# The Object Cache

**by Armin Waibel, Thomas Mahler**

**FIXME (arminw):**
this document is not finished yet.

## 1. Introduction

OJB was shipped with several `ObjectCache` implementations. All implementations can be found in **`org.apache.ojb.broker.cache`** package. To classify the different implementations we differ *local cache* and *shared/global cache* (we use both terms synonymous) implementations.

- Local cache implementation mean that each instance use its own object map to manage cached objects.
- Shared/global cache implementations share one (in most cases static) map to manage cached objects.

A distributed object cache implementation supports caching of objects across different JVM.

## 2. Why a cache and how it works?

OJB provides a pluggable object cache provided by the `ObjectCache` interface.

```
public interface ObjectCache
{
    /**
    * Write to cache.
    */
    public void cache(Identity oid, Object obj);

    /**
    * Lookup object from cache.
    */
    public Object lookup(Identity oid);

    /**
    * Removes an Object from the cache.
    */
    public void remove(Identity oid);

    /**
    * Clear the ObjectCache.
    */
    public void clear();
}
```

Each PersistenceBroker instance using its own `ObjectCache` instance. The `ObjectCache` instances are created by the `ObjectCacheFactory` class.

Each cache implementation holds Objects previously loaded or stored by the PersistenceBroker - dependend on the implementation.
Using a Cache has several advantages:

- It increases performance as it reduces database lookups or/and object materialization. If an object is looked up by Identity the associated PersistenceBroker instance, does not perform a `SELECT` against the database immediately but first looks up the cache if the requested object is already loaded. If the object is cached it is returned as the lookup result. If it is not



Copyright © All rights reserved.

cached a SELECT is performed.
Other queries were performed against the database, but before an object from the ResultSet was materialized the object identity was looked up in cache. If not found the whole object was materialized.

- It allows to perform circular lookups (as by crossreferenced objects) that would result in non-terminating loops without such a cache.

## 3. How to change the used ObjectCache implementation

The `object-cache` element/property can be used to specify the ObjectCache implementation used by OJB. There are three levels of declaration:

in **OJB.properties** file, to declare the standard (default) ObjectCache implementation. The declared ObjectCache implementation was initialized with default properties, it's not possible to pass additional configuration properties on this level of declaration.

```
#-------------------------------------------------------------------
# Object cache
#-------------------------------------------------------------------
# The ObjectCacheClass entry tells OJB which concrete instance Cache
# implementation is to be used.
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl
#
```

on **jdbc-connection-descriptor level** , to declare ObjectCache implementation on a per connection/user level. Additional configuration properties can be passed by using *attribute* element entries:

```
<jdbc-connection-descriptor ...>
...
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
<attribute attribute-name="timeout" attribute-value="900"/>
<attribute attribute-name="useAutoSync" attribute-value="true"/>
</object-cache>
...
</jdbc-connection-descriptor>
```

on **class-descriptor level** , to declare ObjectCache implementation on a per class level. Additional configuration properties can be passed by using *attribute* element entries:

```
<class-descriptor
class="org.apache.ojb.broker.util.sequence.HighLowSequence"
table="OJB_HL_SEQ"
>
<object-cache class="org.apache.ojb.broker.cache.ObjectCacheEmptyImpl">
</object-cache>
...
</class-descriptor>
```

**Note:**

The priority of the declared object-cache elements are:
*per class > per jdbc-connection-descriptor > standard*

E.g. if you declare ObjectCache 'DefaultCache' as standard and set ObjectCache 'CacheA' in class-descriptor for class A and class B does not declare an object-cache element. Then OJB use 'CacheA' as ObjectCache for class A and 'DefaultCache' for class B.

## 4. Shipped cache implementations

### 4.1. ObjectCacheDefaultImpl



Copyright © All rights reserved.

Per default OJB use a shared reference based `ObjectCache` implementation. It's a really fast cache but there are a few drawbacks. There is no transaction isolation, when thread one modify an object, thread two will see the modification when lookup the same object or use a reference of the same object. If you rollback/abort a transaction the corrupted objects will **not** be removed from the cache (when using PB-api, top-level api may support automatic cache synchronization). You have to do this using

```
    broker.removeFromCache(obj);

    // or (using Identity object)
    ObjectCache cache = broker.serviceObjectCache();
    cache.remove(oid);
```

by your own or enable the *useAutoSync* property (more info see below).

This implementation use `SoftReference` to wrap all cached objects. If the cached object was not longer referenced by your application but only by the cache, it can be reclaimed by the garbage collector.
As we don't know when the garbage collector reclaims the freed objects, it is possible to set a `timeout` property. So an cached object was only returned from cache if it was not garbage collected and was not timed out.

To enable this `ObjectCache` implementation

```
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="600"/>
    </object-cache>
```

Implementation configuration properties:

| **Property Key** | **Property Values** |
|---|---|
| timeout | Lifetime of the cached objects in seconds. If expired, the cached object was discarded - default was 900 sec. When set to *-1* the lifetime of the cached object depends only on GC and do never get timed out. |
| autoSync | If set *true* all cached/looked up objects within a PB-transaction are traced. If the the PB-transaction was aborted all traced objects will be removed from cache. Default is *false*.<br>**NOTE:** This does not prevent "dirty-reads" by concurrent threads (more info see above).<br>It's not a smart solution for keeping cache in sync with DB but should do the job in most cases.<br>E.g. if you lookup 1000 objects within a transaction and modify one object and then abort the transaction, 1000 objects will be passed to cache, 1000 objects will be traced and all 1000 objects will be removed from cache. If you read these objects without tx or in a former tx and then modify one object in a tx and abort the tx, only one object was traced/removed. |
| cachingKeyType | Determines how the key was build for the cached objects:<br>0 - Identity object was used as key, this was the *default* setting.<br>1 - Idenity + jcdAlias name was used as key. Useful when the same object metadata model (DescriptorRepository instance) are used for different databases (JdbcConnectionDescriptor)<br>2 - Identity + model (DescriptorRepository) was |



Copyright © All rights reserved.

| | |
|---|---|
| | used as key. Useful when different metadata model (DescriptorRepository instance) are used for the same database. Keep in mind that there was no synchronization between cached objects with same Identity but different metadata model.<br>3 - all together (Idenity + jcdAlias + model) |

**Recommendation:**
If you take care of cache synchronization and be aware of dirty reads, this implementation is useful for read-only or less update centric classes.

### 4.2. ObjectCachePerBrokerImpl

This local cache implementation allows to have dedicated caches per PersistenceBroker instance. All calls are delegated to the cache associated with the current broker instance. When the broker

- does commit a transaction
- does abort/rollback a transaction
- was closed (returned to pool)

the cache was cleared. So no dirty reads will occur, because each thread use it's own PersistenceBroker instance. No corrupted objects will be found in cache, because the cache was cleared after use.

### 4.3. ObjectCacheJCSImpl

A shared `ObjectCache` implementation using a JCS region for each classname. More info see turbine-JCS.

### 4.4. ObjectCacheEmptyImpl

This is an 'empty' ObjectCache implementation. Useful when caching was not desired.

**Note:**
This implementaion does not support *circular References*. Be careful when using this implementaion with references (this may change in further versions).

### 4.5. ObjectCacheOSCacheImpl

A implementation using OpenSymphony's OSCache. More info see in Clustering HOWTO.

### 4.6. More implementations ...

Additional *ObjectCache* implementations can be found in *org.apache.ojb.broker.cache* package.

## 5. Distributed ObjectCache?

If OJB was used in a clustered enviroment it is mandatory to distribute all shared cached objects across different JVM. More information how to realize such a cache see here.

## 6. Implement your own cache

The OJB cache implementations are quite simple but do a good job for most scenarios. If you need a more sophisticated cache (e.g. with MRU memory management strategies) you'll write your own implementation of the interface `ojb.broker.cache.ObjectCache`.
Integration of your implementation in OJB is easy since the object cache is a pluggable component. All you have to do, is to declare it in the `OJB.properties` file by setting the `ObjectCacheClass` property.



Copyright © All rights reserved.

**Note:**

Of course we interested in your solutions! If you have implemented something interesting, just contact us.

## 7. CacheFilter feature

### What does cache filtering mean

TODO

### Default CacheFilter implementations

TODO

### Implement your own filter

TODO

## 8. Future prospects

TODO



Copyright © All rights reserved.