.
     * <p>
     * Multiple calls to this method must return the same {@link CacheManager}
     * instance, except if a previously returned {@link CacheManager} has been
     * closed.
     *
     * @throws SecurityException when the operation could not be performed
     *                           due to the current security settings
     */
    CacheManager getCacheManager();

    /**
     * Closes all of the {@link CacheManager} instances and associated resources
     * created and maintained by the {@link CachingProvider} across all
     * {@link ClassLoader}s.
     * <p>
     * After closing the {@link CachingProvider} will still be operational.  It
```

```
   * may still be used for acquiring {@link CacheManager} instances, though
   * those will now be new.
   *
   * @throws SecurityException when the operation could not be performed
   *                           due to the current security settings
   */
  void close();

  /**
   * Closes all {@link CacheManager} instances and associated resources created
   * by the {@link CachingProvider} using the specified {@link ClassLoader}.
   * <p>
   * After closing the {@link CachingProvider} will still be operational.  It
   * may still be used for acquiring {@link CacheManager} instances, though
   * those will now be new for the specified {@link ClassLoader} .
   *
   * @param classLoader the {@link ClassLoader}  to release
   * @throws SecurityException when the operation could not be performed
   *                           due to the current security settings
   */
  void close(ClassLoader classLoader);

  /**
   * Closes all {@link CacheManager} instances and associated resources created
   * by the {@link CachingProvider} for the specified {@link URI} and
   * {@link ClassLoader}.
   *
   * @param uri         the {@link URI} to release
   * @param classLoader the {@link ClassLoader}  to release
   * @throws SecurityException when the operation could not be performed
   *                           due to the current security settings
   */
  void close(URI uri, ClassLoader classLoader);

  /**
   * Determines whether an optional feature is supported by the
   * {@link CachingProvider}.
   *
   * @param optionalFeature the feature to check for
   * @return true if the feature is supported
   */
  boolean isSupported(OptionalFeature optionalFeature);
}
```

Although optional, in Java SE environments the primary means of acquiring a `CachingProvider` instance is to use the `Caching` bootstrap class.

The `Caching` bootstrap provides three mechanisms for locating and instantiating one or more available `CachingProvider` implementations by:

- assuming implementations are defined as a Service and resolving them through the use of a `java.util.ServiceLoader`

- allowing a developer to specify the default implementation by using the `javax.cache.CachingProvider` Java System Property to define the fully qualified class name of the desired `CachingProvider`

- allowing applications to explicitly request a specific implementation using the fully qualified class name of the desired `CachingProvider`

While developers may alternatively use implementation dependent techniques for acquiring CachingProviders doing so may reduce the portability of their applications between CachingProvider implementations.

For a `CachingProvider` implementation to be automatically located by the Caching bootstrap class `java.util.ServiceLoader`, the fully qualified class name(s) of the `CachingProvider` implementation(s) an application will use must be defined in a `META-INF/services/javax.cache.spi.CachingProvider` configuration file as described in the JAR File Specification.

The `javax.cache.spi.CachingProvider` configuration file serves to define the specific `CachingProvider` implementation class(es) to the `Caching` bootstrap class, thus allowing it to automatically locate, load and provide appropriate instances to applications on request.

The content of a `javax.cache.spi.CachingProvider` configuration file is simply one or more fully qualified class names, each on a separate line, each specifying the name of an available `CachingProvider` implementation.

For example:
A Java Caching API implementor, ACME Caching Products, ships a JAR called `acme.jar`, that contains a `CachingProvider` implementation. The contents of the JAR includes both the `CachingProvider` implementation and the `javax.cache.spi.CachingProvider` configuration file.

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

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

```
com.acme.cache.ACMECachingProvider
```

Applications may use multiple CachingProvider implementations simply by correctly configuring the `META-INF/services/javax.cache.spi.CachingProvider` *file. When multiple CachingProviders are available, a request to return the default CachingProvider from the Caching bootstrap class will result in an exception.*

The methods defined by the `Caching` bootstrap class are defined as follows:

```
/**
 * The {@link Caching} class provides a convenient means for an application to
 * acquire an appropriate {@link CachingProvider} implementation.
 * <p/>
 * While defined as part of the specification, its use is not mandatory.
 * Applications and/or containers may instead choose to directly instantiate a
 * {@link CachingProvider} implementation based on implementation specific
 * instructions.
 * <p/>
 * When using the {@link Caching} class, {@link CachingProvider} implementations
 * are automatically discovered when they follow the conventions outlined by the
 * Java Development Kit {@link ServiceLoader} class.
 * <p/>
 * For a {@link CachingProvider} to be automatically discoverable by the
 * {@link Caching} class, the fully qualified class name of the
 * {@link CachingProvider} implementation must be declared in the following
 * file:
 * <pre>
 *   META-INF/services/javax.cache.spi.CachingProvider
 * </pre>
 * that of which is resolvable via the class path.
 * <p/>
 * For example, in the reference implementation the contents of this file are:
 * <code>org.jsr107.ri.RICachingProvider</code>
 * <p/>
 * Alternatively when the fully qualified class name of a
 * {@link CachingProvider} implementation is specified using the system property
 * <code>javax.cache.CachingProvider</code>, that implementation will be used
 * as the default {@link CachingProvider}.
 * <p/>
 * All {@link CachingProvider}s that are automatically detected or explicitly
 * declared and loaded by the {@link Caching} class are maintained in an
 * internal registry.  Consequently when a previously loaded
 * {@link CachingProvider} is requested, it will be simply returned from the
 * internal registry, without reloading and/or instantiating the
 * implementation again.
 * <p/>
 * As required by some applications and containers, multiple co-existing
 * {@link CachingProvider}s implementations, from the same or different
 * implementors are permitted at runtime.
 * <p/>
 * To iterate through those that are currently registered a developer may use
 * the following methods:
 * <ol>
 * <li>{@link #getCachingProviders()}</li>
 * <li>{@link #getCachingProviders(ClassLoader)}</li>
 * </ol>
 * To request a specific {@link CachingProvider} implementation, a developer
 * should use either the {@link #getCachingProvider(String)} or
```

```
 * {@link #getCachingProvider(String, ClassLoader)} method.
 * <p/>
 * Where multiple {@link CachingProvider}s are present, the
 * {@link CachingProvider} returned by getters {@link #getCachingProvider()} and
 * {@link #getCachingProvider(ClassLoader)} is undefined and as a result a
 * {@link CacheException} will be thrown when attempted.
 *
 * @author Brian Oliver
 * @author Greg Luck
 * @see java.util.ServiceLoader
 * @see javax.cache.spi.CachingProvider
 */
public final class Caching {

  /**
   * Obtains the {@link ClassLoader} to use for API methods that don't
   * explicitly require a {@link ClassLoader} but internally require one.
   * <p/>
   * By default this is the {@link Thread#getContextClassLoader()}.
   *
   * @return the default {@link ClassLoader}
   */
  public static ClassLoader getDefaultClassLoader()

  /**
   * Set the {@link ClassLoader} to use for API methods that don't explicitly
   * require a {@link ClassLoader}, but internally use one.
   *
   * @param classLoader the {@link ClassLoader} or <code>null</code> if the
   *                    calling {@link Thread#getContextClassLoader()} should
   *                    be used
   */
  public void setDefaultClassLoader(ClassLoader classLoader)

  /**
   * Obtains the single {@link CachingProvider} visible to the default
   * {@link ClassLoader}, which is {@link Thread#getContextClassLoader()}.
   *
   * @return the {@link CachingProvider}
   * @throws CacheException should zero, or more than one
   *                        {@link CachingProvider} be available on the
   *                        classpath, or it could not be loaded
   */
  public static CachingProvider getCachingProvider()

  /**
   * Obtains the single {@link CachingProvider} visible to the specified
   * {@link ClassLoader}.
   *
   * @param classLoader the {@link ClassLoader} to use for loading the
   *                    {@link CachingProvider}
   * @return the {@link CachingProvider}
   * @throws CacheException should zero, or more than one
   *                        {@link CachingProvider} be available on the
   *                        classpath, or it could not be loaded
   * @see #getCachingProviders(ClassLoader)
   */
  public static CachingProvider getCachingProvider(ClassLoader classLoader)
```

```
/**
 * Obtains the {@link CachingProvider}s that are available via the
 * {@link #getDefaultClassLoader()}.
 * <p/>
 * If a <code>javax.cache.cachingprovider</code> system property is defined,
 * only that {@link CachingProvider} specified by that property is returned.
 * Otherwise all {@link CachingProvider}s that are available via a
 * {@link ServiceLoader} for {@link CachingProvider}s using the default
 * {@link ClassLoader} (including those previously requested via
 * {@link #getCachingProvider(String)}) are returned.
 *
 * @return an {@link Iterable} of {@link CachingProvider}s loaded by the
 *         specified {@link ClassLoader}
 */
public static Iterable<CachingProvider> getCachingProviders()

/**
 * Obtains the {@link CachingProvider}s that are available via the specified
 * {@link ClassLoader}.
 * <p/>
 * If a <code>javax.cache.cachingprovider</code> system property is defined,
 * only that {@link CachingProvider} specified by that property is returned.
 * Otherwise all {@link CachingProvider}s that are available via a
 * {@link ServiceLoader} for {@link CachingProvider}s using the specified
 * {@link ClassLoader} (including those previously requested via
 * {@link #getCachingProvider(String, ClassLoader)}) are returned.
 *
 * @param classLoader the {@link ClassLoader} of the returned
 *                    {@link CachingProvider}s
 * @return an {@link Iterable} of {@link CachingProvider}s loaded by the
 *         specified {@link ClassLoader}
 */
public static Iterable<CachingProvider> getCachingProviders(
    ClassLoader classLoader)

/**
 * Obtain the {@link CachingProvider} that is implemented by the specified
 * fully qualified class name using the {@link #getDefaultClassLoader()}.
 * Should this {@link CachingProvider} already be loaded it is simply returned,
 * otherwise an attempt will be made to load and instantiate the specified
 * class (using a no-args constructor).
 *
 * @param fullyQualifiedClassName the fully qualified class name of the
 *                                {@link CachingProvider}
 * @return the {@link CachingProvider}
 * @throws CacheException if the {@link CachingProvider} cannot be created
 */
public static CachingProvider getCachingProvider(String fullyQualifiedClassName)

/**
 * Obtain the {@link CachingProvider} that is implemented by the specified
 * fully qualified class name using the provided {@link ClassLoader}.
 * Should this {@link CachingProvider} already be loaded it is returned,
 * otherwise an attempt will be made to load and instantiate the specified
 * class (using a no-args constructor).
 *
 * @param fullyQualifiedClassName the fully qualified class name of the
 *                                {@link CachingProvider}
 * @param classLoader             the {@link ClassLoader} to load the
```

```
   *                                {@link CachingProvider}
   * @return the {@link CachingProvider}
   * @throws CacheException if the {@link CachingProvider} cannot be created
   */
  public static CachingProvider getCachingProvider(
             String fullyQualifiedClassName,
             ClassLoader classLoader)
}
```

11. Caching Annotations

Caching annotations provide method interceptors for user supplied classes that interact with caches. The annotations and support classes are in the `javax.cache.annotation` package. The following annotations are defined:

- `@CacheDefaults`
- `@CacheResult`
- `@CachePut`
- `@CacheRemove`
- `@CacheRemoveAll`

Each annotation defines the underlying cache operations that are to be performed using the Java API. The same result must be achieved whether by annotation or by using the defined Java API operations.

Annotations therefore provide an additional API for interacting with caches. Annotations are provided only for the most commonly used cache methods.

In order to use annotations in an application, a library or framework that processes these annotations and intercepts calls to annotated application objects is required. In an application, the method and configuration of processing caching annotations on classes is left to the implementation.

This would commonly be provided by a dependency injection framework such as defined by CDI. The RI includes example implementations for use with CDI, Spring and Guice.

11.1. Annotations

11.1.1. `@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 specifies a cache named “cities” as the default cache name for annotations in the class. The `@CacheResult` annotation on the `getCity` method will use this cache name at runtime.

Example 1

This example shows how to use the @CacheDefaults Annotation.

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

```
        //...
    }
}
```

It is defined as follows:

```
/**
 * Allows the configuration of defaults for {@link CacheResult}, {@link CachePut},
 * {@link CacheRemove}, and {@link CacheRemoveAll} at the class level. Without
 * the method level annotations this annotation has no effect.
 * <p>
 * Following is an example of specifying a default cache name that is used by
 * the annotations on the getDomain and deleteDomain methods. The annotation for
 * getAllDomains would use the "allDomains" cache name specified in the method
 * level annotation.
 * <pre><code>
 * package my.app;
 *
 * @CacheDefaults(cacheName="domainCache")
 * public class DomainDao {
 *   @CacheResult
 *   public Domain getDomain(String domainId, int index) {
 *     ...
 *   }
 *
 *   @CacheRemove
 *   public void deleteDomain(String domainId, int index) {
 *     ...
 *   }
 *
 *   @CacheResult(cacheName="allDomains")
 *   public List<Domain> getAllDomains() {
 *     ...
 *   }
 * }
 * </code></pre>
 *
 * @author Rick Hightower
 * @since 1.0
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface CacheDefaults {

  /**
   * The default name of the cache for the annotated class
   * <p>
   * If not specified defaults to:
   * package.name.ClassName.methodName(package.ParameterType,package.ParameterType)
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, {@link CacheRemove},
```

```
   * and {@link CacheRemoveAll}
   */
  @Nonbinding String cacheName() default "";

  /**
   * The {@link CacheResolverFactory} used to find the {@link CacheResolver} to
   * use at runtime.
   * <p>
   * The default resolver pair will resolve the cache by name from the default
   * {@link CacheManager}
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, {@link CacheRemove},
   * and {@link CacheRemoveAll}
   */
  @Nonbinding Class<? extends CacheResolverFactory> cacheResolverFactory()
      default CacheResolverFactory.class;

  /**
   * The {@link CacheKeyGenerator} to use to generate the
   * {@link GeneratedCacheKey} for interacting with the specified Cache.
   * <p>
   * Defaults to a key generator that uses {@link Arrays#deepHashCode(Object[])}
   * and {@link Arrays#deepEquals(Object[], Object[])} with the array returned by
   * {@link CacheKeyInvocationContext#getKeyParameters()}
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, and {@link CacheRemove}
   *
   * @see CacheKey
   */
  @Nonbinding Class<? extends CacheKeyGenerator> cacheKeyGenerator()
      default CacheKeyGenerator.class;
}
```

11.1.2. `@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 the cache on later calls with the same parameters.

It is defined as follows:

```
/**
 * When a method annotated with {@link CacheResult} is invoked a
 * {@link GeneratedCacheKey} will be generated and {@link Cache#get(Object)} is
 * called before the annotated method actually executes. If a value is found in the
 * cache it is returned and the annotated method is never actually executed. If no
 * value is found the annotated method is invoked and the returned value is stored
 * in the cache with the generated key.
 * <p>
```

```
* Exceptions are not cached by default. Caching of exceptions can be enabled by
* specifying an {@link #exceptionCacheName()}. If an exception cache is specified
* it is checked before invoking the annotated method and if a cached exception is
* found it is re-thrown.
* <p>
* The {@link #cachedExceptions()} and {@link #nonCachedExceptions()} properties
* can be used to control the exceptions are cached and those that are not.
* <p>
* To always invoke the annotated method and still cache the result set
* {@link #skipGet()} to true. This will disable the pre-invocation
* {@link Cache#get(Object)} call. If {@link #exceptionCacheName()} is
* specified the pre-invocation exception check is also disabled. This feature is
* useful for methods that create or update objects to be cached.
* <p>
* Example of caching the Domain object with a key generated from the
* <code>String</code> and <code>int</code> parameters.
* <p>
* With no {@link #cacheName()} specified a cache name of
* "my.app.DomainDao.getDomain(java.lang.String,int)" will be generated.
* <p><pre><code>
* package my.app;
*
* public class DomainDao {
*   @CacheResult
*   public Domain getDomain(String domainId, int index) {
*     ...
*   }
* }
* </code></pre>
* <p>
* Example using the {@link GeneratedCacheKey} annotation so that only the domainId
* parameter is used in key generation:
* <pre><code>
* package my.app;
*
* public class DomainDao {
*   @CacheResult
*   public Domain getDomain(@CacheKey String domainId, Monitor mon) {
*     ...
*   }
* }
* </code></pre>
* <p>
* If exception caching is enabled via specification of
* {@link #exceptionCacheName()} the following rules are used to determine if a
* thrown exception is cached:
* <ol>
* <li>If {@link #cachedExceptions()} and {@link #nonCachedExceptions()} are both
* empty then all exceptions are cached</li>
* <li>If {@link #cachedExceptions()} is specified and
* {@link #nonCachedExceptions()} is not specified then only exceptions
* that pass an instanceof check against the cachedExceptions list are cached</li>
* <li>If {@link #nonCachedExceptions()} is specified and
* {@link #cachedExceptions()} is not specified then all exceptions
* that do not pass an instanceof check against the nonCachedExceptions list are
* cached</li>
* <li>If {@link #cachedExceptions()} and {@link #nonCachedExceptions()} are both
* specified then exceptions that pass an instanceof check against the
* cachedExceptions list but do not pass an instanceof check against the
```

```
 * nonCachedExceptions list are cached</li>
 * </ol>
 *
 * @author Eric Dalquist
 * @author Rick Hightower
 * @see CacheKey
 * @since 1.0
 */
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface CacheResult {

  /**
   * The name of the cache.
   * <p>
   * If not specified defaults first to {@link CacheDefaults#cacheName()} and if
   * that is not set it defaults to:
   * package.name.ClassName.methodName(package.ParameterType,package.ParameterType)
   */
  @Nonbinding String cacheName() default "";

  /**
   * If set to true the pre-invocation {@link Cache#get(Object)} is
   * skipped and the annotated method is always executed with the returned value
   * being cached as normal. This is useful for create or update methods that
   * should always be executed and have their returned value placed in the cache.
   * <p>
   * If true and an {@link #exceptionCacheName()} is specified the pre-invocation
   * check for a thrown exception is also skipped. If an exception is thrown during
   * invocation it will be cached following the standard exception caching rules.
   * <p>
   * Defaults to false.
   *
   * @see CachePut
   */
  @Nonbinding boolean skipGet() default false;

  /**
   * The {@link CacheResolverFactory} used to find the {@link CacheResolver} to
   * use at runtime.
   * <p>
   * The default resolver pair will resolve the cache by name from the default
   * {@link CacheManager}
   */
  @Nonbinding Class<? extends CacheResolverFactory> cacheResolverFactory()
      default CacheResolverFactory.class;

  /**
   * The {@link CacheKeyGenerator} to use to generate the {@link GeneratedCacheKey}
   * for interacting with the specified Cache.
   * <p>
   * Defaults to a key generator that uses
   * {@link java.util.Arrays#deepHashCode(Object[])} and
   * {@link java.util.Arrays#deepEquals(Object[], Object[])} with the array
   * returned by {@link CacheKeyInvocationContext#getKeyParameters()}
   *
   * @see CacheKey
   */
  @Nonbinding Class<? extends CacheKeyGenerator> cacheKeyGenerator()
```

```
      default CacheKeyGenerator.class;

  /**
   * The name of the cache to cache exceptions.
   * <p>
   * If not specified no exception caching is done.
   */
  @Nonbinding String exceptionCacheName() default "";

  /**
   * Defines zero (0) or more exception {@link Class classes}, that must be a
   * subclass of {@link Throwable}, indicating the exception types that
   * <b>must</b> be cached. Only consulted if {@link #exceptionCacheName()} is
   * specified.
   */
  @Nonbinding Class<? extends Throwable>[] cachedExceptions() default {};

  /**
   * Defines zero (0) or more exception {@link Class Classes}, that must be a
   * subclass of {@link Throwable}, indicating the exception types that
   * <b>must not</b> be cached. Only consulted if {@link #exceptionCacheName()}
   * is specified.
   */
  @Nonbinding Class<? extends Throwable>[] nonCachedExceptions() default {};
}
```

The `@CacheKey` 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. `skipGet`. 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.

11.1.3. `@CachePut`

This is a method level annotation used to mark methods where one of the method parameters should be stored in the cache. One parameter must be annotated with `@CacheValue` marking it as 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.

It is defined as follows:

```
/**
 * Allows the configuration of defaults for {@link CacheResult}, {@link CachePut},
 * {@link CacheRemove}, and {@link CacheRemoveAll} at the class level. Without
 * the method level annotations this annotation has no effect.
 * <p>
 * Following is an example of specifying a default cache name that is used by
```

```
 * the annotations on the getDomain and deleteDomain methods. The annotation for
 * getAllDomains would use the "allDomains" cache name specified in the method
 * level annotation.
 * <pre><code>
 * package my.app;
 *
 * @CacheDefaults(cacheName="domainCache")
 * public class DomainDao {
 *   @CacheResult
 *   public Domain getDomain(String domainId, int index) {
 *     ...
 *   }
 *
 *   @CacheRemove
 *   public void deleteDomain(String domainId, int index) {
 *     ...
 *   }
 *
 *   @CacheResult(cacheName="allDomains")
 *   public List<Domain> getAllDomains() {
 *     ...
 *   }
 * }
 * </code></pre>
 *
 * @author Rick Hightower
 * @since 1.0
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface CacheDefaults {

  /**
   * The default name of the cache for the annotated class
   * <p>
   * If not specified defaults to:
   * package.name.ClassName.methodName(package.ParameterType,package.ParameterType)
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, {@link CacheRemove},
   * and {@link CacheRemoveAll}
   */
  @Nonbinding String cacheName() default "";

  /**
   * The {@link CacheResolverFactory} used to find the {@link CacheResolver} to
   * use at runtime.
   * <p>
   * The default resolver pair will resolve the cache by name from the default
   * {@link CacheManager}
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, {@link CacheRemove},
```

```
   * and {@link CacheRemoveAll}
   */
  @Nonbinding Class<? extends CacheResolverFactory> cacheResolverFactory()
      default CacheResolverFactory.class;

  /**
   * The {@link CacheKeyGenerator} to use to generate the
   * {@link GeneratedCacheKey} for interacting with the specified Cache.
   * <p>
   * Defaults to a key generator that uses {@link Arrays#deepHashCode(Object[])}
   * and {@link Arrays#deepEquals(Object[], Object[])} with the array returned by
   * {@link CacheKeyInvocationContext#getKeyParameters()}
   * <p>
   * Applicable for {@link CacheResult}, {@link CachePut}, and {@link CacheRemove}
   *
   * @see CacheKey
   */
  @Nonbinding Class<? extends CacheKeyGenerator> cacheKeyGenerator()
      default CacheKeyGenerator.class;
}
```

The `@CacheKey` 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.

11.1.4. `@CacheRemove`

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

It is defined as follows:

```
/**
 * When a method annotated with {@link CacheRemove} is invoked a {@link
 * GeneratedCacheKey} will be generated and
 * {@link Cache#remove(Object)} will be invoked on the specified
 * cache.
 * <p>
 * The default behavior is to call {@link Cache#remove(Object)} after
 * the annotated method is invoked, this behavior can be changed by setting
 * {@link #afterInvocation()} to false in which case
 * {@link Cache#remove(Object)} will be called before the annotated
 * method is invoked.
 * <p>
 * Example of removing a specific Domain object from the "domainCache". A {@link
```

```
 * GeneratedCacheKey} will be generated from the String and int parameters and
 * used to call {@link Cache#remove(Object)} after the deleteDomain
 * method completes successfully.
 * <pre><code>
 * package my.app;
 *
 * public class DomainDao {
 *   @CacheRemove(cacheName="domainCache")
 *   public void deleteDomain(String domainId, int index) {
 *     ...
 *   }
 * }
 * </code></pre>
 * <p>
 * Exception Handling, only used if {@link #afterInvocation()} is true.
 * <ol>
 * <li>If {@link #evictFor()} and {@link #noEvictFor()} are both empty then all
 * exceptions prevent the remove</li>
 * <li>If {@link #evictFor()} is specified and {@link #noEvictFor()} is not
 * specified then only exceptions that pass an instanceof check against the
 * evictFor list result in a
 * remove</li>
 * <li>If {@link #noEvictFor()} is specified and {@link #evictFor()} is not
 * specified then all exceptions that do not pass an instanceof check against the
 * noEvictFor result in a
 * remove</li>
 * <li>If {@link #evictFor()} and {@link #noEvictFor()} are both specified then
 * exceptions that pass an instanceof check against the evictFor list but do not
 * pass an instanceof check against the noEvictFor list result in a remove</li>
 * </ol>
 *
 * @author Eric Dalquist
 * @author Rick Hightower
 * @author Greg Luck
 * @see CacheKey
 * @since 1.0
 */
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface CacheRemove {

  /**
   * The name of the cache.
   * <p>
   * If not specified defaults first to {@link CacheDefaults#cacheName()},
   * and if that is not set then to:
   * package.name.ClassName.methodName(package.ParameterType,package.ParameterType)
   */
  @Nonbinding String cacheName() default "";

  /**
   * When {@link Cache#remove(Object)}  should be called. If true it is called
   * after the annotated method invocation completes successfully. If false it is
   * called before the annotated method is invoked.
   * <p>
   * Defaults to true.
   * <p>
   * If true and the annotated method throws an exception the put will not be
   * executed.
```

```
   */
  @Nonbinding boolean afterInvocation() default true;

  /**
   * The {@link CacheResolverFactory} used to find the {@link CacheResolver} to
   * use at runtime.
   * <p>
   * The default resolver pair will resolve the cache by name from the default
   * {@link CacheManager}
   */
  @Nonbinding Class<? extends CacheResolverFactory> cacheResolverFactory()
      default CacheResolverFactory.class;

  /**
   * The {@link CacheKeyGenerator} to use to generate the {@link
   * GeneratedCacheKey} for interacting with the specified Cache.
   * <p>
   * Defaults to a key generator that uses
   * {@link java.util.Arrays#deepHashCode(Object[])}
   * and {@link java.util.Arrays#deepEquals(Object[], Object[])} with the array
   * returned by {@link CacheKeyInvocationContext#getKeyParameters()}
   *
   * @see CacheKey
   */
  @Nonbinding Class<? extends CacheKeyGenerator> cacheKeyGenerator()
      default CacheKeyGenerator.class;

  /**
   * Defines zero (0) or more exception {@link Class classes}, that must be a
   * subclass of {@link Throwable}, indicating the exception types that must cause
   * a cache eviction. Only used if {@link #afterInvocation()} is true.
   */
  @Nonbinding Class<? extends Throwable>[] evictFor() default {};

  /**
   * Defines zero (0) or more exception {@link Class Classes}, that must be a
   * subclass of {@link Throwable}, indicating the exception types that must
   * <b>not</b> cause a cache eviction. Only used if {@link #afterInvocation()} is
   * true.
   */
  @Nonbinding Class<? extends Throwable>[] noEvictFor() default {};
}
```

The `@CacheKey` 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.

`@CacheRemove` will be ignored if placed on static methods.

11.1.5. `@CacheRemoveAll`

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

It is defined as follows:

```
/**
 * When a method annotated with {@link CacheRemoveAll} is invoked all elements in
 * the specified cache will be removed via the
 * {@link Cache#removeAll()} method
 * <p>
 * The default behavior is to call {@link Cache#removeAll()} after the
 * annotated method is invoked, this behavior can be changed by setting {@link
 * #afterInvocation()} to false in which case {@link Cache#removeAll()}
 * will be called before the annotated method is invoked.
 * <p>
 * Example of removing all Domain objects from the "domainCache". {@link
 * Cache#removeAll()} will be called after deleteAllDomains() returns
 * successfully.
 * <pre><code>
 * package my.app;
 *
 * public class DomainDao {
 *   @CacheRemoveAll(cacheName="domainCache")
 *   public void deleteAllDomains() {
 *     ...
 *   }
 * }
 * </code></pre>
 * <p>
 * Exception Handling, only used if {@link #afterInvocation()} is true.
 * <ol>
 * <li>If {@link #evictFor()} and {@link #noEvictFor()} are both empty then all
 * exceptions prevent the removeAll</li>
 * <li>If {@link #evictFor()} is specified and {@link #noEvictFor()} is not
 * specified then only exceptions that pass an instanceof check against the
 * evictFor list result in a removeAll</li>
 * <li>If {@link #noEvictFor()} is specified and {@link #evictFor()} is not
 * specified then all exceptions that do not pass an instanceof check against the
 * noEvictFor result in a removeAll</li>
 * <li>If {@link #evictFor()} and {@link #noEvictFor()} are both specified then
 * exceptions that pass an instanceof check against the evictFor list but do not
 * pass an instanceof check against the noEvictFor list result in a removeAll</li>
 * </ol>
 *
 * @author Eric Dalquist
 * @author Rick Hightower
 * @since 1.0
 */
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface CacheRemoveAll {

  /**
```

```
    * /**
    * The name of the cache.
    * <p>
    * If not specified defaults first to {@link CacheDefaults#cacheName()} and if
    * that is not set it defaults to:
    * package.name.ClassName.methodName(package.ParameterType,package.ParameterType)
    */
   @Nonbinding String cacheName() default "";

   /**
    * When {@link Cache#removeAll()} should be called. If true it is called after
    * the annotated method invocation completes successfully. If false it is called
    * before the annotated method is invoked.
    * <p>
    * Defaults to true.
    * <p>
    * If true and the annotated method throws an exception the put will not be
    * executed.
    */
   @Nonbinding boolean afterInvocation() default true;

   /**
    * The {@link CacheResolverFactory} used to find the {@link CacheResolver} to
    * use at runtime.
    * <p>
    * The default resolver pair will resolve the cache by name from the default
    * {@link CacheManager}
    */
   @Nonbinding Class<? extends CacheResolverFactory> cacheResolverFactory()
       default CacheResolverFactory.class;

   /**
    * Defines zero (0) or more exception {@link Class classes}, that must be a
    * subclass of {@link Throwable}, indicating the exception types that must
    * cause a cache eviction. Only used if {@link #afterInvocation()} is true.
    */
   @Nonbinding Class<? extends Throwable>[] evictFor() default {};

   /**
    * Defines zero (0) or more exception {@link Class Classes}, that must be a
    * subclass of {@link Throwable}, indicating the exception types that must
    * <b>not</b> cause a cache eviction. Only used if {@link #afterInvocation()} is
    * true.
    */
   @Nonbinding Class<? extends Throwable>[] noEvictFor() default {};
}
```

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.

11.1.6. `@CacheKey`

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

Usable with:

- `@CacheResult`,
- `@CachePut`, and
- `@CacheRemove`

11.1.7. `@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`

11.1.8. Example 2

This example shows usage of many of the above annotation.

```
/**
 * An implementation of BlogManager that uses a variety of annotations
 * @author Rick Hightower
 */
@CacheDefaults(cacheName = "blgMngr")
public class ClassLevelCacheConfigBlogManagerImpl implements BlogManager {

  private static Map<String, Blog> map = new HashMap<String, Blog>();

  @CacheResult
  public Blog getEntryCached(String title) {
    return map.get(title);
  }

  public Blog getEntryRaw(String title) {
    return map.get(title);
  }
```

```
  /**
   * @see manager.BlogManager#clearEntryFromCache(java.lang.String)
   */
  @CacheRemove
  public void clearEntryFromCache(String title) {
  }

  public void clearEntry(String title) {
    map.put(title, null);
  }

  @CacheRemoveAll
  public void clearCache() {
  }

  public void createEntry(Blog blog) {
    map.put(blog.getTitle(), blog);
  }

  @CacheResult
  public Blog getEntryCached(String randomArg, @CacheKey String title,
                             String randomArg2) {
    return map.get(title);
  }
```

11.2. Cache Resolution

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

It is defined below:

```
/**
 * Determines the {@link CacheResolver} to use for an annotated method. The
 * {@link CacheResolver} will be retrieved once per annotated method.
 * <p/>
 * Implementations MUST be thread-safe.
 *
 * @author Eric Dalquist
 * @since 1.0
 */
public interface CacheResolverFactory {

  /**
   * Get the {@link CacheResolver} used at runtime for resolution of the
   * {@link Cache} for the {@link CacheResult}, {@link CachePut},
   * {@link CacheRemove}, or {@link CacheRemoveAll} annotation.
   *
   * @param cacheMethodDetails The details of the annotated method to get the
   *                           {@link CacheResolver} for. @return The {@link
   *                           CacheResolver} instance to be
   *                           used by the interceptor.
   */
  CacheResolver getCacheResolver(CacheMethodDetails<? extends Annotation>
```

```
                                        cacheMethodDetails);

  /**
   * Get the {@link CacheResolver} used at runtime for resolution of the {@link
   * Cache} for the {@link CacheResult} annotation to cache exceptions.
   * <p/>
   * Will only be called if {@link CacheResult#exceptionCacheName()} is not empty.
   *
   * @param cacheMethodDetails The details of the annotated method to get the
   *                           {@link CacheResolver} for.
   * @return The {@link CacheResolver} instance to be used by the interceptor.
   */
  CacheResolver getExceptionCacheResolver(CacheMethodDetails<CacheResult>
                                                    cacheMethodDetails);
}
```

11.2.1. Cache Name

If no cache name is specified either on the method level annotation or at the class level with the `@CacheDefaults` annotation then the name is generated as the following:

```
package.name.ClassName.methodName(package.ParameterType,package.ParameterTy
pe)
```

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.

11.2.2. 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.

The default `CacheResolverFactory` does the following, in order:

1. Get the `CacheManager` to use via:

   ```
   CachingProvider provider = Caching.getCachingProvider();
   CacheManager cacheManager =
   provider.getCacheManager(provider.getDefaultURI(),
   provider.getDefaultClassLoader());
   ```

2. Call `CacheManager.getCache(String cacheName)` with the cache name

3. If a `Cache` is not returned, a default cache is created using:

```
Cache cache = cacheManager.createCache(cacheName, new
MutableConfiguration());
```

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`.

11.2.3. 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.

It is defined as follows:

```
/**
 * Determines the {@link Cache} to use for an intercepted method invocation.
 * <p/>
 * Implementations MUST be thread-safe.
 *
 * @author Eric Dalquist
 * @see CacheResolverFactory
 * @since 1.0
 */
public interface CacheResolver {

  /**
   * Resolve the {@link Cache} to use for the {@link CacheInvocationContext}.
   *
   * @param cacheInvocationContext The context data for the intercepted method
   *                               invocation
   * @return The {@link Cache} instance to be used by the intercepter
   */
  <K, V> Cache<K, V> resolveCache(CacheInvocationContext<? extends Annotation>
                                    cacheInvocationContext);

}
```

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

11.3. Key Generation

The `@CacheResult`, `@CachePut`, and `@CacheRemove` 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 `GeneratedCacheKey`.

If `javax.cache.annotation.CacheKeyGenerator` is not 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`.

11.4. Annotation Support Classes

11.4.1. `CacheMethodDetails`

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

It is defined as follows:

```
/**
 * Static information about a method annotated with one of:
 * {@link CacheResult}, {@link CachePut}, {@link CacheRemove}, or {@link
 * CacheRemoveAll}
 * <p/>
 * Used with {@link CacheResolverFactory#getCacheResolver(CacheMethodDetails)} to
 * determine the {@link CacheResolver} to use with the method.
 *
 * @param <A> The type of annotation this context information is for. One of
 *            {@link CacheResult}, {@link CachePut}, {@link CacheRemove}, or
 *            {@link CacheRemoveAll}.
 * @author Eric Dalquist
 * @see CacheResolverFactory
 */
public interface CacheMethodDetails<A extends Annotation> {
  /**
   * The annotated method
   *
   * @return The annotated method
   */
  Method getMethod();

  /**
   * An immutable Set of all Annotations on this method
   *
```

```
   * @return An immutable Set of all Annotations on this method
   */
  Set<Annotation> getAnnotations();

  /**
   * The caching related annotation on the method.
   * One of: {@link CacheResult}, {@link CachePut}, {@link CacheRemove}, or
   * {@link CacheRemoveAll}
   *
   * @return The caching related annotation on the method.
   */
  A getCacheAnnotation();

  /**
   * The cache name resolved by the implementation.
   * <p/>
   * The cache name is determined by first looking at the cacheName attribute of
   * the method level annotation. If that attribute is not set then the class
   * level {@link CacheDefaults} annotation is checked. If that annotation does
   * not exist or does not have its cacheName attribute set then the following
   * cache name generation rules are followed:
   * <p/>
   * "fully qualified class name"."method name"("fully qualified parameter class
   * names")
   * <p/>
   * For example:
   * <p><blockquote><pre>
   * package my.app;
   * <p/>
   * public class DomainDao {
   *   @CacheResult
   *   public Domain getDomain(String domainId, int index) {
   *     ...
   *   }
   * }
   * </pre></blockquote></p>
   * Results in the cache name: "my.app.DomainDao.getDomain(java.lang.String,int)"
   *
   * @return The fully resolved cache name
   */
  String getCacheName();
}
```

11.4.2. `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.

It is defined as follows:

```
/**
 * Runtime information about an intercepted method invocation for a method
 * annotated with {@link CacheResult}, {@link CachePut}, {@link CacheRemove},
 * or {@link CacheRemoveAll}
 * <p>
 * Used with {@link CacheResolver#resolveCache(CacheInvocationContext)} to
 * determine the {@link javax.cache.Cache} to use at runtime for the method
 * invocation.
 *
 * @param <A> The type of annotation this context information is for. One of
 *            {@link CacheResult}, {@link CachePut}, {@link CacheRemove}, or {@link
 *            CacheRemoveAll}.
 * @author Eric Dalquist
 * @see CacheResolver
 */
public interface CacheInvocationContext<A extends Annotation>
    extends CacheMethodDetails<A> {

  /**
   * @return The object the intercepted method was invoked on.
   */
  Object getTarget();

  /**
   * Returns a clone of the array of all method parameters.
   *
   * @return An array of all parameters for the annotated method
   */
  CacheInvocationParameter[] getAllParameters();

  /**
   * Return an object of the specified type to allow access to the
   * provider-specific API. If the provider's
   * implementation does not support the specified class, the {@link
   * IllegalArgumentException} is thrown.
   *
   * @param cls the class of the object to be returned. This is normally either the
   *            underlying implementation class or an interface that it implements.
   * @return an instance of the specified class
   * @throws IllegalArgumentException if the provider doesn't support the specified
   *            class.
   */
  <T> T unwrap(java.lang.Class<T> cls);
}
```

11.4.3. `CacheKeyInvocationContext`

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

It is defined as follows:

```
/**
 * Runtime information about an intercepted method invocation for a method
 * annotated with {@link CacheResult}, {@link CachePut}, or
 * {@link CacheRemove}.
 * <p/>
 * Used with {@link CacheKeyGenerator#generateCacheKey(CacheKeyInvocationContext)}
 * to generate a {@link GeneratedCacheKey} for the invocation.
 *
 * @param <A> The type of annotation this context information is for. One of
 *            {@link CacheResult}, {@link CachePut}, or {@link CacheRemove}.
 * @author Eric Dalquist
 * @see CacheKeyGenerator
 */
public interface CacheKeyInvocationContext<A extends Annotation>
    extends CacheInvocationContext<A> {

  /**
   * Returns a clone of the array of all method parameters to be used by the
   * {@link
   * CacheKeyGenerator} in creating a {@link GeneratedCacheKey}. The returned array
   * may be the same as or a subset of the array returned by
   * {@link #getAllParameters()}
   * <p/>
   * Parameters in this array are selected by the following rules:
   * <ul>
   * <li>If no parameters are annotated with {@link CacheKey} or {@link
   * CacheValue}
   * then all parameters are included</li>
   * <li>If a {@link CacheValue} annotation exists and no {@link CacheKey} then
   * all
   * parameters except the one annotated with {@link CacheValue} are included</li>
   * <li>If one or more {@link CacheKey} annotations exist only those parameters
   * with the {@link CacheKey} annotation are included</li>
   * </ul>
   *
   * @return An array of all parameters to be used in cache key generation
   */
  CacheInvocationParameter[] getKeyParameters();

  /**
   * When a method is annotated with {@link CachePut} this is the parameter
   * annotated with {@link CacheValue}
```

```
   *
   * @return The parameter to cache, will never be null for methods annotated with
   *         {@link CachePut}, will be null for methods not annotated with {@link
   *         CachePut}
   */
  CacheInvocationParameter getValueParameter();
}
```

11.4.4. `CacheInvocationParameter`

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

It is defined as follows:

```
/**
 * A parameter to an intercepted method invocation. Contains the parameter value
 * as well static type and annotation information about the parameter.
 *
 * @author Eric Dalquist
 */
public interface CacheInvocationParameter {

  /**
   * The parameter type as declared on the method.
   */
  Class<?> getRawType();

  /**
   * @return The parameter value
   */
  Object getValue();

  /**
   * @return An immutable Set of all Annotations on this method parameter, never
   * null.
   */
  Set<Annotation> getAnnotations();

  /**
   * The index of the parameter in the original parameter array as returned by
   * {@link CacheInvocationContext#getAllParameters()}
   *
   * @return The index of the parameter in the original parameter array.
   */
  int getParameterPosition();
}
```

11.4.5. GeneratedCacheKey

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

It is defined as follows:

```
/**
 * A {@link Serializable}, immutable, thread-safe object that is used as a key,
 * automatically generated by a {@link CacheKeyGenerator}.
 * <p>
 * The implementation MUST follow the Java contract for {@link Object#hashCode()}
 * and {@link Object#equals(Object)} to ensure correct behavior.
 * <p>
 * It is recommended that implementations also override {@link Object#toString()}
 * and provide a human-readable string representation of the key.
 *
 * @author Eric Dalquist
 * @see CacheKeyGenerator
 * @since 1.0
 */
public interface GeneratedCacheKey extends Serializable {

  /**
   * The immutable hash code of the cache key.
   *
   * @return The hash code of the object
   * @see Object#hashCode()
   */
  @Override
  int hashCode();

  /**
   * Compare this {@link GeneratedCacheKey} with another. If the two objects
   * are equal their {@link #hashCode()} values MUST be equal as well.
   *
   * @param object The other object to compare to.
   * @return true if the objects are equal
   * @see Object#equals(Object)
   */
  @Override
  boolean equals(Object object);
}
```

11.5. Annotations Interactions

11.5.1. Annotation Inheritance and Ordering

This specification defers to section 2.1 of the Common Annotations for Java specification[2] 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.

11.5.2. 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.

12. Management

The javax.cache.management package contains `MXBean` interfaces for cache management and statistics.

12.1. Enabling and Disabling

By default, both management and statistics are disabled. To turn them on at configuration time, use the following methods on `MutableConfiguration`:

- `setManagementEnabled(boolean enabled)` to turn on management
- `setStatisticsEnabled(boolean enabled)` to turn on statistics

To enable or disable them at runtime, the following methods are provided on `CacheManager`:

```
/**
 * Controls whether management is enabled. If enabled the
 * {@link javax.cache.management.CacheMXBean} for each cache is registered in
 * the platform MBean server. THe platform MBeanServer is obtained using
 * {@link java.lang.management.ManagementFactory#getPlatformMBeanServer()}
 * <p/>
 * Management information includes the name and configuration information for
 * the cache.
 * <p/>
 * Each cache's management object must be registered with an ObjectName that
 * is unique and has the following type and attributes:
 * <p/>
 * Type:
 * <code>javax.cache:type=Cache</code>
 * <p/>
 * Required Attributes:
 * <ul>
 * <li>CacheManager the name of the CacheManager
 * <li>Cache the name of the Cache
 * </ul>
 *
 * @param cacheName the name of the cache to register
 * @param enabled   true to enable management, false to disable.
 */
void enableManagement(String cacheName, boolean enabled);


/**
 * Enables or disables statistics gathering for a managed {@link Cache} at
 * runtime.
 * <p/>
 * Each cache's statistics object must be registered with an ObjectName that
 * is unique and has the following type and attributes:
 * <p/>
```

```
   * Type:
   * <code>javax.cache:type=CacheStatistics</code>
   * <p/>
   * Required Attributes:
   * <ul>
   * <li>CacheManager the name of the CacheManager
   * <li>Cache the name of the Cache
   * </ul>
   *
   * @param cacheName the name of the cache to register
   * @param enabled   true to enable statistics, false to disable.
   * @throws IllegalStateException if the cache is {@link #isClosed()}
   * @throws NullPointerException  if cacheName is null
   */
  void enableStatistics(String cacheName, boolean enabled);
```

12.2. MXBean Definitions

The CacheMXBean provides details of cache configuration and is defined as follows:

```
/**
 * A management bean for cache. It provides configuration information. It does not
 * allow mutation of configuration or mutation of the cache.
 * <p>
 * Each cache's management object must be registered with an ObjectName that is
 * unique and has the following type and attributes:
 * <p>
 * Type:
 * <code>javax.cache:type=CacheConfiguration</code>
 * <p>
 * Required Attributes:
 * <ul>
 * <li>CacheManager the URI of the CacheManager
 * <li>Cache the name of the Cache
 * </ul>
 *
 * @author Greg Luck
 * @author Yannis Cosmadopoulos
 * @since 1.0
 */
@MXBean
public interface CacheMXBean {

  /**
   * Determines the required type of keys for this {@link Cache}, if any.
   *
   * @return the fully qualified class name of the key type,
   * or "java.lang.Object" if the type is undefined.
   */
  String getKeyType();

  /**
```

```
 * Determines the required type of values for this {@link Cache}, if any.
 * @return the fully qualified class name of the value type,
 * or "java.lang.Object" if the type is undefined.
 */
String getValueType();

/**
 * Determines if a {@link Cache} should operate in read-through mode.
 * <p>
 * When in read-through mode, cache misses that occur due to cache entries
 * not existing as a result of performing a "get" call via one of
 * {@link Cache#get},
 * {@link Cache#getAll},
 * {@link Cache#getAndRemove} and/or
 * {@link Cache#getAndReplace} will appropriately
 * cause the configured {@link CacheLoader} to be
 * invoked.
 * <p>
 * The default value is <code>false</code>.
 *
 * @return <code>true</code> when a {@link Cache} is in
 *         "read-through" mode.
 * @see CacheLoader
 */
boolean isReadThrough();

/**
 * Determines if a {@link Cache} should operate in "write-through"
 * mode.
 * <p>
 * When in "write-through" mode, cache updates that occur as a result of
 * performing "put" operations called via one of
 * {@link Cache#put},
 * {@link Cache#getAndRemove},
 * {@link Cache#removeAll},
 * {@link Cache#getAndPut}
 * {@link Cache#getAndRemove},
 * {@link Cache#getAndReplace},
 * {@link Cache#invoke}
 * {@link Cache#invokeAll}
 * <p>
 * will appropriately cause the configured {@link CacheWriter} to be invoked.
 * <p>
 * The default value is <code>false</code>.
 *
 * @return <code>true</code> when a {@link Cache} is in "write-through" mode.
 * @see CacheWriter
 */
boolean isWriteThrough();

/**
```

```
    * Whether storeByValue (true) or storeByReference (false).
    * When true, both keys and values are stored by value.
    * <p>
    * When false, both keys and values are stored by reference.
    * Caches stored by reference are capable of mutation by any threads holding
    * the reference. The effects are:
    * <ul>
    * <li>if the key is mutated, then the key may not be retrievable or
    * removable</li>
    * <li>if the value is mutated, then all threads in the JVM can potentially
    * observe those mutations, subject to the normal Java Memory Model rules.</li>
    * </ul>
    * Storage by reference only applies to the local heap. If an entry is moved off
    * heap it will need to be transformed into a representation. Any mutations that
    * occur after transformation may not be reflected in the cache.
    * <p>
    * When a cache is storeByValue, any mutation to the key or value does not affect
    * the key of value stored in the cache.
    * <p>
    * The default value is <code>true</code>.
    *
    * @return true if the cache is store by value
    */
   boolean isStoreByValue();

   /**
    * Checks whether statistics collection is enabled in this cache.
    * <p>
    * The default value is <code>false</code>.
    *
    * @return true if statistics collection is enabled
    */
   boolean isStatisticsEnabled();

   /**
    * Checks whether management is enabled on this cache.
    * <p>
    * The default value is <code>false</code>.
    *
    * @return true if management is enabled
    */
   boolean isManagementEnabled();

}
```

The CacheStatisticsMXBean provides statistics for a cache, and is defined as follows:

```
/**
 * Cache statistics.
```

```
 * <p>
 * Statistics are accumulated from the time a cache is created. They can be reset
 * to zero using {@link #clear}.
 * <p>
 * There are no defined consistency semantics for statistics. Refer to the
 * implementation for precise semantics.
 * <p>
 * Each cache's statistics object must be registered with an ObjectName that is
 * unique and has the following type and attributes:
 * <p>
 * Type:
 * <code>javax.cache:type=CacheStatistics</code>
 * <p>
 * Required Attributes:
 * <ul>
 * <li>CacheManager the URI of the CacheManager
 * <li>Cache the name of the Cache
 * </ul>
 *
 * @author Greg Luck
 * @since 1.0
 */
@MXBean
public interface CacheStatisticsMXBean {

  /**
   * Clears the statistics counters to 0 for the associated Cache.
   */
  void clear();

  /**
   * The number of get requests that were satisfied by the cache.
   * <p>
   * {@link javax.cache.Cache#containsKey(Object)} is not a get request for
   * statistics purposes.
   * <p>
   * In a caches with multiple tiered storage, a hit may be implemented as a hit
   * to the cache or to the first tier.
   * <p>
   * For an {@link javax.cache.processor.EntryProcessor}, a hit occurs when the
   * key exists and an entry processor can be invoked against it, even if no
   * methods of {@link javax.cache.Cache.Entry} or
   * {@link javax.cache.processor.MutableEntry} are called.
   *
   * @return the number of hits
   */
  long getCacheHits();

  /**
   * This is a measure of cache efficiency.
   * <p>
```

```
     * It is calculated as:
     * {@link #getCacheHits} divided by {@link #getCacheGets ()} * 100.
     *
     * @return the percentage of successful hits, as a decimal e.g 75.
     */
    float getCacheHitPercentage();

    /**
     * A miss is a get request that is not satisfied.
     * <p>
     * In a simple cache a miss occurs when the cache does not satisfy the request.
     * <p>
     * {@link javax.cache.Cache#containsKey(Object)} is not a get request for
     * statistics purposes.
     * <p>
     * For an {@link javax.cache.processor.EntryProcessor}, a miss occurs when the
     * key does not exist and therefore an entry processor cannot be invoked
     * against it.
     * <p>
     * In a caches with multiple tiered storage, a miss may be implemented as a miss
     * to the cache or to the first tier.
     * <p>
     * In a read-through cache a miss is an absence of the key in the cache that
     * will trigger a call to a CacheLoader. So it is still a miss even though the
     * cache will load and return the value.
     * <p>
     * Refer to the implementation for precise semantics.
     *
     * @return the number of misses
     */
    long getCacheMisses();

    /**
     * Returns the percentage of cache accesses that did not find a requested entry
     * in the cache.
     * <p>
     * This is calculated as {@link #getCacheMisses()} divided by
     * {@link #getCacheGets()} * 100.
     *
     * @return the percentage of accesses that failed to find anything
     */
    float getCacheMissPercentage();

    /**
     * The total number of requests to the cache. This will be equal to the sum of
     * the hits and misses.
     * <p>
     * A "get" is an operation that returns the current or previous value. It does
     * not include checking for the existence of a key.
     * <p>
     * In a caches with multiple tiered storage, a gets may be implemented as a get
```

```
 * to the cache or to the first tier.
 *
 * @return the number of gets
 */
long getCacheGets();

/**
 * The total number of puts to the cache.
 * <p>
 * A put is counted even if it is immediately evicted.
 * <p>
 * Replaces, where a put occurs which overrides an existing mapping is counted
 * as a put.
 *
 * @return the number of puts
 */
long getCachePuts();

/**
 * The total number of removals from the cache. This does not include evictions,
 * where the cache itself initiates the removal to make space.
 *
 * @return the number of removals
 */
long getCacheRemovals();

/**
 * The total number of evictions from the cache. An eviction is a removal
 * initiated by the cache itself to free up space. An eviction is not treated as
 * a removal and does not appear in the removal counts.
 *
 * @return the number of evictions
 */
long getCacheEvictions();

/**
 * The mean time to execute gets.
 * <p>
 * In a read-through cache the time taken to load an entry on miss is not
 * included in get time.
 *
 * @return the time in µs
 */
float getAverageGetTime();

/**
 * The mean time to execute puts.
 *
 * @return the time in µs
 */
float getAveragePutTime();
```

```
  /**
   * The mean time to execute removes.
   *
   * @return the time in µs
   */
  float getAverageRemoveTime();

}
```

12.3. Accessing Management Information

There are no accessor methods provided for either management or statistics. When enabled the `MXBeans` are registered with an implementation specific `MBeanServer` by the caching implementation.

The beans can then be obtained from the same `MBeanServer` in the usual way as defined by JMX.

Simple in-process implementations might simply use the platform MBeanServer, accessed using `ManagementFactory.getPlatformMBeanServer()`. *The caching implementation must document how to resolve the* `MBeanServer` *that was used to store cache management information.*

The convention for JMX attribute names follows the JavaBeans[15] convention for properties. So, the accessor `getCacheHitPercentage()` on `CacheStatisticsMXBean` corresponds to the JMX attribute CacheHitPercentage.

12.3.1. Example 1

This example shows how to read the CacheHitPercentage for the cache named “simpleCache”.

```
    CachingProvider cachingProvider = Caching.getCachingProvider();
    CacheManager cacheManager = cachingProvider.getCacheManager();

    MutableConfiguration<String, Integer> config =
        new MutableConfiguration<String, Integer>();
    config.setTypes(String.class, Integer.class)
        .setExpiryPolicyFactory(AccessedExpiryPolicy.factoryOf(ONE_HOUR))
        .setStatisticsEnabled(true);

    cacheManager.createCache("simpleCache", config);
    Cache<String, Integer> cache = cacheManager.getCache("simpleCache",
        String.class, Integer.class);

    Set<ObjectName> registeredObjectNames = null;
    MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();

    ObjectName objectName = new ObjectName("javax.cache:type=CacheStatistics"
        + ",CacheManager=" + (cache.getCacheManager().getURI().toString())
        + ",Cache=" + cache.getName());
```

```
    System.out.println(mBeanServer.getAttribute(objectName,
        "CacheHitPercentage"));
  }
```

12.4. Statistics Effects of Cache Operations

The following table outlines the cache operations affect the statistics counters. In the table some if a hit will occur if a mapping exists, and a miss if one does not the table will have yes in each column. If a cache is set to read-through mode, the lack of a mapping will cause a miss, even if a CacheLoader loads an entry and the cache operation returns it from the call.

Method	Puts	Removals	Hits	Misses
`boolean containsKey(K key)`	No	No	No	No
`V get(K key)`	No	No	Yes	Yes
`Map<K,V> getAll(Collection<? extends K> keys)`	No	No	Yes	Yes
`V getAndPut(K key, V value)`	Yes	No	Yes	Yes
`V getAndRemove(K key)`	No	Yes	Yes	Yes
`V getAndReplace(K key, V value)`	Yes	No	Yes	Yes
`<T> T invoke(K key, EntryProcessor<K, V, T> entryProcessor, Object... arguments)`	Yes, if setValue(V value) was called.	Yes, if remove() was called.	Yes	Yes
`<T> Map<K, EntryProcessorResult<T>> invokeAll(Set<? extends K> keys, EntryProcessor<K, V, T> entryProcessor, Object... arguments);`	Yes, if setValue(V value) was called.	Yes, if remove() was called.	Yes	Yes
`Iterator<Cache.Entry<K, V>> iterator() ?`	No	Yes, if remove() was called.	Yes	No
`void loadAll(Set<? extends K> keys, boolean replaceExistingValues, CompletionListener completionListener)`	No	No	No	No
`void put(K key, V value)`	Yes	No	No	No

`void putAll(Map<? extends K,? extends V> map)`	Yes	No	No	No
`boolean putIfAbsent(K key, V value)`	Yes	No	Yes	Yes
`boolean remove(K key)`	No	Yes, , if the method returns true	No	No
`boolean remove(K key, V oldValue)`	No	Yes, if the method returns true	Yes	Yes
`void removeAll()`	No	One removal per entry that is removed	No	No
`void removeAll(Set<? extends K> keys)`	No	One removal per entry that is removed	No	No
`boolean replace(K key, V value)`	Yes	No	Yes	Yes
`boolean replace(K key, V oldValue, V newValue)`	Yes	No	Yes	Yes

13. Portability Recommendations

The following recommendations should be followed to improve application portability between implementations of the Java Cache API, it is recommended that:

1. Custom Key classes correctly override and implement the `Object.hashcode()` and `Object.equals()` methods. Custom Value classes correctly override and implement `Object.equals()` method.

   Although recommended, implementations are not required to call either the `Object.hashCode` or `Object.equals` methods defined by custom classes. Implementations are free to use optimizations whereby the invocation of these methods is avoided.

   As this specification does not define the concept of object equivalence it should be noted that applications making use of custom classes and relying on implementation specific optimizations to determine equivalence may not be portable.

2. to support the default store-by-value `Cache` semantics, Custom Key and Value classes are serializable.

3. Caches do not use forward slashes (/) or colons (:) as part of their names. Additionally it is recommended that cache names starting with `java.` or `javax.` should not be used.

4. Applications use default URIs and Properties when requesting CacheManagers.

5. Applications avoid using optional features of the specification, or use the `CachingProvider.isSupported` method to exploit optional features when they are present.

   For example, store-by-reference in-process implementations may have much higher performance than store-by-value because keys and values may be referenced directly.

6. Applications keep proprietary configuration in proprietary declarative configuration files rather than using proprietary programmatic `Cache` construction.

7. The `unwrap` method in `CacheManager`, `Cache`, `Cache.Entry` and `CacheInvocationContext` not be used.

   These are used to gain access the proprietary backing Cache and Cache.Entry respectively. Using proprietary APIs reduces portability.

8. Applications do not make assumptions about `Cache` topology.

   For example, assuming a listener will be executed locally, and creating a dependence on local

application class instances, may not be portable across implementations.

9. For CAS operations, store-by-reference will use the `equals()` method for comparison, but store-by-value will not necessarily. Ensure that the `equals()` implementation on custom value types takes account of all non-transient fields so that a comparison of a serialized form will give the same result as a comparison using the `equals()` method.

10. Applications avoid making references and calls to `Cache` and `CacheManager` methods in implementations of `EntryProcessor`, `CacheEntryListener`, `CacheLoader`, `CacheWriter` or `ExpiryPolicy`.

    Re-entrant algorithms may lead to unpredictable application behavior including runtime exceptions, deadlock and/or unbounded resource consumption.

14. Glossary

Application	A Java application that uses the Java Caching API.
Cache	A named and configured collection of Entries.
Cache Manager	A container for caches, that holds references to them.
Cache Operation	An invocation of a method on `Cache`.
Caching Provider	An implementation of this specification. See Caching Implementation
CacheLoader	A user-defined Class that is used to load key/value pairs into a `Cache` on demand.
CacheWriter	A user-defined Class that 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 that listens to `Cache` events.
CAS	The compare and swap cache operations named after the CPU operations that operate in the same way.
Compile Time	The time when source code is compiled into Java byte code
Configuration Time	The time when a new cache is being configured and before it is available for use
Developer	An application developer using the Java Caching API.
Entry	A cache entry, consisting of a unique key and a value
Eviction	The process of removing entries from a `Cache` when the `Cache` has exceeded a resource limit.
Expiry	The process of ensuring entries are no longer available to an application because they are no longer considered valid.
Expiry Policy	A policy that defines when a `Cache.Entry` is considered expired, and therefore should not be available to an application.
External Resource	A resource, external to the cache, loaded from or written to by a `CacheLoader` or `CacheWriter` respectively.
Implementer	The supplier of a caching implementation.
Implementation	An implementation of this specification.
Key	A way of unambiguously identifying a unique item in a Cache.

Operational Method	A method that when executed attempts to change or access entries in a Cache e.g., `Cache.put(...)`, `Cache.get(...)`, or change the state of a Cache (enabling management etc). Non-Operational methods however are those that typically request the status of a Cache and do not rely on underlying resources or entries being available e.g., `Cache.isClosed()`, `Cache.getName()`. The list of Operational methods for `CacheManager` and `Cache` are defined in the chapters on each respectively.
Read-Through	If a mapping is missing from the cache, a loader will be invoked to read data in from an external resource.
Runtime	The time when methods on a cache are invocable.
Store By Reference	The cache stores entries by making a copy of the provided references to the keys and values.
Store By Value	The cache stores entries by making a complete copy of the keys and values state.
Value	The value stored in a `Cache`. Any Java `Object` can be a value.
Write-Through	On a cache mutation, a writer will be invoked to write data to an external resource.

15. Bibliography

[1] JSR345: Enterprise JavaBeans, v. 3.2. EJB Core Contracts and Requirements. http://jcp.org/en/jsr/proposalDetails?id=345

[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.*http://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-338: Java™ Persistence 2.1 http://jcp.org/en/jsr/detail?id=338

[8] JSR-336:Java™ SE 7. http://jcp.org/en/jsr/detail?id=336

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

[10] JTA Specification 2.0, http://jcp.org/en/jsr/detail?id=907

[11] CDI Specification 1.1, http://jcp.org/en/jsr/detail?id=346

[12] Expression Language 3.0 http://jcp.org/en/jsr/detail?id=341

[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)

16. Appendix A - Revision History

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

16.1. Early Draft 1

Created initial draft. This document was incomplete.

16.2. Public Review Draft

Created first complete draft.

16.3. Second Public Review Draft

1. https://github.com/jsr107/jsr107spec/issues/170 - Minor Typo.
2. https://github.com/jsr107/jsr107spec/issues/169 - MInor Typo.
3. https://github.com/jsr107/jsr107spec/issues/168 - Minor Typo.
4. https://github.com/jsr107/jsr107spec/issues/167 - Minor Typo.
5. https://github.com/jsr107/jsr107spec/issues/166 - Minor Typo.
6. https://github.com/jsr107/jsr107spec/issues/162 - `DefaultCacheResolver` updated for new creational mechanism.
7. https://github.com/jsr107/jsr107spec/pull/172 - Minor JavaDoc formatting.
8. https://github.com/jsr107/jsr107spec/issues/184 Minor JavaDoc corrections on `CacheLoader`.
9. https://github.com/jsr107/jsr107spec/issues/182 Fix omission in Section 8.4, invocations of listeners.
10. https://github.com/jsr107/jsr107spec/issues/181 Clarify behaviour of when write-through is called by `invoke` and `invokeAll` in Section 7.3.
11. https://github.com/jsr107/jsr107spec/issues/180 Setion 7.2, Read-Through Caching. Add behaviour to `invoke` and `invokeAll` where `getValue()` is called.
12. https://github.com/jsr107/jsr107spec/issues/178 Section 6. Expiry Policy. Changes to the method list table.
13. https://github.com/jsr107/jsr107spec/issues/176 Minor Typos.

14. https://github.com/jsr107/jsr107spec/issues/173 Contradiction around the read-through behaviour of `getAndRemove` and `getAndReplace`

15. https://github.com/jsr107/jsr107spec/issues/179 Renamed `TouchedPolicy` to `TouchedExpiryPolicy`

16. https://github.com/jsr107/jsr107spec/issues/191 Remove obsolete entries from the expert group listing.

17. https://github.com/jsr107/jsr107spec/issues/190 Numerous spec doc typos, grammar and formatting edits after proof read.

18. https://github.com/jsr107/jsr107spec/issues/188 Fix threading bug identified by FindBugs in `CompletionListenerFuture`

19. https://github.com/jsr107/jsr107spec/issues/185 Change MutableEntry.getValue() to Entry.getValue() in JavaDoc.

20. https://github.com/jsr107/jsr107spec/issues/177 Changes to examples in Section 5.2, Type-Safety.

21. https://github.com/jsr107/jsr107spec/issues/175 Removed IllegalStateException on `CacheManager.destroyCache(String cacheName)`

22. https://github.com/jsr107/jsr107spec/issues/174 Changed `configureCache` to two methods: `getOrCreateCache`, that works the same, and `createCache`, for create only.

23. https://github.com/jsr107/jsr107spec/issues/187 Added dynamic methods for registering and deregistering `CacheEntryListener`s on `Cache`.

24. https://github.com/jsr107/jsr107spec/issues/210 Add a very simple `Caching.getCache(String cacheName, Long.class, String.class);`

25. https://github.com/jsr107/jsr107spec/issues/194 Remove `CacheManager.isSupported` as it serves no purpose above having it on `CachingProvider`.

26. https://github.com/jsr107/jsr107spec/issues/207 Remove `Cache.getOrCreate`. Separate methods are provided for creation and lookup.

27. https://github.com/jsr107/jsr107spec/issues/212 Typo.

28. https://github.com/jsr107/jsr107spec/issues/211 Minor clarifications to Section 2.2 and Section 15.

29. https://github.com/jsr107/jsr107spec/issues/197 Clarify cache topologies.

30. https://github.com/jsr107/jsr107spec/issues/200 Simplfiy and update simple example.

31. https://github.com/jsr107/jsr107spec/issues/202 Clarified the scope and appropriate use of Cache Names.

32. https://github.com/jsr107/jsr107spec/issues/199 Documented the purpose of CacheManager URIs.

33. https://github.com/jsr107/jsr107spec/issues/213 Clarify that it's a non-objective for the Caching API to keep a Cache in sync with an external resource.

34. https://github.com/jsr107/jsr107spec/issues/205 Clarified when Cache Configuration validation occurs.

35. https://github.com/jsr107/jsr107spec/issues/186 Clarified EntryProcessor atomicity semantics.

36. https://github.com/jsr107/jsr107spec/issues/206 Introduced Portability Recommendations section.

37. https://github.com/jsr107/jsr107spec/issues/201 Specification Corrections

38. https://github.com/jsr107/jsr107spec/issues/203 JavaDoc Corrections

16.4. Proposed Final Draft Specification

1. https://github.com/jsr107/jsr107spec/issues/204 Add comment on security

2. https://github.com/jsr107/jsr107spec/issues/220 Correct `CacheWriter` behaviour for the atomic replace methods.

3. https://github.com/jsr107/jsr107spec/issues/221 Clarify `CacheWriter` behaviour for the `cache.removeAll` methods.

4. https://github.com/jsr107/jsr107spec/issues/222 Clarify JavaDoc around `CacheWriter`s

5. https://github.com/jsr107/jsr107spec/issues/223 `CachingProviderRegistry` should be private.

6. https://github.com/jsr107/jsr107spec/pull/225 Typo in `Cache` JavaDoc.

7. https://github.com/jsr107/jsr107spec/issues/218 Clarify that `CacheWrtier.remove` and `CacheWriter.removeAll` are invoked on remove even if no mapping exist(s).

8. https://github.com/jsr107/jsr107spec/issues/224 Wrap end user code called by `Cache` in an appropriate `CacheException` subclass.

9. https://github.com/jsr107/jsr107spec/issues/227 Move `EntryProcessor` related interfaces to their own package.

10. https://github.com/jsr107/jsr107spec/issues/215 Clarify `EntryProcessor` semantics.

11. https://github.com/jsr107/jsr107spec/issues/226 createCache should return the created cache or throw an exception (not be a void method)

12. https://github.com/jsr107/jsr107spec/issues/228 Changed the use of “null” to represent “untyped Caches” to use Object.class.

13. https://github.com/jsr107/jsr107spec/issues/143 Changed `ExpiryPolicy` methods to take key as a parameter rather than `Cache.Entry` to allow more efficient implementations.

14. https://github.com/jsr107/jsr107spec/issues/230 Import JavaDoc link and see references to make JavaDoc more readable.

15. https://github.com/jsr107/jsr107spec/issues/229 System property `javax.cache.cachingprovider` should be `javax.cache.spi.cachingprovider`.

16. https://github.com/jsr107/jsr107spec/issues/143 Changed `ExpiryPolicy` methods from `L extends K` to `K`.

17. https://github.com/jsr107/jsr107spec/issues/214 Brought JavaDoc and Specification into syn regarding `CacheManager` identity.

18. https://github.com/jsr107/jsr107spec/issues/217 Clarified requirements around `Object.equals` and `Object.hsdhcode` for keys.

19. https://github.com/jsr107/jsr107spec/issues/231 Rename `@CacheRemoveEntry` to `@CacheRemove`.

20. https://github.com/jsr107/jsr107spec/issues/236 Remove transactions, following a vote by the EG.

21. https://github.com/jsr107/jsr107spec/issues/235 Clarify `ExpiryPolicy` table for invoke.

22. https://github.com/jsr107/jsr107spec/issues/239 Loosen contract on equals for CAS so that server side comparisons can easily be implemented.

23. https://github.com/jsr107/jsr107spec/issues/233 Removed key parameter from ExpiryPolicy. Renamed methods as Entry Keys and Values are no longer provided. Consequently removed the ShoppingCart custom expiry example.

24. https://github.com/jsr107/jsr107spec/issues/237 Resolved issue where a FactoryBuilder could not be used to produce instances of static inner classes.

25. https://github.com/jsr107/jsr107spec/issues/243 Change EntryProcessor to an interface of the form Functional Interface so that it can be used by Lambda in Java 8.

26. https://github.com/jsr107/jsr107spec/issues/198 Clarify Execution Context.

27. https://github.com/jsr107/jsr107spec/issues/196 Clarify Default Consistency.

28. https://github.com/jsr107/jsr107spec/issues/249 CacheLoader.load method changed to return V instead of Entry<K, V>.

29. https://github.com/jsr107/jsr107spec/issues/246 Clarified semantics of removeAll() in terms of when CacheWriter.deleteAll(...) is called.

30. https://github.com/jsr107/jsr107spec/issues/234 Resolved inconsistent treatment of expiry checking between the Cache.replace(K, V, V) and Cache.remove(K, V) methods.

31. https://github.com/jsr107/jsr107spec/issues/248 Removed optional features from the samples.

32. https://github.com/jsr107/jsr107spec/issues/242 Added LICENSE.txt to each module of the source code.

33. https://github.com/jsr107/jsr107spec/pull/238 Fixed JavaDoc warnings.

34. https://github.com/jsr107/jsr107spec/issues/240 Added recommendations about avoiding re-entrant Cache and Cache Manager method invocation.

35. https://github.com/jsr107/jsr107spec/issues/250 Change from the platform MBeanServer to an implementation specific MBeanServer for management.

16.5. Final Release Specification

1. https://github.com/jsr107/jsr107spec/issues/251 Change Configuration from an abstract class to interface.

2. https://github.com/jsr107/jsr107spec/issues/252 Introduce `<C extends Configuration>` to provide more flexibility in configuration.

3. https://github.com/jsr107/jsr107spec/issues/253 Exclude `Cache.containsKey` from cache hit and miss statistics.

4. https://github.com/jsr107/jsr107spec/issues/255 Changed status of java and javax in cache namespaces from reserved to recommended not to be used for portability.

5. https://github.com/jsr107/jsr107spec/issues/256 Refactored `Cache.invokeAll(...)` to return a `Map<K, EntryProcessorResult<T>>` to allow returning values and exceptions per entry.

6. https://github.com/jsr107/jsr107spec/issues/257 `Cache.remove(K key)` should only update remove statistics if the method returns true.

7. https://github.com/jsr107/jsr107spec/issues/258 `Cache.removeAll()` and `Cache.removeAll(Set keys)` should update remove statistic for each key removed.

8. https://github.com/jsr107/jsr107spec/issues/259 `Cache` should not implement `Closeable`.

9. https://github.com/jsr107/jsr107spec/issues/260 Clarify runtime type enforcement

10. https://github.com/jsr107/jsr107spec/issues/266 The `Caching.setDefaultClassLoader(...)` method was not static.

11. https://github.com/jsr107/jsr107spec/issues/267 Refined the requirement that implementations, while not required to call `Object.hashCode` or `Object.equals` on custom classes, are required to provide the same semantics for this functionality if they are not called on the custom classes.

12. https://github.com/jsr107/jsr107spec/issues/268 Clarified that although the specification defines a minimal `Configuration` and `CompleteConfiguration` interface, only implementations that support the `CompleteConfiguration` interface will be compliant to the specification.

13. https://github.com/jsr107/jsr107spec/issues/270 Added a new section to define the concepts of store-by-value and store-by-reference with respect to Caching.

14. https://github.com/jsr107/jsr107spec/issues/272 Added missing `CacheManager.getClassLoader()` method.

15. https://github.com/jsr107/jsr107spec/issues/274 Clarified that the Default Consistency model must be supported by implementations at a minimum.

16. https://github.com/jsr107/jsr107spec/issues/275 Improve the clarity of Execution Context for `EntryProcessor`s, `CacheLoader`s, `CacheWriter`s, `CacheEntryListener`s and `ExpiryPolicy`.

17. https://github.com/jsr107/jsr107spec/issues/278 Corrected misleading javadoc for `CacheManager.createCache(...)` method.

18. https://github.com/jsr107/jsr107spec/issues/279 Added javadoc for those methods that are likely to throw a `SecurityException` in a secure environment.

19. https://github.com/jsr107/jsr107spec/issues/280 Removed ambiguous and potentially misleading Cache name and scoping statement regarding the identity of Caches between Cache Managers. This may not be implementable.

20. https://github.com/jsr107/jsr107spec/issues/281 Allowed `CacheManager.getCacheNames()` to return an `Iterable` that excludes implementation and platform specific `Cache`s.

21. https://github.com/jsr107/jsr107spec/issues/283 Clarified that the semantics of `Cache`s that may be defined by the `URI` of a `CacheManager` is implementation dependent.

22. https://github.com/jsr107/jsr107spec/issues/284 Removed speculative statements about Java EE semantics and support requirements.

23. https://github.com/jsr107/jsr107spec/issues/285 Introduced missing remove support for CacheEntryListenerConfigurations in MutableConfiguration. Changed the return type to an Iterable (as internally they are no longer a set).

24. https://github.com/jsr107/jsr107spec/issues/286 Numerous editorial changes including grammatical changes.

25. https://github.com/jsr107/jsr107spec/issues/287 Remove cacheNull @CachePut parameter. Cache values cannot be null.

26. https://github.com/jsr107/jsr107spec/issues/288 Added definitions of Operational Methods.

27. https://github.com/jsr107/jsr107spec/issues/289 Correct inconsistency in CacheWriter.deleteAll method.

28. https://github.com/jsr107/jsr107spec/issues/290 Resolved incorrect uses of the word “which” with “that”.

29. https://github.com/jsr107/jsr107spec/issues/273 Added description to package-info.java.