

# cache2k User Guide

Jens Wilke

Version 1.0.2.Final

# Table of Contents

1. About	i1
1.1. Versioning	i1
1.2. How to read the documentation	i1
1.3. Copyright	i2
2. Getting Started	i3
2.1. Obtaining cache2k	i3
2.2. Using a Cache	i3
2.3. Cache Aside	i4
2.4. Read Through	i5
2.5. Using Null Values	i6
2.6. Composite Keys	i6
2.7. Keys Need to be Immutable	i7
2.8. Mutating Values	i8
2.9. Exceptions and Caching	i8
2.10. Don't Panic!	i8
3. Types	i9
3.1. Constructing a Cache with Generic Types	i9
3.2. Constructing a Cache without Generic Types	i9
3.3. Key Type	i9
3.4. Value Type	i10
3.5. Untyped Caches	i10
3.6. Future Enhancements	i10
4. Atomic Operations	i11
4.1. Example	i11
4.2. Built-in Atomic Operations	i11
4.3. Entry Processor	i12
5. Loading / Read Through	i14
5.1. Benefits of Read Through Operation	i14
5.2. Defining a Loader	i14
5.3. Advanced Loader	i14
5.4. Using Lambda Loaders in Java 8	i15
5.5. Prefetching	i15
5.6. Invalidating	i15
5.7. Transparent Access	i15
6. Expiry and Refresh	i17
6.1. Specifying an Expiry Duration	i17
6.2. Variable Expiry	i17
6.3. Lagging Expiry	i17

6.4. Sharp Expiry .....	17
6.5. Loader Exceptions and Expiry .....	18
6.6. Resetting the Expiry of a Cache Value .....	18
6.7. Wall Clock and Clock Skew .....	18
7. Refresh Ahead .....	20
7.1. Setup .....	20
7.2. Semantics .....	20
7.3. Sharp Expiry vs. Refresh Ahead .....	20
7.4. Rationale: No separate refresh timing parameter? .....	21
7.5. Future Outlook .....	21
8. Null Values .....	22
8.1. The Pros and Cons of Nulls .....	22
8.2. Negative Result Caching .....	22
8.3. Alternatives .....	22
8.4. Default Behavior .....	22
8.5. How to Enable Null Values .....	22
8.6. How to Operate with Null Values .....	23
8.7. Loader and Null Values .....	24
8.8. Performance .....	24
8.9. Rationale .....	24
9. Exceptions and Resilience .....	26
9.1. Behavior .....	26
9.2. Retry .....	26
9.3. Exception Propagation .....	27
9.4. Invalidating .....	27
9.5. Entry Status and <code>containsKey</code> .....	27
9.6. Configuration options .....	27
9.7. Examples .....	28
9.8. Custom resilience policy .....	29
9.9. Debugging .....	29
10. Event Listeners .....	31
10.1. Listening to Events .....	31
10.2. Async Listeners .....	31
11. Configuration via XML .....	32
11.1. Using the XML Configuration .....	32
11.2. Combine Programmatic and XML Configuration .....	33
11.3. Reference Documentation .....	34
11.4. Rationale .....	37
12. Logging .....	38
12.1. Supported Log Infrastructure .....	38
12.2. Wiring a Custom Log Target .....	38

13. Statistics .....	39
13.1. JMX Metrics .....	39
13.2. <code>toString()</code> Output .....	39
13.3. Accuracy, Overhead and Performance .....	40
13.4. Internal Statistics .....	40
13.5. Noteworthy Metrics .....	40
14. Android .....	42
14.1. Usage .....	42
14.2. XML Configuration .....	42
15. JCache .....	43
15.1. Maven Dependencies .....	43
15.2. Getting Started with the JCache API .....	43
15.3. Configuration .....	43
15.4. Control Additional JCache Semantics .....	45
15.5. Implementation Details .....	45
15.6. Performance .....	47
15.7. Compliance Testing .....	47

# Chapter 1. About

cache2k is a compact in-process cache implementation that has the following main design goals:

## *Performance*

fundamentally different eviction algorithm to allow fast and lock free reads

## *Ease of use*

defined API, solve common problems with one configuration option

## *Compact and modular*

small core size

## 1.1. Versioning

The JBoss versioning scheme is followed (<https://developer.jboss.org/wiki/JBossProjectVersioning>). Furthermore, a Tick-Tock scheme is used to mark development releases. Examples:

### *1.0.0.Final*

Major version.

### *1.0.1.Final*

Service release. Should be binary compatible to previous release.

### *1.1.0.Beta*

Odd minor version, development version. A beta version may be used in production, but additional features may still change the API and may not completely tested.

### *1.2.0.Final*

Even minor version, stable release, new features and compatible changes to the previous version. Not be strictly binary compatible to the previous stable release. Interfaces not meant for extension may get new methods.

### *2.0.0.Final*

New Major version, adds and removes features, may have incompatible changes to the previous version.

## 1.2. How to read the documentation

The documentation is intended as a overview guide through the functionality of cache2k and will help you discover every important feature. At some points rationale or background information is given. It is not complete. You will find additional information in the API JavaDoc, in examples, and in the test cases.

A [cache2k User Guide PDF Version](#) is available as well.

## 1.3. Copyright

The documentation is licensed under the terms of [CC BY 4.0](#).

# Chapter 2. Getting Started

## 2.1. Obtaining cache2k

The latest cache2k version is available on maven central. The recommended way to include it in your project is to add the following dependencies:

```

<properties>
  <cache2k-version>1.0.2.Final </cache2k-version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.cache2k</groupId>
    <artifactId>cache2k-api </artifactId>
    <version>${cache2k-version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.cache2k</groupId>
    <artifactId>cache2k-all </artifactId>
    <version>${cache2k-version}</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

This will add the single `cache2k-all` JAR file to your application deliverable. For compiling and development the `cache2k-api` is included which contains API classes only. The above requires at least a `Java SE 6` compatible runtime. For usage with Android, see the [Android](#) chapter.

## 2.2. Using a Cache

For starting with cache2k, let's construct a cache that looks up the preferred airline for one of our frequent flight routes.

```

Ê Cache<String, String> cache = new Cache2kBuilder<String, String>() {}
Ê     .name("routeToAirline")
Ê     .eternal(true)
Ê     .entryCapacity(100)
Ê     .build();
Ê // populate with our favorites
Ê cache.put("MUC-SFO", "Yeti Jet");
Ê cache.put("SFO-LHR", "Quality Air");
Ê cache.put("LHR-SYD", "Grashopper Lifting");
Ê // query the cache
Ê String route = "LHR-MUC";
Ê if (cache.containsKey(route)) {
Ê     System.out.println("We have a favorite airline for the route " + route);
Ê } else {
Ê     System.out.println("We don't have a favorite airline for the route " + route);
Ê }
Ê String airline = cache.peek(route);
Ê if (airline != null) {
Ê     System.out.println("Let's go with " + airline);
Ê } else {
Ê     System.out.println("You need to find one yourself");
Ê }

```

To obtain the cache a new `Cache2kBuilder` is created. Mind the trailing `{}`. There are a dozens of options that can alter the behavior of a cache. In the example above the cache gets a name and is instructed to keep entries forever via `eternal(true)`. A name needs to be unique and may be used to find and apply additional configuration parameters and to make statistics available via JMX. Find the details about naming a cache at in JavaDoc of `Cache2kBuilder.name()`.

The cache has a maximum capacity of 100 entries. When the limit is reached an entry is automatically removed that is not used very often. This is called eviction.

In the example `Cache.put()` is used to store values. The cache content is queried with `Cache.peek()` and `Cache.containsKey()`.

## 2.3. Cache Aside

Let's consider we have an operation called `findFavoriteAirline()`, that checks all our previous flights and finds the airline we liked the most. If we never did fly on that route it will ask all our friends. Of course this is very time consuming, so we first ask the cache, whether something for that flight route is already known, if not, we call the expensive operation.



```

Ê Cache<String, String> routeToAirline = new Cache2kBuilder<String, String>() {}
Ê   .name("routeToAirline")
Ê   .build();

Ê private String findFavoriteAirline(String origin, String destination) {
Ê     // expensive operation to find the best airline for this route
Ê     // for example, ask all friends...
Ê }

Ê public String lookupFavoirteAirline(String origin, String destination) {
Ê   String route = origin + "-" + destination;
Ê   String airline = routeToAirline.peek(route);
Ê   if (airline == null) {
Ê     airline = findFavoriteAirline(origin, destination);
Ê     routeToAirline.put(route, airline);
Ê   }
Ê   return airline;
Ê }

```

The above pattern is called *cache aside*.

## 2.4. Read Through

An alternative way to use the cache is the so called *read through* operation. The cache configuration gets customized with a loader, so the cache knows how to retrieve missing values.

```

Ê Cache<String, String> routeToAirline = new Cache2kBuilder<String, String>() {}
Ê   .name("routeToAirline")
Ê   .loader(new CacheLoader<String, String>() {
Ê     @Override
Ê     public String load(final String key) throws Exception {
Ê       String[] port = key.split("-");
Ê       return findFavoriteAirline(port[0], port[1]);
Ê     }
Ê   })
Ê   .build();

Ê private String findFavoriteAirline(String origin, String destination) {
Ê   // expensive operation to find the best airline for this route
Ê }

Ê public String lookupFavoirteAirline(String origin, String destination) {
Ê   String route = origin + "-" + destination;
Ê   return routeToAirline.get(route);
Ê }

```

Now we use `Cache.get` to request a value from the cache, which transparently invokes the loader and populates the cache, if the requested value is missing. Using a cache in read through mode has

various advantages:

- ¥ No boilerplate code as in cache aside
- ¥ Protection against the cache stampede (See [Wikipedia: Cache Stampede](#))
- ¥ Automatic refreshing of expired values is possible ([refresh ahead](#))
- ¥ Built-in exception handling like suppression and retries (see [Resilience](#) chapter)

## 2.5. Using Null Values

The simple example has a major design problem. What happens if no airline is found? Typically caches don't allow `null` values. When you try to store or load a `null` value into cache2k you will get a `NullPointerException`. Sometimes it is better to avoid `null` values, in our example we could return a list of favorite airlines which may be empty.

In case a `null` value is the best choice, it is possible to store it in cache2k by enabling it with `permitNullValues(true)`. See the [Null Values chapter](#) for more details.

## 2.6. Composite Keys

In the example the key is constructed by concatenating the origin and destination airport. This is ineffective for several reasons. The string concatenation allocates two temporary objects (the `StringBuilder` and its character array); if we need the two parts again we have to split the string again. A better way is to define a dedicated class for the cache key that is a tuple of origin and destination.

```

Ê public final class Route {
Ê     private String origin;
Ê     private String destination;

Ê     public Route(final String origin, final String destination) {
Ê         this.destination = destination;
Ê         this.origin = origin;
Ê     }

Ê     public String getOrigin() {
Ê         return origin;
Ê     }

Ê     public String getDestination() {
Ê         return destination;
Ê     }

Ê     @Override
Ê     public boolean equals(final Object other) {
Ê         if (this == other) return true;
Ê         if (other == null || getClass() != other.getClass()) return false;
Ê         Route route = (Route) other;
Ê         if (!origin.equals(route.origin)) return false;
Ê         return destination.equals(route.destination);
Ê     }

Ê     @Override
Ê     public int hashCode() {
Ê         int hashCode = origin.hashCode();
Ê         hashCode = 31 * hashCode + destination.hashCode();
Ê         return hashCode;
Ê     }
Ê }

```

Cache keys need to define a proper `hashCode()` and `equals()` method.

## 2.7. Keys Need to be Immutable



### *Don't mutate keys*

For a key instance it is illegal to change its value after it is used for a cache operation. The cache uses the key instance in its own data structure. When defining your own keys, it is therefore a good idea to design them as immutable object.

The above isn't special to caching or `cache2k`, it applies identically when using a Java `HashMap`.

## 2.8. Mutating Values

It is illegal to mutate a cached value after it was stored in the cache, unless `storeByReference` is enabled. This parameter instructs the cache to keep all cached values inside the heap.

Background: `cache2k` stores its values in the Java heap by the object reference. This means mutating a value, will affect the cache contents directly. Future versions of `cache2k` will have additional storage options and allow cache entries to be migrated to off heap storage or persisted. In this case mutating cached values directly will lead to inconsistent results.

## 2.9. Exceptions and Caching

When using read through and a global expiry time (`expireAfterWrite`) is set, exceptions will be cached and/or suppressed.

A cached exception will be rethrown every time the key is accessed. After some time passes, the loader will be called again. A cached exception can be spotted by the expiry time in the exception text, for example:

```
org.cache2k.integration.CacheLoaderException: expiry=2016-06-04 06:08:14.967, cause:
java.lang.NullPointerException
```

Cached exceptions can be misleading, because you may see 100 exceptions in your log, but only one was generated from the loader. That's why the expiry of an exception is typically shorter than the configured expiry time.

When a previous value is available a subsequent loader exception is suppressed for a short time. For more details on this behavior see the [Resilience chapter](#).

## 2.10. Don't Panic!

Also those familiar with caching might get confused by the many parameters and operations of `cache2k` controlling nuances of caching semantics. Except for the exceptions caching described above everything will work as you will expect from a cache. There is no need to know every feature in detail, yet. Think of them as a parachute. Usually you don't need them, but when in trouble, there is one parameter that will save you.

Whenever in doubt: For asking questions please use the *Stackoverflow* tag `cache2k`. Please describe your scenario and the problem you try to solve first before asking for specific features of `cache2k` and how they might help you.

# Chapter 3. Types

This section covers:

- ¥ How to construct a cache with concrete types
- ¥ Why you should use types
- ¥ How generic types are captured

## 3.1. Constructing a Cache with Generic Types

When using generic types, the cache is constructed the same way as already shown in the *Getting started* chapter.

```
Ê Cache<Long, List<String>> cache =  
Ê     new Cache2kBuilder<Long, List<String>>() {}  
Ê     .eternal(true)  
Ê     .build();
```

The `{}` is a trick which constructs an anonymous class, which contains the complete type information. If just an object would be created the complete type information would not be available at runtime and could not be used for configuring the cache.

Caches can be constructed dynamically with arbitrary generic types. The type information can be specified via the interface `CacheType`.

## 3.2. Constructing a Cache without Generic Types

If the cache types do not contain generic types, then the following simpler builder pattern can be used:

```
Ê Cache<Long, String> cache =  
Ê     Cache2kBuilder.of(Long.class, String.class)  
Ê     .eternal(true)  
Ê     .build();
```

The additional generated class as in the previous version is not needed, which saves a few bytes program code.

## 3.3. Key Type

The key type needs to implement `equals()` and `hashCode()`. Arrays are not valid for keys.

## 3.4. Value Type

Using arrays as values is discouraged, because some cache operations testing for value equality, like `Cache.replaceEquals()`, will not work as desired on arrays. To prevent problems, cache2k refuses to build a cache with an array value type specified at the configuration. However, this protection can be circumvented by not providing the proper type in the cache configuration.

If the value type is implementing `ValueWithExpiryTime`, an expiry policy is added automatically.

## 3.5. Untyped Caches

It is possible to construct an untyped cache via `Cache2kBuilder.forUnknownTypes()`. But, the use of untyped caches is discouraged. If different types need to be stored in a cache, construct a separate cache for each type with the proper type information.

## 3.6. Future Enhancements

Future versions of cache2k will leverage the type information for:

- ¥ optimizations depending on the type
- ¥ optional strict type checking
- ¥ optional copying
- ¥ derive a optimal marshaller for off heap overflow and persistence

# Chapter 4. Atomic Operations

Atomic operations allow a combination of read or compare and modify a cached entry without interference of another thread.

## 4.1. Example

The method `Cache.replaceIfEquals()` has the following semantics:

```
if (cache.containsKey(key) && Objects.equals(cache.get(key), oldValue)) {
    cache.put(key, newValue);
    return true;
} else
    return false;
```

When executing the above statements sequentially, the outcome can vary because other threads can interfere. The atomic operation semantic guarantees that this is not possible.

These kind of operations are also called CAS (compare and swap) operations.

## 4.2. Built-in Atomic Operations

In general all operations on a single entry have atomic guarantee. Bulk operations, meaning operations on multiple keys, like `getAll()`, don't have atomic guarantees.

Operation	Description
<code>boolean containsAndRemove(key)</code>	remove and return true if something was present
<code>V peekAndRemove(key)</code>	remove and return existing value
<code>boolean putIfAbsent(key, value)</code>	insert value only if no value is present
<code>V peekAndReplace(key, value)</code>	Replace value and return old value
<code>boolean Cache.replace(key, value)</code>	Replace value and return true when successful
<code>V peekAndPut(key, value)</code>	Insert or update value and return the previous value
<code>boolean replaceIfEquals(key, expectedValue, newValue)</code>	Replace value only when old value is present in the cache
<code>boolean removeIfEquals(key, expectedValue)</code>	Remove only if present value matches

## 4.3. Entry Processor

With the entry processor it is possible to implement arbitrary complex operations that are processed atomically on a cache entry. The interface `EntryProcessor` must be implemented with the desired semantics and invoked on a cached value via `Cache.invoke()`. The bulk operation `Cache.invokeAll()` is available to process multiple cache entries with one entry processor.

Here is an example which implements the same semantics as `replaceAllEquals()`:

```
Ê    final K key = ...
Ê    final V oldValue = ...
Ê    final V newValue = ...
Ê    EntryProcessor<K, V, Boolean> p = new EntryProcessor<K, V, Boolean>() {
Ê        public Boolean process(MutableCacheEntry<K, V> entry) {
Ê            if (!entry.exists()) {
Ê                return false;
Ê            }
Ê            if (oldValue == null) {
Ê                if (null != entry.getValue()) {
Ê                    return false;
Ê                }
Ê            } else {
Ê                if (!oldValue.equals(entry.getValue())) {
Ê                    return false;
Ê                }
Ê            }
Ê            entry.setValue(newValue);
Ê            return true;
Ê        }
Ê    };
Ê    return cache.invoke(key, p);
Ê }
```

Since it is an atomic operation multiple calls on `MutableCacheEntry` may have no effect if neutralising each other. For example:

```
Ê    entry.setValue("xy");
Ê    entry.setValue("abc");
Ê    entry.remove();
```

The effect will be:

- ¥ If an entry for the key is existing, the entry will be removed
- ¥ If no entry for the key is existing, the cache state will not change
- ¥ If a cache writer is attached, `CacheWriter.delete()` will be called in any case

Via the entry processor, it is also possible to specify an expiry time directly. Here is an example



formulated as Java 8 lambda expression, which inserts a value and sets the expiry after 120 minutes:

```
cache.invoke("key",  
    e -> e.setValue("value")  
        .setExpiry(System.currentTimeMillis() + TimeUnit.MINUTES.toMillis(120)));
```

# Chapter 5. Loading / Read Through

In read through operation the cache will fetch the value by itself when the value is retrieved, for example by `Cache.get()`. This is also called a *loading cache* or a *self populating cache*.

## 5.1. Benefits of Read Through Operation

When caching reads, using the cache in read through operation has several advantages:

- ¥ No boilerplate code as in cache aside
- ¥ Data source becomes configurable
- ¥ Blocking load operations for identical keys, protection against the cache stampede (See [Wikipedia: Cache Stampede](#))
- ¥ Automatic refreshing of expired values (refresh ahead)
- ¥ Build-in exception handling like suppression and retries (see [Resilience](#) chapter)

## 5.2. Defining a Loader

A loader is defined by implementing the abstract class `CacheLoader`. See the [\[getting started\]](#) example about read through.

```
EV load(K key) throws Exception;
```

The loader actions may only depend on the input key parameter. In case the load operation will yield an exception it may be passed on to the cache. How exceptions are handled by the cache is defined by the resilience policy and explained in the [Resilience chapter](#).

The JavaDoc about the `CacheLoader` contains additional details.

## 5.3. Advanced Loader

For more sophisticated load operations the `AdvancedCacheLoader` is available.

```
EV load(K key, long currentTime, CacheEntry<K,V> currentEntry) throws Exception;
```

The information of the current entry can be used to optimize the data request. A typical example is the optimization of HTTP requests. When the current cached value and its time is known the request header `If-Modified-Since` can be set from `Entry.getLastModification()`.

The JavaDoc about the `AdvancedCacheLoader` contains additional details.

## 5.4. Using Lambda Loaders in Java 8

Instead the abstract loader class, there is also a functional interface available for use with Java 8.

## 5.5. Prefetching

The cache can be instructed to load one or several values in the background with the `prefetch()` or `prefetchAll()` operations. This way data retrieval can be done in parallel and latencies can be reduced.

The number of threads used for prefetching is configured by `loaderThreadCount`.

The operation may have no effect if no loader is defined or if not enough threads are available.

## 5.6. Invalidating

In case the data was updated in the external source, the current cache content becomes invalid. To notify the cache and eventually update the cached value several options exist.

`Cache.remove(key)`

Invalidating an entry with `Cache.remove()` will cause the entry to be removed from the cache ultimately. The next time the entry is requested with `Cache.get(key)` the cache will invoke the loader (if defined). In case the loader yields an exception, this exception will be propagated to the application since there is no previous value for fallback. `Cache.remove(key)` is useful if the data is outdated and old data is not allowed to be delivered. `Cache.remove()` will also invoke `CacheWriter.delete()`, if specified. Priority is on data consistency.

`Cache.expireAt(key, Expiry.NOW)`

The entry is expired immediately. If refresh ahead is enabled the loader will be invoked in the background. Subsequent calls to `Cache.get` will block until the loading is completed and return the new value. The operation will have no effect, if there is no cached entry associated with the key. The value is still available in the cache as fallback if a loader exception occurs. This variant is the better choice if outdated values are allowed to be visible in the event of a temporary failure. An inconsistency is only allowed when a temporary failure occurs.

`Cache.expireAt(key, Expiry.REFRESH)`

When invalidating an entry via `Cache.expireAt(key, Expiry.REFRESH)` the loader gets invoked instantly if refresh ahead is enabled. If the loader is invoked, the current value will stay visible until the updated value is available. If the loader cannot be invoked, the entry is expired. The operation will have no effect, if there is no cached entry associated with the key. The value is still available in the cache as fallback if a loader exception occurs. This variant is the better choice if outdated values are allowed to be visible and the cache should continuously serve data. Priority is on availability.

## 5.7. Transparent Access

When using the cache in read through and/or in write through operation, some methods on the cache will often be misinterpreted and present pitfalls. For example, the method `Cache.containsKey`

will not return true when the value exists in the system of authority, but only reflects the cache state.

To prevent pitfalls a reduced set of interfaces is available: `KeyValueSource`, `AdvancedKeyValueSource` and `KeyValueStore` are available. These interfaces only contain methods that act transparently when a loader or writer is defined.

# Chapter 6. Expiry and Refresh

A cached value may expire after some specified time. When expired, the value is not returned by the cache any more. The cache may be instructed to do an automatic reload of an expired value, this is called *refresh ahead*.

Expiry does not mean that a cache entry is removed from the cache. The actual removal from the cache may lag behind the time of expiry.

## 6.1. Specifying an Expiry Duration

Expiry after an entry is created or modified can be specified via the `expireAfterWrite` parameter. This is also known as *time to live*. It is possible to specify different expiry values for created or modification with a custom `ExpiryPolicy`.

■

*Expiry after access* or *time to idle* is not supported, since it would compromise the high performance of cache2k. There is seldom a functional requirement for TTL, but it is used very often to minimize memory consumption in times of low activity. How a similar feature can be constructed that, at the same time, is not counter productive for performance needs further research. For discussion, see: [GH 39](#)

## 6.2. Variable Expiry

Each entry may have a different expiry time. This can be achieved by specifying an `ExpiryPolicy`. The `ExpiryPolicy` calculates a point in time, when a value expires. The configuration parameter `expireAfterWrite` is used as a maximum value.

## 6.3. Lagging Expiry

In standard operation time checks when accessing the entry are saved and the actual expiry of an entry may lag behind, meaning that entries that should expire are visible some milliseconds longer.

## 6.4. Sharp Expiry

In case there is a business requirement that data becomes invalid or needs refreshed at a defined point in time the parameter `sharpExpiry` can be enabled. This will cause that the expiry happens exactly at the point in time determined by the expiry policy. For more details see the JavaDoc or `Cache2kBuilder.sharpExpiry` and `ExpiryPolicy`.

If *sharp expiry* and *refresh ahead* is both enabled, the contract of refresh ahead is relaxed. The resulting semantics will be:

- ¥ Entries will expire exactly at the specified time
- ¥ A refresh starts at expiry time

`contains()` is `false`, if the entry is expired and not yet refreshed

¥ A `get()` on the expired entry will stall until refreshed

Sharp expiry and normal, lagging expiry can be combined. For example, if the parameter `expiryAfterWrite` and an `ExpiryPolicy` is specified and `sharpExpiry` is enabled. The sharp expiry will be used for the time calculated by the `ExpiryPolicy`, but the duration in `expiryAfterWrite` is used if this will be sooner. If the expiry is result of a duration calculation via `expiryAfterWrite` sharp expiry of the entry will not be enabled.

## 6.5. Loader Exceptions and Expiry

When an expiry duration is specified via `expiryAfterWrite`, resilience features are automatically active. See the resilience chapter for details.

## 6.6. Resetting the Expiry of a Cache Value

The expiry value can be reset with the method `Cache.expireAt(key, time)`. Some special values exist:

Table 1. `Cache.expireAt()` constants

constant	meaning
<code>Expiry.NOW</code>	The value expires immediately. An immediate load is triggered if <code>refreshAhead</code> is enabled.
<code>Expiry.REFRESH</code>	An immediate load is triggered if <code>refreshAhead</code> is enabled. If loading is not possible the value expires.
<code>Expiry.ETERNAL</code>	keep indefinitely or to a maximum of what's set with via <code>expiryAfterWrite</code>

It is possible to atomically examine a cached entry and update its expiry with the `EntryProcessor` and `MutableCacheEntry.setExpiry()`.

## 6.7. Wall Clock and Clock Skew

For timing reference the Java `System.currentTimeMillis()` is used. As with any application that relies on time, it is good practice that the system clock is synchronized with a time reference. When the system time needs to be corrected, it should adapt slowly to the correct time and keep continuously ascending.

In case a clock skew happens regularly a premature or late cache expiry may cause troubles. It is possible to do some countermeasures. If the time decreases, entries may expire more early. This can be detected and with the `AdvancedCacheLoader` the previously loaded value can be reused. If there is a time skew forward, expiry can be triggered programmatically with `expireAt()`.

Outlook: It is planed for version 1.2 to have a configurable time source, which enables better adaption to different operating environments. Ideas and requests are welcome.

# Chapter 7. Refresh Ahead

With *refresh ahead* (or *background refresh*, in a read through configuration, values about to expire will be refreshed automatically by the cache.

## 7.1. Setup

Refresh ahead can be enabled via `refreshAhead` switch. The number of threads used for refreshing is configured by `loaderThreadCount()`. A (possibly shared) thread pool for refreshing can be configured via `prefetchExecutor()`.

## 7.2. Semantics

The main purpose of refresh ahead is to ensure that the cache contains fresh data and that an application never is delayed when data expires and needs a reload. This leads to several compromises: Expired values will be visible until the new data is available from the load operation, more then the needed requests will be send to the loader.

After the expiry time of a value is reached, the loader is invoked to fetch a fresh value. The old value will be returned by the cache, although it is expired, and will be replaced by the new value, once the loader is finished. When a refresh is needed and not enough loader threads are available, the value will expire immediately and the next `get()` request will trigger the load.

Once refreshed, the entry is in a trail period. If it is not accessed until the next expiry, no refresh will be done, the entry expires and will be removed from the cache. This means that the time an entry stays within the trail period is determined by the configured expiry time or the the `ExpiryPolicy`.

■

Refresh ahead only works together with the methods invoking the loader, for example `get()` and `getAll()`. After a value is refreshed, an entry will not be visible with `containsKey` or `peek`. The first call to `get()` or `load()` on a previously refreshed item will make the loaded value available in the cache.

## 7.3. Sharp Expiry vs. Refresh Ahead

The setting `sharpExpiry` conflicts with the idea of refresh ahead. When using refresh ahead and sharp expiry in combination, the value will expire at the specified time and the background refresh is initiated. When the application requests the value between the expiry and before the new value is loaded, it blocks until the new value is available.

■

Combining sharp expiry, lagging expiry and refresh ahead, leads to an operation mode that cannot guarantee the sharp expiry contract under all circumstances. If an ongoing load operation is not finished before a sharp expiry point in time, non fresh data will become visible.

Sharp timeout can also applied on a dynamic per entry basis only when needed.



## 7.4. Rationale: No separate refresh timing parameter?

Caches supporting refresh ahead typically have separate configuration parameters for its timing. In cache2k, refreshing is done when the value would expire, which is controlled by the expiry policy and `expireAfterWrite` parameter. Why? It should be possible to enable refresh ahead with a single switch. Refreshing and expiring are two sides of the same coin: When expired, we need to refresh.

## 7.5. Future Outlook

More options to control refreshing and the trail period are added in the next releases.

The effects on the event listeners and statistics when refreshing may change in the future.

# Chapter 8. Null Values

While a `HashMap` supports `null` keys and `null` values most cache implementations and the JCache standard do not. By default, cache2k does not permit `null` values, to avoid surprises, but storing `null` values can be allowed with a configuration parameter.

## 8.1. The Pros and Cons of Nulls

A good writeup of the topic can be found at [Using and Avoiding Null Explained](#). The bottom line is, that for a map it is always better to store no mapping instead of a mapping to a `null` value. For a cache it is a different story, since the absence of a mapping can mean two things: The data was not requested from the source yet, or, there is no data.

## 8.2. Negative Result Caching

Caching that there is no result or a failure, is also called "negative result caching" or "negative caching". An example use case is the request of a database entry by primary key, for example via JPA's `EntityManager.find()` which returns an object if it is available in the database or `null` if it is not. Caching a negative result can make sense when requests that generate a negative result are common.

In a Java API negative results are quite often modeled with a `null` value. By enabling `null` support in cache2k no further wrapping is required.

## 8.3. Alternatives

In a JCache application a `null` from the `CacheLoader` means the entry should be removed from the cache. This semantic is a consistent definition, but if `Cache.get()` is used to check whether data is existing, no caching happens if no data is present. A `null` value is passed through consistently, however, the cache performs badly if a `null` response is common.

Being able to store a `null` value is no essential cache feature, since it is always possible to store a wrapper object in the cache. However, with the `null` support in cache2k, it is possible to store a `null` value with no additional overhead.

## 8.4. Default Behavior

By default, every attempt to store a `null` in the cache will yield a `NullPointerException`.

In case `peek()` returns `null`, this means there is no associated entry to this key in the cache. The same holds for `get()` if no loader is defined. For one point in time and one key there is the following invariant: `cache.contains(key) == (cache.peek(key) != null)`.

## 8.5. How to Enable Null Values

Storing of `null` values can be enabled by `permitNullValues(true)`. Example:

```

Ê Cache<Integer, Person> cache =
Ê     new Cache2kBuilder<Integer, Person>(){}
Ê     .name("persons")
Ê     .entryCapacity(10000)
Ê     .expireAfterWrite(5, TimeUnit.MINUTES)
Ê     .permitNullValues(true)
Ê     .build();

```

## 8.6. How to Operate with Null Values

When `null` values are legal, additional care must be taken. The typical cache aside pattern becomes invalid:

```

Ê Cache<Integer, Person> c = ...
Ê Person lookupPerson(int id) {
Ê     Person p = cache.peek(id);
Ê     if (p == null) {
Ê         p = retrievePerson(id);
Ê         cache.put(id, p);
Ê     }
Ê     return p;
Ê }

```

In case `retrievePerson` returns `null` for a non-existing person, it will get called again the next time the same person is requested. To check whether there is a cached entry `containsKey` could be used. However, using `containsKey()` and `get()` sequentially is faulty. To check for the existence of a cache entry and return its value the method `peekEntry` (or `getEntry` with loader) can be used. The fixed version is:

```

Ê Cache<Integer, Person> c = ...
Ê Person lookupPerson(int id) {
Ê     CacheEntry<Person> e = cache.peekEntry(id);
Ê     if (e != null) {
Ê         return e.getValue();
Ê     }
Ê     p = retrievePerson(id);
Ê     cache.put(id, p);
Ê     return p;
Ê }

```

The cache aside pattern serves as a good example here, but is generally not recommended. Better use read through and define a loader.

## 8.7. Loader and Null Values

If a loader is defined a call of `Cache.get()` will either return a non-null value or yield an exception. If the loader returns `null`, a `NullPointerException` is generated and wrapped and propagated via a `CacheLoaderException`. This behavior is different from JSR107/JCache which defines that an entry is removed, when the loader returns `null`.

In case the loader returns `null` and this should not lead to an exception the following options exist:

- ¥ Always return a non-null object and include a predicate for the `null` case, use a list or Java 8 `Optional`
- ¥ Enable `null` value support
- ¥ Remove entries from the cache when the loader returns `null` (as in JCache)

The expiry policy can be used to remove entries from the cache when the loader returns `null`:

```
Ê ...
Ê builder.expiryPolicy(new ExpiryPolicy<Integer, Person>() {
Ê     @Override
Ê     public long calculateExpiryTime(K key, V value, long loadTime, CacheEntry
<Integer, Person> oldEntry) {
Ê         if (value == null) {
Ê             return NO_CACHE;
Ê         }
Ê         return ETERNAL;
Ê     }
Ê })
Ê ...
```

This works, since the cache checks the `null` value only after the expiry policy has run and had decided to store the value.

## 8.8. Performance

Storing `null` values has no additional memory or CPU overhead.

## 8.9. Rationale

### 8.9.1. Why support `null`?

Supporting `null` needs a more careful design inside the cache and its API. When this is done, it basically comes for free and makes the cache very effective for use cases where `null` values are common.

### 8.9.2. Why is rejecting `null` values the default?

We were using cache2k for 16 years, with the capability to store `null` values by default. For the 1.0

version we changed this behavior and don't allow nulls. Here is why:

- ¥ Most caches do not support `null` values. Allowing `null` by default may lead to unexpected and incompatible behavior.
- ¥ Use cases with `null` are rare.
- ¥ Returning or storing a `null` may be a mistake most of the time.
- ¥ In case a `null` is allowed it is better to specify this explicitly to make the different behavior more obvious.

### 8.9.3. Why rejecting `null` from the loader?

If the loader returns `null`, a `NullPointerException` is generated and propagated via the `CacheLoaderException`. This behavior is different from JSR107/JCache which defines that an entry is removed, if the loader returns `null`.

The JCache behavior is consistent, since a `get()` in JCache returns `null` only in the case that no entry is present. The JCache behavior is also useful, since nulls from the loader pass through transparently. But as soon as nulls are passed through regularly, the cache is rendered useless, since a `null` from the loader means "no caching". This will be unnoticed during development but will lead to performance trouble in production.

In cache2k there are different options when `null` comes into play. A failure by default will, hopefully, lead to an explicit choice for the best option.

# Chapter 9. Exceptions and Resilience

In a read through configuration the cache can tolerate temporary loader failures. The write through does not provide resilience capabilities at the moment, which means a writer exception will always be propagated to the cache client.

## 9.1. Behavior

The default behavior depends on the general expiry (`expiryAfterWrite` or `eternal`) setting of the cache.

### 9.1.1. No Expiry

If no expiry is specified or `eternal (true)` is specified, all exceptions will be propagated to the client. The loader will be called immediately again, if the key is requested again.

### 9.1.2. With Expiry

When an expiry time is specified, the cache also enables the resilience features.

If a load yields an exception and there is data in the cache: The exception will not be propagated to the client, and the cache answers requests with the current cache content. Subsequent reload attempts that yield an exception, will also be *suppressed*, if the time span to the first exception is below the resilience duration setting.

If the loader produces subsequent exceptions that is longer than the resilience duration, the exception will be *propagated*. The resilience duration can be set with the parameter `resilienceDuration`, if not set explicitly it is identical to the `expiryAfterWrite` time span.

## 9.2. Retry

After an exception happens the cache will do a retry to load the value again. The retry is started after the configured retry interval (`retryInterval`), or, if not explicitly configured after 5% of the resilience duration. The load is started when the client accesses the value again, or the cache is doing this by itself if `refreshAhead` is enabled.

To keep the system load in limits in the event of failure, the duration between each retry increases according to an *exponential backoff* pattern by the factor of 1.5. Each duration is further randomized between one half and the full value. For example, an `expiryAfterWrite` set to 200 seconds will lead to an initial retry time of 10 seconds. If exceptions persist the retry time will develop as follows:

Table 2. Retry intervals with exponential backoff starting at 10 seconds

base duration	randomized duration range
10 seconds	5 - 10 seconds
15 seconds	7.5 - 115 seconds

base duration	randomized duration range
22.5 seconds	11.25 - 22.5 seconds
33.75 seconds	16.875 - 33.75 seconds

When reaching the configured expiry the cache will retry at this time interval and not increase further. The start retry interval and the maximum retry interval can be specified by `retryInterval` and `maxRetryInterval`.

## 9.3. Exception Propagation

If an exception cannot be suppressed, it is propagated to the client immediately. The retry attempts follow the same pattern as above.

When propagated, a loader exception is wrapped and rethrown as `CacheLoaderException`. A loader exception is potentially rethrown multiple times, if the retry time is not yet reached. In this situation a rethrown exception contains the text `expiry=<timestamp>`. This behavior can be customized by the `ExceptionHandler`.

## 9.4. Invalidating

An application may need to invalidate a cache entry, so the cache will invoke the loader again the next time the entry is requested. How the value should be invalidated depends on the usage scenario and whether availability or consistency has to be priority.

To be able to use the resilience features and increase availability in the event of failure the method `expireAt` should be preferred for invalidation. See the detailed discussion in the loading chapter.

## 9.5. Entry Status and `containsKey`

In case an exception is present, the method `containsKey` will return `true`, the methods `putIfAbsent` and `computeIfAbsent` act consequently. This means `putIfAbsent` can not be used to overwrite an entry in exception state with a value.

To examine the state of an entry, e.g. whether it contains a value or exception, the method `Cache.peekEntry` can be used. To examine the state of an entry and modify it, the entry processor can be used.

## 9.6. Configuration options

To customize the behavior the following options exist.

### *suppressExceptions*

Default is true. If set to false, do not suppress exceptions.

### *expireAfterWrite*

Time duration after insert or updated an cache entry expires

### *resilienceDuration*

Time span the cache will suppress loader exceptions if a value is available from a previous load.  
Defaults to `expiredAfterWrite`

### *mayRetryInterval*

The maximum time interval after a retry attempt is made. Defaults to `resilienceDuration`

### *retryInterval*

Initial time interval after a retry attempt is made. Defaults to 10% of `mayRetryInterval`, or a minimum of 2 seconds.

### *resiliencePolicy*

Sets a custom resilience policy to control the cache behavior in the presence of exceptions

### *exceptionPropagator*

Sets a custom behavior for exception propagation

### *refreshAhead*

Either the option `refreshAhead` or `keepDataAfterExpired` must be enabled to do exception suppression if an expiry is specified

### *keepDataAfterExpired*

Either the option `refreshAhead` or `keepDataAfterExpired` must be enabled to do exception suppression if an expiry is specified

## 9.7. Examples

### 9.7.1. No expiry

Values do not expire, exceptions are not suppressed. After an exception, the next `Cache.get()` will trigger a load.

```
Cache<Integer, Integer> c = new Cache2kBuilder<>() {}  
    .eternal(true)  
    /* ... set loader ... */  
    .build();
```

### 9.7.2. Expire after 10 minutes

Values expire after 10 minutes. Exceptions are suppressed for 10 minutes as well, if possible. A retry attempt is made after 1 minute. If the cache continuously receives exceptions for a key, the retry intervals are exponentially increased up to a maximum interval time of 10 minutes.



```

Ê Cache<Integer, Integer> c = new Cache2kBuilder<>() {}
Ê   .expireAfterWrite(10, TimeUnit.MINUTES)
Ê   .keepDataAfterExpired(true)
Ê   /* ... set loader ... */
Ê   .build();

```

### 9.7.3. Reduced suppression time

Expire entries after 10 minutes. If an exception happens we do not want the cache to continue to service the previous (and expired) value for too long. In this scenario it is preferred to propagate an exception rather than serving a potentially outdated value. On the other side, there may be temporary outages of the network for a maximum of 30 seconds we like to cover for.

```

Ê Cache<Integer, Integer> c = new Cache2kBuilder<Integer, Integer>() {}
Ê   .expireAfterWrite(10, TimeUnit.MINUTES)
Ê   .resilienceDuration(30, TimeUnit.SECONDS)
Ê   .keepDataAfterExpired(true)
Ê   /* ... set loader ... */
Ê   .build();

```

### 9.7.4. Cached exceptions

No suppression, because values never expire. The only way that a reload can be triggered is with a reload operation. In this case we do not want suppression, unless specified explicitly. The loader is not totally reliable, or a smart developer uses an exception to signal additional information. If exceptions occur, the cache should not be ineffective and keep exceptions and defer the next retry for 10 seconds. For requests between the retry interval, the cache will rethrow the previous exception. The retry interval does not increase, since a maximum timer interval is not specified.

```

Ê Cache<Integer, Integer> c = new Cache2kBuilder<Integer, Integer>() {}
Ê   .eternal(true)
Ê   .retryInterval(10, TimeUnit.SECONDS)
Ê   /* ... set loader ... */
Ê   .build();

```

## 9.8. Custom resilience policy

By registering a custom implementation of the `ResiliencePolicy` it is possible to implement a special behavior that is used to determine the cache duration of an suppressed or cached an exception . Use the existing implementation as an example and starting point.

## 9.9. Debugging

The cache has no support for logging exceptions. If this is needed, it can be achieved by an adaptor of the `CacheLoader`.

The statistics expose counters for the total number of received load exceptions and the number of suppressed exception.

# Chapter 10. Event Listeners

The cache generates events when an entry is inserted or updated.

## 10.1. Listening to Events

Event listeners can be added via the cache builder, for example:

```
Cache2kBuilder.of(Integer.class, Integer.class)
    .addListener(new CacheEntryCreatedListener<Integer, Integer>() {
        @Override
        public void onEntryCreated(final Cache<Integer, Integer> cache,
                                   final CacheEntry<Integer, Integer> entry) {
            System.err.println("inserted: " + entry.getKey());
        }
    });
```

Different listener types are available for insert, update, removal and expiry. There is currently no possibility to listen to an eviction.

Listeners are executed synchronously, meaning the cache operation will not complete until all listeners are run. The expiry event is always asynchronous.



It is illegal to mutate cache values inside the listeners.

## 10.2. Async Listeners

Listeners will be executed asynchronously when added with `addAsyncListener()`. By default a shared unbounded executor is used. A custom executor can be set via `asyncListenerExecutor`.



The cached value is not copied during the cache operation. If a value instance is mutated after it was handed over to the cache, asynchronous listeners may not see the value as it was present during the cache operation.

# Chapter 11. Configuration via XML

cache2k supports an XML configuration file. The configuration file can contain additional settings that should not be "buried" inside the applications code.

## 11.1. Using the XML Configuration

When `cache2k-all` artifact is used the configuration via XML is included, otherwise the artifact `cache2k-xml-configuration` must be added as dependency. In a Java SE environment the default SAX parser is used. In an Android environment the XML pull parser is used. There are no additional dependencies to other libraries.

If only one cache manager is used, a configuration file can be put at `/cache2k.xml` in the class path. Here is an example:

```

<cache2k>
  <!-- An configuration example for the documentation -->
  <version>1.0</version>
  <defaultManagerName>default</defaultManagerName>
  <skipCheckOnStartup>true</skipCheckOnStartup>
  <properties>
    <user>
      <smallCacheCapacity>12_000</smallCacheCapacity>
      <userHome>${ENV.HOME}</userHome>
    </user>
  </properties>
  <defaults>
    <cache>
      <entryCapacity>100_000</entryCapacity>
    </cache>
  </defaults>
  <templates>
    <cache>
      <name>regularExpiry</name>
      <expireAfterWrite>5m</expireAfterWrite>
    </cache>
    <cache>
      <name>lessResilient</name>
      <resilienceDuration>1m</resilienceDuration>
    </cache>
  </templates>
  <caches>
    <cache>
      <name>users</name>
      <entryCapacity>${TOP.properties.user.smallCacheCapacity}</entryCapacity>
      <loader>
        <byClassName>
          <className>org.example.MyLoader</className>
        </byClassName>
      </loader>
    </cache>
    <cache>
      <name>products</name>
      <include>regularExpiry, lessResilient</include>
    </cache>
  </caches>
</cache2k>

```

## 11.2. Combine Programmatic and XML Configuration

The XML configuration can provide a default setup and a specific setup for a named cache. The specific setup from the XML configuration is applied after setup from the builder, thus overwriting any defaults or settings via the builder from the program code.

When a cache is created via the builder it needs to have mandatory properties on the programmatic level:

¥ Type for key and value

¥ Cache name

¥ Manager with classloader, if another manager or classloader should be used

It is recommended to do settings in the builder, that belong to the application code, for example:

`eternal(true)`

if no expiry is needed, in case the values are immutable or never change

`eternal(false)`

if entries need to expire, but no specific time is set on programmatic level

`permitNullValues(true)`

if the application needs to store `null`s in the cache

`storeByReference(true)`

if the application relies on the fact that the objects are only stored in the heap and not copied.

For example, the cache is created in the code with:

```
Cache<String, String> b =
    new Cache2kBuilder<String, String>(){}
        .name("various")
        .eternal(true)
        .permitNullValues(true)
        .build();
```

In the configuration file only the capacity is altered:

```
<cache>
  <name>various</name>
  <entryCapacity>10K</entryCapacity>
</cache>
```

## 11.3. Reference Documentation

### 11.3.1. File Location

The configuration for the default cache manager (`CacheManager.getInstance()`) is expected in the class path at `/cache2k.xml`. If a different class loader is used to create the cache manager, that is used to look up the configuration. If multiple cache managers are used, each cache manager can have its own configuration, which is looked after at `/cache2k-${managerName}.xml`.

### 11.3.2. Options

Some options control how the configuration is interpreted.

#### *version*

Version which controls how the configuration is interpreted. Needed for possible future changes. Always **1.0** at the moment.

#### *defaultManagerName*

Set another name for the default cache manager, default is **"default"**.

#### *ignoreAnonymousCache*

If **true**, allows cache without name. If a cache has no name a special configuration cannot be applied. The default is **false**, enforcing that all caches are named on the programmatic level.

#### *skipCheckOnStartup*

Do not check whether all cache configurations can be applied properly at startup. Default is **false**.

### 11.3.3. Default Configuration

A default configuration may be provided in **default.ts.cache** (see example above). The defaults will be used for every cache created in the cache manager.

### 11.3.4. Templates

Multiple template configurations can be provided under **templates**. Templates have a name. In the cache configuration, a template can be included via **include**. Multiple templates can be included when separated with comma.

Templates can be used for other configuration sections as well.

### 11.3.5. Parameters

The values may contain the parameters in the style **\${scope.name}**. A parameter name starts with a scope. Predefined scopes are:

#### *ENV*

environment variable, e.g. **\${ENV.HOME}** is the user home directory.

#### *TOP*

references the configuration root, e.g. **\${TOP.caches.flights.entryCapacity}** references the value of the **entryCapacity** of the cache named **flights**.

#### *PROP*

a Java system property, e.g. **\${PROP.java.home}** for the JDK installation directory.

The scope prefix can also reference a parent element name. If the scope prefix is empty an value at the same level is referenced.

A configuration can contain user defined properties in the `properties.user` section.

### 11.3.6. Primitive Types

The configuration supports basic types.

*Table 3. Supported Types in XML Configuration*

type	example	description
boolean	<code>true</code>	Boolean value either true or false
int	<code>4711</code>	Integer value
long	<code>20_000Mi B</code>	Long value with optional suffixes (see below)
String	<code>alice</code>	A string value

For additional convenience the long type supports a unit suffix:

suffix	value
KiB	1024
MiB	1024^2
GiB	1024^3
TiB	1024^4
k	1000
M	1000^2
G	1000^3
T	1000^4
s	1000
m	1000*60
h	1000*60*60
d	1000*60*60*24

A long value may also contain the character '\_' for structuring. This character is ignored. Example: `12_000_000`. The unit suffix is intended to make the configuration more readable. There is no enforcement that a unit actually matches with the intended unit of the configuration value.

### 11.3.7. Sections

The cache configuration may contain additional sections. At the moment only the section `jcache` is available which tweaks JCache semantics.



### 11.3.8. Customizations

Customizations, for example, loaders, expiry policy and listeners, may be configured. The simplest method is to specify a class name of the customization that gets created when the cache is build (see also example above).

```
É <loader>
É   <byClassName>
É     <className>org.example.MyLoader</className>
É   </byClassName>
É </loader>
```

It is also possible to implement an own `CustomizationSupplier` which can take additional parameters for additional configuration of the customization. In this case the `type` element is used to specify the supplier class.

```
É <loader>
É   <bean>
É     <type>org.example.LoaderSupplier</type>
É     <!-- Additional bean properties to set on the supplier follow. -->
É     <database>jdbc://...</database>
É   </bean>
É </loader>
```

## 11.4. Rationale

Most of the configuration processing is not limited to XML. The general structure of the configuration can be represented in YAML or JSON as well. This is why we do not use any XML attributes. Readers for other formats can implement the `ConfigurationTokenizer`.

The configuration code is mostly generic and uses reflection. In case new properties or configuration classes are added, there is no need to update configuration code. This way the extra code for the XML configuration keeps small. Eventually we will separate the configuration into another project so it can be used by other applications or libraries with their configuration beans.

The structure adheres to a strict scheme of `container.type.property.type.properties [ .type.property É ]`. This simplifies the processing and also leaves room for extensions.

# Chapter 12. Logging

The log output of cache2k is very sparse, however, some critical information could be send to the log, so proper logging configuration is essential.

## 12.1. Supported Log Infrastructure

cache2k supports different logging facades and the JDK standard logging. The supported mechanisms include:

- ¥ SLF4J

- ¥ Apache Commons Logging

- ¥ JDK standard logging

The availability is evaluated in the above order and the first match is picked and used exclusively for log output. E.g. if the slf4j-api is present, the log output will be directed to SLF4J. This scheme should have the desired results, without the need of additional configuration of the used logging facade.

## 12.2. Wiring a Custom Log Target

In case none of the above logging infrastructure can be used the service provider interface `org.cache2k.core.util.LogFactory` can be implemented and provided via the `ServiceLoader` mechanism.

# Chapter 13. Statistics

cache2k collects statistics by default and exposes them via JMX.

## 13.1. JMX Metrics

When using the `cache2k-all` artifact for runtime, JMX support is available. Otherwise the additional `cache2k-server-side` artifact needs to be added as dependency.

The management beans are registered with the platform MBean server. The object name of a cache follows the pattern `org.cache2k:type=Cache,manager=<managerName>,name=<cacheName>`, the object name of a cache manager follows the pattern `org.cache2k:type=CacheManager,name=<managerName>`.

More detailed information can be found in the API documentation:

¥ [MXBean for the cache](#)

¥ [MXBean for the cache manager](#)

### 13.1.1. Conflicting Manager Names in JMX

Multiple cache managers with the identical name may coexist under different class loaders. With JMX enabled, this will lead to identical JMX objects and refusal of operation. A workaround is to use unique cache manager names. The name of the default manager, which is usually "default" can be changed via the XML configuration, JNDI or a call to `CacheManager.setDefaultName` early in the application startup.

## 13.2. `toString()` Output

The output of the `toString()` method is extensive and also includes internal statistics. Example:

```
Cache{database}(size=50003, capacity=50000, get=102876307, miss=1513517, put=0,
load=4388352, reload=0, heapHit=101362790, refresh=2874835, refreshFailed=42166,
refreshedHit=2102885, loadException=0, suppressedException=0, new=1513517,
expire=587294, remove=8156, clear=0, removeByClear=0, evict=868064, timer=3462129,
goneSpin=0, hitRate=98.52%, msecs/load=0.425, asyncLoadsStarted=2874835,
asyncLoadsInFlight=0, loaderThreadsLimit=8, loaderThreadsMaxActive=8, created=2016-12-
02 03:41:34.367, cleared=-, infoCreated=2016-12-02 14:34:34.503,
infoCreationDeltaMs=21, collisions=8288, collisionSlots=7355, longestSlot=5,
hashQuality=83, noCollisionPercent=83, impl=HeapCache,
eviction0(impl=ClockProPlusEviction, chunkSize=11, coldSize=749, hotSize=24252,
hotMaxSize=24250, ghostSize=12501, coldHits=11357227, hotHits=38721511,
ghostHits=294065, coldRunCnt=444807, coldScanCnt=698524, hotRunCnt=370773,
hotScanCnt=2820434), eviction1(impl=ClockProPlusEviction, chunkSize=11, coldSize=778,
hotSize=24224, hotMaxSize=24250, ghostSize=12501, coldHits=11775594, hotHits=39508458,
ghostHits=283324, coldRunCnt=423258, coldScanCnt=674762, hotRunCnt=357457,
hotScanCnt=2689129), evictionRunning=0, keyMutation=0, internalException=0,
integrityState=0.17.a6c585b1)
```

#

Do to the internal organization of the internal data structures retrieving statistics is a very costly operation, since it involves scanning through the cache content.

#

The output of `toString()` may change between releases. It should give valuable information for debugging, but it shouldn't be consumed by application logic.

## 13.3. Accuracy, Overhead and Performance

The statistics gathering has a very low operational overhead and is enabled by default. Obtaining the statistics is a costly operation, since the cache needs to aggregate counters of all cache entries.

Since high poll frequencies on the statistic values may produce a high system load, the cache has countermeasures and only aggregates new data after some time has passed. The `toString` output and the JMX bean contains the timestamp `infoCreated` and the value `infoCreationDeltaMs` to get more insight in this behavior.

The switch `disableStatistics` disables some statistic values that have a significant overhead, for example the counter for misses or updates.

The metric `getCount` is not totally accurate and may count fewer accesses depending on the amount of concurrency. The implementation uses a dirty counter per cache entry. The counter is used for the eviction as well, thus the access statistics are a by-product. The error is relatively small. Future releases may contain the option to obtain precise statistics.

## 13.4. Internal Statistics

The cache implementation has a lot statistics counters, that are exposed in an internal interface. This can be revealed via `((InternalCache) cache).getInfo()` and needs the `cache2k-core` artifact in the compile scope. However, internal interfaces may change from release to release.

## 13.5. Noteworthy Metrics

Some special metrics need some more explanation.

### 13.5.1. Hash Quality

Value between 100 and 0 to help evaluate the quality of the hashing function. 100 means perfect. This metric takes into account the collision to size ratio, the longest collision size and the collision to slot ratio. The value reads 0 if the longest collision size gets more then 20.

The number of collisions and the longest size of a collision slot is also available via JMX.

### 13.5.2. Key Mutation Counter

Storing objects by reference means it is possible for the application to alter the object value after it was involved in a cache operation. In case of the key object, this means that the internal data structure of the cache will be invalid after an illegal mutation.

Within the cache eviction a key mutation is detected. The counter is incremented and a warning is written to the log. Per cache the warning is only logged once.

Whenever `keyMutationCount` is non-zero, check and correct your application.

### 13.5.3. Alert

The cache as well as the cache manager have a single value health status which is intended for systems monitoring. The value 0 means everything is okay, 1 means warning, 2 means failure.

# Chapter 14. Android

cache2k is compatible with Java 6 and Android. Regular testing is done against API level 16.

## 14.1. Usage

Include the following dependencies:

```
Ê <dependency>
Ê   <groupId>org.cache2k</groupId>
Ê   <artifactId>cache2k-api</artifactId>
Ê   <version>${cache2k-version}</version>
Ê </dependency>
Ê <dependency>
Ê   <groupId>org.cache2k</groupId>
Ê   <artifactId>cache2k-core</artifactId>
Ê   <version>${cache2k-version}</version>
Ê   <scope>runtime</scope>
Ê </dependency>
```

## 14.2. XML Configuration

The XML configuration is usable for Android. The additional library needs to be included:

```
Ê <dependency>
Ê   <groupId>org.cache2k</groupId>
Ê   <artifactId>cache2k-xml-configuration</artifactId>
Ê   <version>${cache2k-version}</version>
Ê   <scope>runtime</scope>
Ê </dependency>
```

# Chapter 15. JCache

cache2k supports the JCache API standard. The implementation is compatible to the JSR107 TCK.

## 15.1. Maven Dependencies

Additionally to the normal cache2k dependencies the following dependencies need to be added:

```
Ê <dependency>
Ê   <groupId>org.cache2k</groupId>
Ê   <artifactId>cache2k-all</artifactId>
Ê   <version>${cache2k-version}</version>
Ê   <scope>runtime</scope>
Ê </dependency>
Ê <dependency>
Ê   <groupId>org.cache2k</groupId>
Ê   <artifactId>cache2k-jcache</artifactId>
Ê   <version>${cache2k-version}</version>
Ê   <scope>runtime</scope>
Ê </dependency>
Ê <dependency>
Ê   <groupId>javax.cache</groupId>
Ê   <artifactId>cache-api</artifactId>
Ê   <version>1.1.0</version>
Ê </dependency>
```

## 15.2. Getting Started with the JCache API

Since cache2k is JCache compatible, any available JCache introduction can be used for the first steps. The following online sources are recommended:

¥ <https://dzone.com/refcardz/java-caching>

¥ <https://www.youtube.com/watch?v=EugtmOaZn9w>

## 15.3. Configuration

JCache does not define a complete cache configuration, for example, configuring the cache capacity is not possible. To specify a meaningful cache configuration, the cache2k configuration mechanism needs to be utilized.

### 15.3.1. Programmatic Configuration

To create a JCache with an additional cache2k configuration the interface `ExtendedConfiguration` and the class `MutableExtendedConfiguration` is provided. The configuration classes are available in separate API packages, that need to be included in the compile scope:

```

<dependency>
  <groupId>org.cache2k</groupId>
  <artifactId>cache2k-api</artifactId>
  <version>${cache2k-version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.cache2k</groupId>
  <artifactId>cache2k-jcache-api</artifactId>
  <version>${cache2k-version}</version>
</dependency>

```

Example usage:

```

CachingProvider p = Caching.getCachingProvider();
CacheManager cm = p.getCacheManager();
Cache<Long, Double> cache = cm.createCache("aCache", ExtendedMutableConfiguration
.of(
  new Cache2kBuilder<Long, Double>(){}
    .entryCapacity(10000)
    .expireAfterWrite(5, TimeUnit.MINUTES)
));

```

When a cache2k configuration is provided, the default behavior is not 100% JCache compatible any more, because the defaults of cache2k apply. Strict JCache behavior can be enabled again, if needed. The areas of difference are explained below:

### *Expiry*

With pure JCache configuration the expiry is set to sharp expiry by default for maximum compatibility. When the cache2k configuration is used, sharp expiry is off and need to be enabled explicitly.

### *Configuration.isStoreByValue*

When the cache2k configuration is active the parameter will be ignored, if true. Store by value semantics can be enabled again in the JCache configuration section of cache2k `JCacheConfiguration.Builder.copyAlwaysIfRequested`. See example below how to specify the additional parameters.

The JCache configuration and the cache2k configuration may have settings that control the same feature, for example expiry. In this case the two configurations need to be merged and conflicting settings have to be resolved. The policy is as follows:

### *Expiry settings*

Settings in cache2k configuration take precedence. A configured expiry policy in the standard JCache `CacheConfiguration` will be ignored if either `expireAfterWrite` or `expiryPolicy` is specified in the cache2k configuration.



## Loader and Writer

Settings in JCache configuration take precedence. If a loader or a writer is specified in the JCache `CacheConfiguration` the setting in the `cache2k` configuration is ignored.

## Event listeners

Registered listeners of both configurations will be used.

# 15.4. Control Additional JCache Semantics

The JCache implementation has additional options that control its semantics. These options are available in the `JCacheConfiguration` configuration section, which is provided by the `cache2k-jcache-api` module.

Example usage:

```
Ê CachingProvider p = Caching.getCacheProvider();
Ê CacheManager cm = p.getCacheManager();
Ê Cache<Long, Double> cache = cm.createCache("aCache", ExtendedMutableConfiguration
. of(
Ê     new Cache2kBuilder<Long, Double>() {}
Ê     .entryCapacity(10000)
Ê     .expireAfterWrite(5, TimeUnit.MINUTES)
Ê     .with(new JCacheConfiguration.Builder()
Ê         .copyAlwaysIfRequested(true)
Ê     )
Ê );
```

The example enables store by value semantics again and requests that keys and values are copied when passed to the cache or retrieved from the cache.

# 15.5. Implementation Details

## 15.5.1. Semantic Changes Since JCache 1.0

The JCache specification team has made some changes to its TCK since the original 1.0 release. The `cache2k` implementation already adheres to the latest corrected TCK 1.1.

Table 4. Corrected or Enforced JSR107 Semantics in TCK 1.1

Affected Component	JSR107 GitHub issue
<code>EntryProcessorException</code>	<a href="https://github.com/jsr107/jsr107tck/issues/85">https://github.com/jsr107/jsr107tck/issues/85</a>
Customizations may implement <code>Closeable</code>	<a href="https://github.com/jsr107/jsr107tck/issues/100">https://github.com/jsr107/jsr107tck/issues/100</a>
<code>CacheEntry.getOldValue()</code> for removed event	<a href="https://github.com/jsr107/jsr107spec/issues/391">https://github.com/jsr107/jsr107spec/issues/391</a>

Affected Component	JSR107 GitHub issue
Statistics of <code>Cache.putIfAbsent()</code>	<a href="https://github.com/jsr107/jsr107tc/issues/63">https://github.com/jsr107/jsr107tc/issues/63</a>
<code>CacheManager.getCacheNames()</code>	<a href="https://github.com/jsr107/jsr107tc/issues/87">https://github.com/jsr107/jsr107tc/issues/87</a>
<code>CacheManager.getCache()</code>	<a href="https://github.com/jsr107/jsr107spec/issues/340">https://github.com/jsr107/jsr107spec/issues/340</a>
JMX statistics	<a href="https://github.com/jsr107/jsr107tc/issues/83">https://github.com/jsr107/jsr107tc/issues/83</a>

### 15.5.2. Expiry Policy

If configured via cache2k mechanisms, the cache2k expiry settings take precedence.

If a JCache configuration is present for the expiry policy the policies `EternalExpiryPolicy`, `ModifiedExpiredPolicy` and `CreatedExpiredPolicy` will be handled efficiently. A custom implementation of the `ExpiryPolicy` will induce additional operational overhead.

The use of `TouchedExpiryPolicy` or `ExpiryPolicy.getExpiryAccess()` is discouraged. Test performance carefully before use in production.

### 15.5.3. Store by Value

If configured via cache2k mechanisms, store by value semantics are not provided by cache2k by default. Instead the usual in process semantics are provided. Applications should not rely on the fact that values or keys are copied by the cache in general.

For heap protection cache2k is able to copy keys and values. This can be enabled via the parameter `JCacheConfiguration.setCopyAlwaysIfRequested`, see the configuration example above.

### 15.5.4. Loader exceptions

cache2k is able to cache or suppress exceptions, depending on the situation and the configuration.

If an exception is cached, the following behavior can be expected:

- ¥ Accessing the value of the entry, will trigger an exception
- ¥ `Cache.containsKey()` will be true for the respective key
- ¥ `Cache.iterator()` will skip entries that contain exceptions

### 15.5.5. Listeners

Asynchronous events are delivered in a way to achieve highest possible parallelism while retaining the event order on a single key. Synchronous events are delivered sequentially.

### 15.5.6. Entry processor

Calling other methods on the cache from inside an entry processor execution (reentrant operation), is not supported. The entry processor should have no external side effects. To enable asynchronous operations, the execution may be interrupted by a `RestartException` and restarted.

### 15.5.7. `Cache.getConfiguration()`

It is not possible to retrieve the additional effective cache2k configuration with this method.

## 15.6. Performance

Using the JCache API does not deliver the same performance as when the native cache2k API is used. Some design choices in JCache lead to additional overhead, for example:

- ¥ Event listeners are attachable and detachable at runtime
- ¥ Expiry policy needs to be called for every access
- ¥ Store-by-value semantics require keys and values to be copied

## 15.7. Compliance Testing

To pass the TCK tests on statistics, which partially enforce that statistic values need to be updated immediately. For compliance testing the following system properties need to be set:

- ¥ `org.cache2k.core.HeapCache.Tunable.minimumStatisticsCreationTimeDeltaFactor=0`
- ¥ `org.cache2k.core.HeapCache.Tunable.minimumStatisticsCreationDeltaMillis=-1`

Since immediate statistics update is not a requirement by the JSR107 spec this is needed for testing purposes only.