Functional Specification for Object Caching Service for Java (OCS4J), 2.0

Author: Jerry Bortvedt

Project Manager: jerry.bortvedt@oracle.com

Table of Contents

1. Project Overview

The Internet in its most basic form is just a series of requests and responses. A client sends a request to a server, the server creates a response and returns it to the user. To service these requests, a server must manage information and executable objects that fall into 3 basic categories, objects that never change, objects that are different with every request and everything in between. Java is well equipped to handle the first two cases but offers little help for the third. If the object never changes, create a static object when the server is initialized. If the object is unique to every request, create a new object each time. For everything in between, objects or information that can change and is shared across requests, between users or between processes, there is the "Object Caching Service for Java".

The Object Caching Service for Java (OCS4J), allows applications to share objects across requests, across users and coordinates the life cycle of the objects across processes. This system can manage any Java object. All objects are located by name and can be shared by all threads in a process. Creation of objects is handled by a user defined loader object. Creations are coordinated by the caching system to avoid creating an object unnecessarily. Objects can be invalidated explicitly by the application, by associating a "time to live" or "idle time" with an object or if the capacity of the caching system has been reached (this is a configurable value), objects that have not been used recently will be removed by the caching system. When an object is removed from the memory cache by the system, if may be spooled to disk. The spooling decision is based on the user defined attribute associated with the object. A callback method can be registered with the cache to be called when an object is invalidated or removed from the system. Objects within the cache can be viewed as individual objects or as a group of related objects. Objects are updated by creating a new version of the object. This allows access to objects without requiring any read locks.

For simplicity, availability, and performance, the object cache is specific to each process. There is no centralized control of object creation within the caching system. However, there is coordination of updates and invalidations between processes. If an object is explicitly invalidated or update in one process, the name and associated information is broadcast to all other instances of the cache. This allows the entire system of processes to stay synchronized without the overhead of centralized control.

2. Concepts

2.1. Description

The primary purpose of the caching service is to improve server performance by managing static and non-static java objects. The performance gain is from reducing the number of trips to the database or other external sources of information, avoiding the cost of repeatedly recreating objects, sharing objects between threads in a process and, when possible between processes, and efficient use of process resources.

The object caching service is designed for general application use. All objects within the cache are accessed by name. The name can be any Java object, however the Object used must override the Object.equals and Object.hashCode methods. A String object would typically be used. This makes it easy for various applications within a process to share an object without having to define common global structures. The caching service doesn't impose a structure on objects being cached. The name, structure, type and original source of the object are all defined by the application.

To maximize system resources, all objects within the cache are shared. However, access to cached objects is not serialized by access locks, allowing for a high level of concurrent access. Objects managed by the cache remain in the cache as long as they are being referenced. When an object is invalidated or updated the invalid version of the object will remain in the cache as long as there are references to that particular version of the object. It is thus possible to have multiple versions of an object in the cache at the same time, however, there is never more than 1 valid version of the object. The old or invalid versions of an object are only visible to applications that had references to the version before it was invalidated. If an object is updated a new copy of the object is created in the cache and the old version is marked as invalid. An object implementing the CacheEventListener interface may be associated with a cached object. When the cached object is invalidated, the CacheEventListener.handleEvent() method is called.

Since objects are shared in the cache, they should not be modified directly. A private copy of the object should be created, modified, then placed back into the cache using the replace method.

Objects within the cache can be declared as local or distributed. For local objects, updates and invalidations are not propagated outside of the process. For distributed objects, the caching service provides a mechanism for maintaining object consistency across a distributed server. This allows for both distributed update and distributed object invalidation. If an application allows updates from multiple locations, the cache can synchronize the updates across a distributed server. The caching service provides a mechanism for applications to achieve various levels of consistency between objects outside the cache and their cached copies. See the section on distributed cache consistency for more detail.

Cached objects can be invalidated explicitly by an application or automatically by the cache based on a specified "time to live" or "idle time". When an application explicitly invalidates an object, the invalidation will be propagated based on the local/distributed attribute of the object. Time based invalidation only

affects the local copy of the object, it is never propagated to other caches.

Valid objects remain in the cache as long as there is space available. When the cache is nearing its capacity, unreferenced valid objects will be discarded from the cache based on their usage patterns. Memory objects may be spooled to the disk cache rather than being discarded. Objects in the disk cache may be removed as well to meet space restrictions, however, the disk cache is presumably much larger than the memory cache so objects will be less likely to be purged. Invalid objects are removed from both the memory and disk cache when their reference count reaches zero.

2.1.1. Object Types

To help applications organize objects within the cache, five general categories of objects are defined, regions, memory objects, disk objects and group objects.

2.1.1.1. Region Object

A region object is used to define a private name space within the cache. All other objects (excluding other regions) in the cache are managed within a region. A user may define as many regions as necessary for an application, although generally one per application should be sufficient. All access to the cache is based on a CacheAccess "handle" associated with a region. Region names are user defined.

2.1.1.2. Memory Objects

Memory objects are stored in process private memory. They are typically loaded from an external source such as a database or directory. Objects are loaded using an application supplied loader object so the source of the object is not controlled by the caching service. Objects in the cache are shared by all threads within a process. To update a cached object a private copy should be obtained. Once the required changes to the object have been made, the object can be placed back into the cache with the appropriate CacheAccess.replace call. If the cache is distributed, the update will be propagated based on the object's attributes.

An application can request a memory object be spooled to local disk. This is useful if the object is large or is costly to recreate and is seldom updated. Objects on disk will tend to stay in the disk cache longer than objects in memory as the disk cache is presumably much larger than the memory cache. The write to disk is done asynchronously by default to limit the impact on the loading application. An object will be written to disk either when it is removed from the memory cache by the caching system or when it is loaded. This decision is based on the attributes associated with the object.

2.1.1.3. Group Objects

Group objects can be created to associate other objects. This allows related objects to be manipulated together. Any type of object can be associated within a group, (memory, disk or group), however all members of a group must be in the same region (see section 2.2 Naming Convention). An object can belong to only one group at a time. The attributes of a group object may apply to the group as a whole or be inherited by the members of the group. See section 2.1.2, **Object Attributes**, for details. Inherited attributes can be over ridden by explicitly setting the attributes of a particular member. Objects in a group can be invalidated individually or as a group. Invalidating a group will

invalidate all non-group objects within the group. If there are other group objects within the group the invalidation will cascade to all sub-objects. Groups can be invalidated explicitly or with a "time to live" attribute. Explicit invalidation of groups are propagated to other caches by the group name so it is important to group objects in the same way in all caches. There is currently no concept of a group update. Members of a group need to be updated individually.

Group objects must be explicitly created before objects can be associated in the group. Groups are not implicitly created. When a group is invalidated, all non-group objects within the group are invalidated and subject to removal from the cache when the reference count is zero. The group objects will remain valid in the cache so they are available when new objects are loaded. A group can be removed with the destroy command. Destroying a group will destroy/invalidate all members of the group including the group itself.

A group is typically used to associate objects that should be invalidated together or to associate object with a common set of attributes.

2.1.1.4. Disk Objects

Disk objects are stored on a local disk and accessed directly from the disk by the application. Disk objects may be shared by all server processes on a node or be local to a particular process based on the local/distributed attribute of the object. They are managed in the same way as memory objects. That is they can be invalidated explicitly, with a time to live or idle time, they can be updated by obtaining a private copy of the file and objects not being referenced may be removed from the cache if space is required. Distributed objects (objects not marked as local to a process) on disk (including spooled memory objects) will survive process termination if the validity of the object can be confirmed when the server becomes available again. The validity of an object is based on the "time to live" value. If no time to live is specified the object is assumed to be valid.

2.1.1.5. StreamAccess Objects

A streamAccess object is accessed by a user as a stream, loaded as an OutputStream and read as an InputStream. The stream interface is a convenient way for some objects to be accessed by the user. This also gives the cache more latitude as to how the object is stored. Smaller objects can be stored in memory while larger objects can be streamed directly from disk. The cache will determine where to store the object based on the size of the object and the capacity of the cache. The cache user doesn't need to worry about where the object is stored. Their access is always through an InputStream. All the attributes that apply to memory and disk objects also apply to a streamAccess object. If the object needs to survive process termination it should be explicitly saved to disk as there is no guarantee it will automatically be store on disk. A streamAccess object is not a mechanism to manage a stream, i.e., this could not be used to manage a socket endpoint. InputStream and OutputStream objects are being used as an access method to fix sized (potentially very large) objects.

2.1.2. Object Attributes

Each Object in the cache has attributes associated with it. These attributes affect how the object is managed in the cache. The following attributes can be set by the application:

- **Time to Live**, Time to live establishes the maximum amount of time an object will remain in the cache before being invalidated automatically by the cache. If this attribute is associated with a group or region, all objects in the group or region will be invalidated when the time expires. If the group or region is not destroyed (destroy group

on ttl is not set) the Time to live value will be reset. The default is no automatic invalidation.

- **Default Time to Live** The default time to live applies only to groups and regions. This attribute establishes a default value for the time to live that is applied to all objects individually within the group or region. This value can be overridden be setting the time to live on individual objects. The default is no automatic invalidation.
- **Group TTL Destroy** By default a group object is not invalidated when the associated "time to live" expires, only the members of the group are invalidated. This flag indicates that the group object should also be destroyed when the associated time to live expires.
- **Idle Time** is the amount of time an object may remain idle (with a reference count of 0) in the cache before being invalidated. If the Time to Live or Default Time to Live attribute is set, the idle time attribute is ignored. The idle time attribute is always applied to individual objects, so if this attribute is set for a region group it is interrupted as a default to be applied individually to the objects in the region or group. The default is no automatic idle time invalidation.
- **Distributed vs Local**, An object may be declared as a distributed object so updates and invalidations of the object are propagated to other caches in the site. By default all objects are local.
- **A reply is requested**, Objects are expecting a reply from remote caches when a request for object update or invalidation has completed. This flag should be set when a high level of consistency is required between cached objects. By default no reply is sent. If the local attribute is set, the reply attribute is ignored.
- **Synchronize updates**, If updates can occur at multiple locations within a site, updates may need to be synchronized. See the section on Distributed consistency for details. By default updates are not synchronized.
- **Don't flush the object if updates could be missed**, In a distributed environment it is possible for a cache to be isolated from other caches in the site for short periods of time. If an object is updated frequently, updates or invalidations could be missed causing the cache object to become stale and inconsistent with other caches. An application can determine whether the object should be flushed from the cache in this situation or whether it is safe to allow it to remain cached. By default the object will be flushed from the cache. This attribute is ignored if the object is marked as local or if a time to live has been established.
- **Version number**, An application may set a version number for each instance of an object in the cache. The version number is available for application convenience and verification. It is not used by the caching system. By default the version number is 0.
- **Spool to disk**, A memory object should be stored on disk rather than being lost when the cache system removes it from memory to regain space. This attribute only applies to memory objects. An object must be serializable to be spooled. If the object is distributed it will survive the death of the process that spooled it. Local objects are only accessible by the process that spooled them, therefore if the process dies the object on disk will not be available to subsequent processes.
- **Original** The original attribute is intended to be used to indicate objects that have been created by the application in the cache rather than loaded from an external source. Original objects will not be removed from the cache when the reference goes to zero. If allowed by the application, the object may be spooled to disk when not being referenced. Original objects must be explicitly destroyed (invalidated) by the application when they are no longer useful. These objects may be propagated to other caches using the update functionality. At remote servers the object can be considered read only by declaring the object synchronized for update (see the section on Distributed Cache Consistency). In this case only the process that created the object (the owner) can invalidate or update the object. The "ownership" of an object can be transferred from one process to another.

The object attributes are set when the object is created or loaded into the cache. Attributes are inherited from the region or group. These attributes can be overridden by explicitly setting the attributes of the object. If no attributes are set for the object either explicitly or through inheritance then the default values are used. By default the object is local with on time to live or idle time associated with it. The object will not be spooled, is not synchronized and is not original. Not all attributes have meaning in all situations. The following table describes which attributes are valid for each object type.

	Region	Group	Memory	StreamAccess	Disk
Time To Live	Applies to the region as a whole	Applies to the group as a whole	Applies to the memory object	Applies to the streamAccess object	Applies to the disk object
Default Time to Live	Default value applied to individual objects in the region	Default value applied to individual objects in the group	The same as Time to Live. TTL takes precedence if both set	The same as Time to Live. TTL takes precedence if both set	The same as Time to Live. TTL takes precedence if both set
Group destroy on TTL	Region will be destroyed when TTL expires	Group will be destroyed when TTL expires	Doesn't apply	Doesn't apply	Doesn't apply
idle time	Applies individually to objects in the region	Applies individually to objects in the group	Applies to the memory object	Applies to the streamAccess object	Applies to the disk object
Distribute flag	Default value applied to individual objects in the region	Default value applied to individual objects in the group	Applies to the memory object	Applies to the streamAccess object	Applies to the disk file. File is associated with a process
Reply flag	Default value applied to individual objects in the region	Default value applied to individual objects in the group	Applies to the memory object (only meaningful if the object is distributed)	Applies to the streamAccess object (only meaningful if the object is distributed)	Applies to the disk object (only meaningful if the object is distributed)
Flush if net down	Default value applied to individual objects in the region	Default value applied to individual objects in the group	Applies to the memory object (only meaningful if the object is distributed)	Applies to the streamAccess object (only meaningful if the object is distributed)	Applies to the disk object (only meaningful if the object is distributed)
Synchronize updates	Default value applied to individual objects in the region	Default value applied to individual objects in the group	Applies to the memory object	Applies to the streamAccess object	Applies to the disk object
Spool to Disk	Default value applied to	Default value applied to	Applies to the memory object	Applies to the streamAccess	Doesn't apply

	individual objects in the region	individual objects in the group		object	
Original	Default value applied to individual objects in the region	Default value applied to individual objects in the group	Applies to the memory object	Applies to the streamAccess object	Applies to the disk object
CacheEvent Listener	Applies to the region as a whole not to objects in the region	Applies to the group as a whole not to objects in the group	Applies to the memory object	Applies to the streamAccess object	Applies to the disk object

2.2. Naming Convention

The cache is divided into user defined regions. Each region is identified by name and defines a unique name space. Within each region all objects are referenced by name. The combination of the region and the object name must uniquely identify an object. Thus region names must be unique from other region names and all objects within a region must be uniquely named relative to the region. As described above, objects may be associated into groups. The group name is **not** however used in identifying individual objects. A group should be thought of as defining a set or collection of objects that have something in common. It does not define a hierarchical name space. Object type is also not used to distinguish objects, i.e., within a region there could not be a group "foo" and a memory object "foo".

For individual objects in the cache, any Java Object may be used as the name. This "name" object must override the the Object.equals and Object.hashCode methods. The equals method should not rely on comparing reference ids. (addresses). The instance of the name object in the cache may not be the same as that of the requester. In most cases a String object would be used. Region and group names are currently restricted to Strings. Any legal string character can be used in the region name or group name. There are no restrictions as to the length of the names, however excessively long names will add to the cost of object lookups and stores.

2.3. Distributed Cache Consistency

The caching service maintains cache-to-cache consistency using a broadcast reply mechanism for object invalidation and update. When an application invalidates or updates an object, the caching service broadcasts the invalidation or update to all caches. To insure consistency of a cached object across caches the application can request a reply from all remote caches indicating the modification has completed. If a high level of consistency is required, the application should wait for the replies before committing the change to the definitive storage (the database or directory). The replies are received asynchronously so the application can do other work while waiting for the reply. If the consistency requirements are less stringent, the application can do the invalidation or update without a reply or, for invalidation, a time to live attribute can be used.

To help coordinate multiple updates within the cache, the cache manager offers a concept of object

ownership. Cached objects can be owned by a reference to the object. If an object is declared for synchronized update, the updating reference must “own” the object before the update can occur. Only one reference can own an object at a time. A reference can request ownership of an object at anytime. If the current owner is not updating the object the ownership will be transferred to the requester. If an update is underway, it must complete before ownership can be transferred. If ownership currently resides at the requesting process, the request is a local operation. Otherwise the request is broadcast to all caches in the site. Synchronized updates only synchronize between objects in the cache. There is no relationship with external data sources. Synchronized updates are not defined for group objects. Object ownership has no effect on read access to an object.

Cache updates are not inherently transactional. If an object in the cache is updated before the database update commits, then the database update fails, the cache update is not automatically rolled back.

3. Requirements

3.1. Functionality

The main purpose of the Object Caching Service is to provide an easy to use mechanism to manage instances of Java objects to improve the performance of the server using it. Maintaining a copy of an object in memory and/or on local disk can improve response time by avoiding the cost of finding or creating the object on each request, improve throughput by making more efficient use of resources and improve scalability by making more copies of the object available thus avoiding centralized accesses. To accomplish these goal the service must meet the following requirements:

- Easy to Use
- Be able to manage any Java object

  There should be no restrictions on the type of object that can be cached. (Not all objects will be able to take advantage of all cache functionality i.e., if the object can't be serialized it can't be spooled or distributed)

- Manage objects loaded from any source

  There should be no restrictions on the original source of the data being cached.

- Allow sharing of objects within a process

  Since the assumption is that objects in the cache are read mostly, they should be shared by all threads within a process, one instance per process.

- Should be able to run in a zero configuration mode and or be integrated into an existing management system.

  The basic cache functionality should be usable without any configuration.

- Object updates and invalidation should be coordinated across multiple processes

  Since most servers are multi-process the caches within each process need to be coordinated so that the cached objects meet the consistency requirements of the user.

- Be able to invalidate a collection of objects with a single operation.

  It should be possible to associate objects in the cache so they can be managed as a group. In particular, it should be possible to invalidate a group of objects with a single call.

- Manage objects on disk as well as in memory

  The cache should manage objects on disk as well as in memory. This includes spooling objects to disk as well as managing objects that are typically read directly from disk.

3.2. Performance

The cache can be used in many different scenarios, managing many different types of objects. The exact gain in performance will vary significantly depending on the cost of creating or acquiring the object and what the ratio of reads to writes is . The more costly the object is to create and the more reads per write, the greater the benefit the cache can provide. One of the most common uses will likely be managing objects created from data in the database. Using a database request as the standard, the goal is to have the retrieval

from the cache to be many times faster (100x) than retrieving from a the database even in the simplest case. The simplest case being, fetching a single row from a small table. To evaluate the overhead of using the cache we will compare the cache to using the Java Hashtable class. A hash table is often used as a very simple cache mechanism for static data. The goal here is, for simple retrievels, to be within percentages (< 2x) of retrieving from a hash table. Ideally it would be nice to be as fast or faster than a simple hash table even in the simplest cases but with the significantly richer set of functionality available in the cache it may not realistic.

3.3. Availability

The cache service itself should not degrade the availability of the server using it. It can in some cases be used to improve availability. By maintaining local copies of objects, some operations can continue locally even though the original source is unavailable. It is also possible to build a state management system such as HttpSessions, on top of the cache which uses the distributed aspects of the cache to increase availability.

3.4. Scalability

Like availability, scalability of a server can be improved by using the cache to manage multiple copies of an object across multiple processes and nodes.

3.5. System/Database Management

To totally integrate the cache into a server, the servers system management will need to provide a mechanism to get and set the cache configuration values and possibly to display monitoring information. The cache will provide API's to set and retrieve this information.

3.6. Ease-of-Use

The object cache should be very easy to use. There are two aspects of usage that need to be addressed. One is the user interface to the cache, the other is how easy is it to integrate into an existing system. The interface should be easy to understand, and easy to incorporate into an application. The interface to the cache consists of a small number of Java methods and 3 to 4 Java interfaces that can be implemented by the user to more efficiently use the cache. See the API section for more detail.

The cache service itself has no configuration management or configuration file. It is intended to be used with zero configuration for development or simple servers, using default values for configuration. For larger systems it should be integrated into the existing management structure. The small number of configuration parameters needed for the cache can be added to an existing configuration file.

3.7. Usage Model

There are a wide range of potential uses for the cache. Any application that has non-static data that is shared across threads or requests could benefit from using the cache. Listed below are a few application that could benefit from the cache.

- Configuration management

  The cache is used to manage configuration information in each server process. The distributed update and invalidation features are used to keep the configuration information consistent across processes. The invalidation callback is used to notify sub systems when configuration information has changed, allowing for dynamic updates. Customized loading is used to load configuration information from configuration sources in different formats and only load the information a process is interested in .

- Portal

A portal needs to manage both user profiles and the objects available at the portal. Both can be cached. The objects can be aged out of the cache to free up space or the idle time feature can be used to remove objects not being accessed. This will simplify the management of objects as the application doesn't need constant monitor which objects are in demand at any given time. The "hot" objects will automatically be available in the cache. Objects that are expensive to create or fetch can be written to local disk and transparently retrieved as needed.

- Server Subsystems

  Server subsystems such as the servlet engine can improve server performance by pooling such things as request, response and buffer objects. The servlet objects themselves can be stored in the cache. The group invalidation feature can then be used when application reload is required. All servlets and related objects within an application can be cleaned up with a single method call.

- Response Caching

  Part or all of a response can be cached if it is applicable to more than one response. This can significantly improve response time.

3.8. Reliability

The cache should have no negative effect on the reliability of the server it is running in. It may, depending on how it is used, improve reliability by offering better management of Java objects than is currently being used.

3.9. Maintainability

The caching system will include logging facilities to help trace problems within the code. This includes the ability to "dump" the metadata for the entire cache. This should make it easy to identify any problems that should occur in the cache or by improper use of the cache. The logging facility will be written as an interface with a default implementation. This will allow the cache logging to be easily integrated with any logging service provided by the server. This logging service is intended for use within the caching service not as a general logging service for application use.

3.10. Security

In the first release security is assumed to be handled by the user of the cache. There is no authentication or authorization checking done before objects are returned from the cache. The information transmitted between caches is not encrypted. If sensitive information is being stored in the cache and the environment the cache is running in is not secure, the process hosting the cache should not be shared between applications and the cache should not be configured for distributed use. In future releases encryption between caches and authorization checking at the region level will be available.

3.11. Compatibility

3.11.1. Distribution/Replication

The cache will work distributed across multiple processes and nodes using its own messaging structure. The messaging system is based on a "group" protocol build on top of TCP. Updates and invalidates of distributed cached objects are multicast to all the caches registered in the system. Cache registration with the system is handled through a well known configurable port. The group protocol handles the joining and leaving of caches to the system to maintain a consistent view of cached objects across the different cache instances. For more detail see "Functional Specification for Group Communication, Java Object Caching".

3.11.2. Internationalization

All error messages will be internationalized using Java ResourceBundles. Cache object names should be accepted in any language.

4. Client Interfaces

4.1. User Interface

All access to the cache is through the class **CacheAccess**. This handle is associated with a region and can be used to access any object in that region. For transparent loading of objects into the cache, the **CacheLoader** class can be extended by the user. This allows the cache to load objects as necessary, synchronizing the loading of objects without restricting the user on how and from where the object to be cached is loaded. Cache administration methods are defined in the class **Cache**.

The **CacheAccess** class implements the following user methods:

1. static void **defineRegion (**String **name**) throws ObjectExistsException, NullObjectNameException, CacheNotAvailableException

2. static void **defineRegion (**String **name,** Attributes **attr**) throws ObjectExistsException, NullObjectNameException, CacheNotAvailableException

   defineRegion is a static method that will create a named region within the cache. This defines a name space for storing objects. **Name** must be globally unique. If the cache system has not been initialized, it will be initialized using the Cache attributes defined in the properties file OCS4J.properties. If the properties file is not found default values will be used. **Attr** can be used to set default object attributes for objects in the region. If a region of **name** already exists a ObjectExistsException will be thrown

3. static CacheAccess **getAccess**() throws CacheException

   static CacheAccess **getAccess**(String **region**) throws CacheException

   GetAccess is a static method that will return a CacheAccess object allowing access to the cache region specified. If no **region** is supplied, a CacheAccess object to the default region is returned.

4. Object **get**(Object **name**) throws ObjectNotFoundException, NotARetrievableObjectException, InvalidHandleException

   Object **get**(Object **name**, Object **args**) throws ObjectNotFoundException, NotARetrievableObjectException, InvalidHandleException

   Object **get**(Object **name**, String **group**, Object **args**) throws ObjectNotFoundException, InvalidGroupException, NotARetrievableObjectException, InvalidHandleException

   Get returns a reference to the object associated with **name.** If the object is a streamAccess object, an InputStream is returned, if the object is a disk object a String containing the full path to the object is returned. The **name** object must override the Object.equals and Object.hashCode methods. If the object is not currently in the cache and a loader object has been registered, the object will be loaded into the cache passing the **args** parameter to the load method of the loader object. If a loader object has not been registered for the object the default load method will do a netSearch for the object. If a **group** is specified and the object is loaded, it will be associated with the group. The object returned by **get** is always a reference to a shared object. **Get** will always return the latest version of the object. A CacheAccess object will only maintain a reference to one cached object at any given time. If **get** is called multiple times, the object accessed previously, will be released. If the object is not found in the cache, an ObjectNotFoundException is thrown. If the group is specified but doesn't exist an InvalidGroupException will be thrown. If name refers to a group a NotARetrievableObjectException is thrown.

5. void **put**(Object **name**, Object **obj**) throws CacheException

   void **put**(Object **name**, String **group**, Object **obj**) throws CacheException

   void **put**(Object **name**, Attributes **attr,** Object **obj**) throws CacheException

   void **put**(Object **name**, String **group**, Attributes **attr**, Object **obj**) throws CacheException

   Put allows a new object to be placed into the cache identified by **name**. If there is currently an object associated with **name** in the region, an ObjectExistException is thrown. Names are scoped to a region so they must be unique within the region they are placed. **Put** is intended for very simple caching situations. In general it is better to create a CacheLoader object and allow the cache to manage the creation and loading of objects. Attributes to associate with the object may be specified with **attr**. If **attr** is not supplied, default attributes are assumed. The **name** object must override the Object.equals and Object.hashCode methods.

6. object **replace**(Object **name**, Object **obj**) throws CacheException

   object **replace**(Object **name**, String **group**, Object **obj**) throws CacheException

   Replace will create a new version of the object identified by **name**, replacing the current version with the object **obj**. If the object doesn't exist in the cache **replace** is equivalent to a **put**. The attributes will be inherited from the existing object or if no object exists, from the group or region the object is associated with. Names are in the scope of a region so they must be unique within the region they are placed. **Replace** is not valid on a disk, streamAccess or group object. **Replace** returns a reference to the newly cached object. The **name** object must override the Object.equals and Object.hashCode methods.

7. void **invalidate**() throws CacheException

   void **invalidate**(Object **name**) throws CacheException

   Invalidate will mark all objects within the scope of **name** as invalid. If **name** refers to a group object the invalidate will cascade to all objects associated with the group or any subgroups. The group objects themselves are not invalidated. Destroy must be called to remove groups. If no name is specified, all objects in the region will be invalidated. Invalidate doesn't "unregister" an object. The loader object will remain associated with the object name. To completely remove any knowledge of an object from the cache, destroy must be called. The **name** object must override the Object.equals and Object.hashCode methods.

8. void **close**()

   Close will return the CacheAccess object to the cache. Any attempts to use a CacheAccess object after close has been called will result in an InvalidHandleException.

9. void **defineGroup**(String **name**) throws CacheException

   void **defineGroup**(String **name,** String **group**) throws CacheException

   void **defineGroup**(String **name**, Attributes **attr**) throws CacheException

   void **defineGroup**(String **name**, String **group,** Attributes **attr**) throws CacheException

   DefineGroup is used to create a new group object. Attributes may be set on the group. If no attributes are specified, the attributes of the region or group the new group is associated with are used. If **group** is specified the new group will be associated with the group specified.

10. void **defineObject**(Object **name**, Attributes **attr**) throws CacheException

    void **defineObject**(Object **name**, String **group**, Attributes **attr**) throws CacheException

    DefineObject is used to specify the attributes to associate with an object when it is loaded. This can include the loader object, cache event handlers and specific attributes for the object, such as distribute, spool, etc. Attributes (with the exception of the CacheLoader object itself) can also be specified within the load method of the CacheLoader object using the setAttributes method, If the attributes for an object are not defined, the attributes of the region will be used.

11. void **destroy** () throws CacheException

void **destroy** (Object **name**) throws CacheException

Destroy will invalidate all objects associated with **name** removing all references to the objects from the cache including any loader registered for the object. If **name** is not specified, the region and all objects within the region will be destroyed. If destroy is called without a name attribute, the CacheAccess object can no longer be used as it will be closed and returned to the cache pool. The **name** object must override the Object.equals and Object.hashCode methods.

12. void **waitForResponse**(int **timeout**) throws CacheException

WaitForResponse may be used to wait for replies returned from invalidates or updates when a reply is requested. This method will block the calling thread until all the responses associated with the CacheAccess object have been received or the time indicated by **timeout** has expired. **Timeout** is the maximum number of milliseconds to wait for all remote caches to reply. If the time out is reached before all responses are received a TimeoutException is thrown. If this method times out and the caller does not intend to call **WaitForResponse** again on this event **cancelResponse** should be called. If the object is local or a reply has not been requested, this call returns immediately.

13. void **cancelResponse**() throws CacheException

CancelResponse terminates the request for a reply from the previous invalidate or update. If a response was requested then either **cancelResponse** or **waitForResponse** should be called to terminate the request and free up related structures. If the **waitForResponse** method times out, **cancelResponse** should be called. All response associated with the CacheAccess object are canceled. If the object is local or a reply has not been requested, the call returns immediately.

14. boolean **getOwnership**(Object **name,** int **timeout**) throws CacheException

GetOwnership will claim the ownership of the object, **name,** for this instance of **CacheAccess**. If ownership is not available it will block for the specified time out period, **timeout**. The local cache is checked first, if the local cache doesn't hold ownership of the object, a message is sent to all other caches in the system. Ownership is only relevant for synchronized objects. Ownership is maintained until an update or invalidation completes (this includes the receipt of replies when applicable) or until ownership is explicitly released with a call to the **releaseOwnership** method. An instance of **CacheAccess** can only hold ownership of one object at a time. Ownership only applies to individual objects it is not available on groups. If ownership was obtained the boolean value of true is returned, otherwise false is returned. The **name** object must override the Object.equals and Object.hashCode methods.

15. void **releaseOwnership**() throws CacheException

ReleaseOwnership is called to explicitly give up ownership of an object. Ownership is only relevant for synchronized objects. If ownership is not held releaseOwnership is ignored. If the object is not synchronized however, a CacheException is thrown.

16. void **resetAttributes**(Attributes **attr**) throws CacheException, InvalidHandleException

void **resetAttributes**(Object **name**, Attributes **attr**) throws CacheException, InvalidHandleException

ResetAttributes allows for some of the attributes of a region to be reset in particular expiration time attriubtes, time to live, default time to live and idle time, and event handlers. The cacheloader object and attributes set as flags can't be reset with **resetAttributes**, the object must be destroyed and redefined to cahcne those parameters. Changing default settings on groups and regions will not affect existing objects. Only object loaded after the reset will use the new defaults. If no **name** argument is provided, the reset is applied to the region.

17. Attributes **getAttributes**() throws CacheException

Attributes **getAttributes**(Object **name**) throws CacheException

GetAttributes will return an attribute object describing the current attributes associated with the object **name**. If no **name** parameter is available, the attributes for the region will be returned. The **name** object must override the Object.equals and Object.hashCode methods.

18. void **save**()

void **save**(Object **name**) throws CacheException

The **save** method will cause all the objects within the scope of **name** (the region if no name is provided) to be saved to the disk cache. If **name** refers to a specific object and the object is not serializable a CacheException will be logged. If the **save** references a group or region, objects that can't be serialized will be ignored. All exceptions encountered will be logged (assuming logging is on) Local objects will be saved in the process specific cache, distributed objects will be saved in the machine global disk cache.

19. void **preLoad**(Object **name**)

void **preLoad**(Object **name,** Object **args**)

20. void **preLoad**(Object **name,** String **group,** Object **args**)

PreLoad allows for asynchronous loading of objects into the cache. This method will schedule a background task to the registered load method then return. Any exceptions that occur during the load will be written to the log (if one is available). The **name** object must override the Object.equals and Object.hashCode methods. The object will be loaded into the cache passing the **args** parameter to the load method of the loader object. If a loader object has not been registered an ObjectNotFoundException will be thrown. If a **group** is specified the object will be associated with the group. The group must already exist at the time of the load or an InvalidGroupException will be logged.

21. boolean **isPresent**(Object **name**)

IsPresent returns true if a valid copy of the named object is current in the cache. In all other cases false is returned.

The **Attributes** class defines the following constants and methods:

Static Flag Values

- **DISTRIBUTE:** indicates the object is distributed, updates and invalidations are distributed to other processes Default is to not distribute changes.
- **NOFLUSH** indicates not to flush the object from the cache if the object is distributed and the cache is isolated from the other caches. Default is to flush the object This flag is ignored if a "time to live" is specified or the object is local.
- **REPLY** indicates a reply should be sent from remote caches if this object is updated or invalidated. The default is not to reply. This flag is ignored if the object is local.
- **SYNCHRONIZE** indicates that updates to this object should be synchronized. If this flag is set only the "owner" of an object can update or invalidate the object. The default is not to synchronize updates.
- **SPOOL** indicates the object should be spooled to disk when the object is being removed from the memory cache because of space limitations. This flag is only valid for memory objects**.**
- **GROUP_TTL_DESTROY** indicates that the group object should be destroyed when the associated time to live expires. In the default case only the child objects are invalidated the group remains valid.
- **ORIGINAL** indicates the object was created in the cache and can't be recreated if it was removed from the cache. Original objects don't get removed from the cache when they are not referenced they must be invalidated before they get removed from the cache.

1. void **setFlags**(long **flags**)

SetFlags is used to specify which of the above listed attributes should be set in the Attributes object. The flags may be "or'ed" together, i.e., Attributes.LOCAL|Attributes.SPOOL.

2. void **setLoader**(CacheLoader **loader**)

   SetLoader will associate a loader object with attribute object.

3. void **setVersion**(long **version**)

   SetVersion sets the version attribute. A version number is maintained for the users convenience. It is not use internally by the cache.

4. void **setTimeToLive** (long **ttl**) throws InvalidArgumentException

   SetTimeToLive will set the maximum time the associated cache object will stay in the cache before it is invalidated. The time starts when the object is loaded into the cache (by a CacheLoader object or the put or replace method) or when the time to live attribute is set via the setAttributes method. **Ttl** is in seconds. The **timeToSeconds** method may be used to convert days, hours, and/or minutes to seconds. If a negative value for **ttl** is supplied an InvalidArgumentException will be thrown.

5. void **setDefaultTimeToLive** (long **ttl**) throws InvalidArgumentException

   SetDefaultTimeToLive will set the maximum time the associated cache object will stay in the cache before it is invalidated. **Ttl** is in seconds. For regions and groups, this will establish a default time to live that is applied individually to each member of the group or region. It will not cause the entire group or region to "time out" as a whole. For individual objects the default time to live is equivalent to time to live. If both are set the default time to live is ignored. The **timeToSeconds** method may be used to convert days, hours, and/or minutes to seconds. If a negative value for **ttl** is supplied an InvalidArgumentException will be thrown.

6. void **setIdleTime**(long **idle**) throws InvalidArgumentException

   SetIdleTime will set the maximum time the associated cache object will remain in the cache without being referenced before it is invalidated. **idle** is in seconds. The **timeToSeconds** method may be used to convert days, hours, and/or minutes to seconds. If a negative value for **idle** is supplied an InvalidArgumentException will be thrown.

7. void **setListener**(int **event**, CacheEventListener **listener**)

   SetListener registers an event listener object to be executed when the specified event occurs with relationship to the associated object. Currently the only the invalidate event being monitored is (Attributes.INVALIDATE_EVENT).

8. void **setSize**(int **size**)

   The **setSize** method is used to specify the size in bytes of the object being cached. This is used to determine when the cache capacity is reached. If the cache is not using object size to determine the capacity (it can also use an object count) this value is ignored.

9. int **getSize**()

   The **getSize** method returns the specified size of the object. This size is set by the setSize method or,in the cache of StreamAccess objects, the size is calculated by the cache. If the size has not been set a 0 is returned.

10. boolean **isSet**(long **flags**)

    The **isSet** method returns true if the specified attribute is set, false otherwise. **Flags** may be "or'ed" together in which case **isSet** will return true only if all of the attributes are set.

11. long **getCreateTime**()

    GetCreateTime returns the time the object was loaded into the cache. The time is the number of milliseconds from midnight, January 1, 1970 (UTC).

12. CacheLoader **getLoader**()

    GetLoader returns the cacheLoader object attribute.

13. long **getVersion**()

    GetVersion returns the current value of version.

14. long **getIdleTime**()

    GetIdleTime returns the current value for the idle time interval.

15. long **getTimeToLive** ()

    GetTimeToLive returns the current value for the time to live interval.

16. long **timeToSeconds** (int **days**, int **hours**, int **minutes**, int **seconds**) throws InvalidArgumentException

    TimeToSeconds will convert the time specified in **days**, **hours**, **minutes** and **seconds** to seconds. If a negative value is supplied for any of the arguments, an InvalidArgumentException will be thrown.

The **CacheLoader** class defines the following methods. This class should be extended by the user to implement custom loaders.

1. abstract Object **load**(Object **handle,** Object **args**) throws CacheException

   Load is a user written method to load the object into the cache. This will typically be a call to the database or directory, or extracting information from a file. This method should return a reference to the newly loaded object. If the object being loaded is a StreamAccess object or a disk object, the OutputStream object created by the **createStream** method or the File object created by the **createDiskObject** method should be returned. **Handle** is supplied by the cache and is used by the netSearch and setAttributes method to access information about the object being searched for. **Args** is the object pass to the cache in the **get** method.

2. void **setAttributes**(Object **handle,** Attributes **attr)** throws CacheException

   SetAttributes will cause the attributes associated with the object being loaded to be set to the values provided in **attr**. **Handle** is the object passed into the load method. If the object being loaded is a StreamAccess or Disk object, the attributes should be set with the create call rather than with setAttributes.

3. Object **getName** (Object **handle**)

   GetName returns the name Object associated with the object being loaded. This method is available to be called by application overrides of the load method. This is a protected method so it is available only from the load method.

4. String **getRegion**(Object **handle**)

   GetRegion returns the name of the region for the object being loaded. This method is available to be called application overrides of the load method. This is a protected method so it is available only from the load method.

5. Object **netSearch**(Object **handle,** int **timeout)** throws CacheException

   NetSearch will search other caches for the object to be loaded. This method is called from the default load method and is available to be called by application overrides of the load method. This is a protected method so it is available only from the load method. If the search is successful a reference to a local copy of the object is returned. If the object is not found an ObjectNotFoundException is thrown.

6. OutputStream **createStream**(Object **handle**) throws ObjectExistsException

   OutputStream **createStream**(Object **handle,** Attributes **attr**) throws ObjectExistsException

   The **createStream** method is called from the load method to create a streamAccessed object. The OutputStream object returned is used to load the object into the cache. **Handle** is the object passed into the load method. The attributes for the object should be set on the createStream call. If attr is null, default attributes are assumed . If the object is declared as distributed, it is possible another cache has already

loaded the object or is in the process of loading it. In this case an ObjectExistsException is thrown. The applications can either allow the exception to propagate back to the caller of the load or catch the exception and return null to the caller. In both cased the cache will recognize that the object has been loaded by another cache and return the object to the user.

7. File **createDiskObject**(Object **handle**, String **extension**) throws ObjectExistsException

   File **createDiskObject**(Object **handle**, Attributes **attr,** String **extension**) throws ObjectExistsException

   The **createDiskObject** method is called from the load method to create a disk object. The File returned can then be used to load the object into the disk cache. The **extension** parameter is used as the extension to the file name (java, class, exe, etc.). If the parameter is null, no extension is added to the file name. **Handle** is the object passed into the load method. The attributes for the object should be set on the createStream call. If attr is null, default attributes are assumed . If the object is declared as distributed, it is possible another cache has already loaded the object or is in the process of loading it. In this case an ObjectExistsException is thrown. The applications can either allow the exception to propagate back to the caller of the load or catch the exception and return null to the caller. In both cased the cache will recognize that the object has been loaded by another cache and return the object to the user.

8. void **log**(String **msg**)

   The **log** method is called from the load method to record a message in the cache's log. How and where the logging occurs is dependent of the configuration of the logger in the cache.

9. CacheException **exceptionHandler**(String **msg**, Exception **ex**)

   The **exceptionHandler** method is called from the load method to convert any non CacheExceptions into CacheExceptions with the base exception set to the original exception (see CacheException for details). This allows the load method to only throw CacheExceptions without loosing important information. The exception will also be logged (assuming that logging is configured and the logging severity is set sufficiently high). For all CacheExceptions, if CacheException.printStackTrace() is called and there is a base exception, the stack for the base exception will be printed.

The **CacheEventListener** interface defines the following methods.

1. void **handleEvent**(CacheEvent **event**) throws CacheException

   HandleEvent is a callback method. When a registered event happens, the cache invokes this method and passes in a CacheEvent object.

The **CacheEvent** class extents the java.util.EventObject class.

CacheEvent represents an internal cache event. If an event happens on a cached object, the source object in CacheEvent is the cached object which relates to the event that just happened. If an event happens on a cache group, the source object in CacheEvent is the group name which relates to the event that just happened. Event id is used to identify different types of events. Applications can register a CacheEventListener to handle events. Currently, only the OBJECT_INVALIDATED event is defined.

1. int **getId**()

   getId returns the event identifier associated with the event. Currently the only event supported is CacheEvent.OBECT_INVALIDATED.

4.2. Administrative Interface

The **CacheAttributes** class defines the following methods:

1. void **setLocal**()

   SetLocal sets the attribute to indicate the cache is local. Invalidations and updates will not be propagated to other caches in the system.

2. void **setMaxObjects**(int **size**)

   SetMaxObjects is used to set the attribute to determine the maximum number of objects allowed in the memory cache. If the max number of objects or the cache size is set, the default for the one not set is ignored. If both are set, both are used to determine the capacity of the cache, i.e., object will be removed from the cache if either limit is reached.

3. void **setMemoryCacheSize**(int size)

   SetMemoryCacheSize sets the attribute to indicate the maximum size of the memory cache. **Size** is in megabytes. If the max number of objects or the cache size is set, the default for the one not set is ignored. If both are set, both are used to determine the capacity of the cache, i.e., object will be removed from the cache if either limit is reached.

4. void **setDiskCacheSize**(int **size**)

   SetDiskCacheSize sets the attribute to indicate the maximum size of the disk cache. **Size** is in megabytes.

5. void **setDiskPath**(String **path**)

   SetDiskPath sets the attribute indicating the root location for the disk cache.

6. void **setLogger**(CacheLogger **pLogger**)

   SetLogger sets the logger object which will be used to log cache messages.

7. void **setDefaultLogFileName**(java.lang.String **pDefaultLogFileName**)

   SetDefaultLogFileName sets the log file name for the DefaultCacheLogger. If the default logger is being used (a new CacheLogger has not been implemented and set), all cache log messages will be written to the file “javacache.log” which is created in the directory the server process is started up in. This method changes the location and name of the file to be used. **PDefaultLogFileName** is a full path name for the log file.

8. void **setCleanInterval**(int **seconds**)

   SetCleanInterval sets the attribute indicating the how often the cache should be checked for objects invalidated by “time to live” or “idle time” attributes.

9. void **addCacheAddr**(inetAddress **ipAddr**, int **port**)

   AddCacheAddr is used to specify the network address and port to be used by the cache messaging system. At least one known address is required by the cache to allow discovery when a process using the cache is first brought on line. If no address is specified, localhost with a default port is use. If the system of caches is across multiple nodes, it is best to have an address specified for each node to protect against unavailable nodes.

10. Enumeration **getCacheAddrs**()

    GetCacheAddrs returns an enumeration of Strings representing the address for all the cache address configured. If no address were configured the default value is returned. The address is in the form of ipaddress:port (127.0.0.1:12345)

The **CacheLogger** class implements the following administrative methods:

CacheLogger is an abstract class. Applications can extend this class to implement a customized logging mechanism. The caching service uses this API to log cache related messages. Users can use Cache.setLogSeverity(int) to change the desirable cache logging severity. The severity levels are defined as:

- OFF
- FATAL
- ERROR
- DEFAULT
- WARNING
- TRACE
- INFO
- DEBUG

A default cache logger is implemented. If no cache logger is provided, the default cache logger will be used. By default, DefaultCacheLogger will log all the messages to a file called "javacache.log" in the directory where the server process is started. Users can set a different log file name for the default logger when initializing the cache by calling CacheAttributes.setLogFileName(String).

1. abstract void **log**(String message)

   abstract void **log**(String message, Throwable cause)

   This method is an abstract method. Application writers can implement this method and provide their own mechanism to log a message. The first version of **log**, logs a string, the second is intended for logging exceptions.

2. abstract void **init**(String fileName, int severity)

   Init is called by the caching system when the CacheLogger object is instanciated to complete any initialization requires. The **fileName** and **severity** parameters passed in are the values set in the CacheAttributes object passed into the Cache.init method.

3. abstract void **flush**()

   Log messages are buffered by the cache. **Flush** will force the messages to be written out to the file and the buffer is reset.

4. int **getSeverity**()

   GetSeverity returns the current severity level defined in this class. The severity level determines the amount of information that will be logged. The default setting is DEFAULT.

5. int **setSeverity**(int severity)

   SetSeverity sets the severity level to **severity**. The severity level determines the amount of information that will be logged. The default setting is DEFAULT.

The **Cache** class implements the following administrative methods:

1. void **init (**CacheAttribute **attributes)** throws CacheException

   Init initializes the cache, allocating space for metadata and starting service threads. The cache is a process wide service so it can only be initialized onces per process. Subsequent init calls are ignored. The **attributes** parameter contains configuration information to initialize the cache system.

2. void **open()**

   void **open(**String **configFile)**

   Open will create a CacheAttributes object based on the values in a Java properties then call **init**. **ConfigFile** is the name of the properties file to be used. If the parameter is not supplied the properties file "OracleJavaCacheConfig" will be used. If the **open** method is used **init** doesn't need to be called. **Init** should only be called directly if the configuration information originates from some source other than a Java properties file.

3. void **close()**

   Close will mark the cache as " not ready" and shutdown the cache. Marking the cache as "not ready" will prevent any threads from accessing the cache during shutdown. If the cache is distributed, **close** will unregister with the distributed caching system. **Close** should be called as part of process termination.

4. void **flush()** throws CacheException

   The **flush** method will mark all objects in the cache, both on disk and in memory as invalid, forcing objects to be reloaded. All processes sharing the disk cache are notified when the cache is flushed.

5. void **flushMemory()** throws CacheException

   The **flushMemory** method will mark all objects in the cache as invalid, forcing objects to be reloaded. Flushing the memory cache will also invalidate memory objects spooled to disk. Objects that are only cached on disk will not be affected.

6. **void flushDisk()** throws CacheException

   The **flushDisk** method will mark all objects in the disk cache as invalid, forcing objects to be reloaded. Flushing the disk cache will also invalidate memory objects that have been spooled to disk. All processes sharing the disk cache are notified when the cache is flushed.

7. float **getVersion()**

   GetVersion returns the current version of the cache.

8. boolean **isReady()**

   IsReady returns true if the cache has been initialized and not closed, false otherwise.

9. boolean **isDistributed()**

   IsDistributed returns true if the cache is currently in distributed mode, that is it is distributing updates and invalidates within the site, false if all cache actions are local only.

10. Enumeration **listCacheObjects**()

    Enumeration **listCacheObjects**(String **region**)

    ListCacheObjects will return an Enumeration of CacheObjectInfo objects describing the objects in the specified region or in all regions in the cache. CacheObjectInfo will include information such as the object name, the type, what group it is associated with the, reference count, expiration time if any and object attributes.

11. CacheAttributes **getAttributes**() throws CacheNotAvailableException

    GetAttributes returns the current attributes of the cache including the cache version number, whether the cache is local or distributed maximum number of objects in the cache, the disk cache location and the disk cache size.

12. void **setLogSeverity(**int **severity)**

    setLogSeverity sets the log severity of the cache system. This determines which messages the cache formats and logs into the log file. Severity's are defined in the CacheLogger class.

The **CacheObjectInfo** class encapsulates the following object information:

String region	The region the object resides in .
String name	The object name
String type	The object type (Memory Object , Disk Object, Group, Region)

String group	The group the object is associated with
int refcount	The current reference count to the object.
int accesses	The total number of accesses to this object
String expire	The time the object will expire, if any.

Exceptions

The following exceptions may be generated by the cache:

CacheException An exception was caught or generated within the cache.

DiskCacheException An exception was caught or generated within the disk cache management code.

CallbackException An exception was generated by a user supplied callback method.

InvalidHandleException The handle used to reference the object is not valid (CacheAccess.close was probably called). Call CacheAccess.getAccess to reset the handle.

InvalidObjectException An invalid object was unexpectedly encountered.

ObjectNotFoundException The object requested could not be found in the cache and there was insufficient information available to create or load the object.

ObjectExistsException An ObjectExistsException exception is thrown if put is called with the name of an existing cache object.

CacheNotAvailableException The caching system has not been initialized or is temporarily unavailable

InvalidArgumentException One of the arguments to the method is not valid

InvalidGroupException The group object accessed is no longer valid.

NotImplementedException A request for functionality that is not yet available has been made

ResponseFailedException A request from a remote cache has failed.

TimeoutException A blocking call has timed out before the requested task has completed.

CacheFullException The maximum number of objects in the cache has been reached.

CachePermissionsException This object is synchronized and this handle is not the current owner.

CantSynchronizeGroupException Group objects can't be synchronized.

GroupNameException An attempt was made to add a group to itself.

GroupUseException The requested functionality is not supported on a group.

LoadConflictException An update was requested on an object that is currently being updated or loaded.

NetworkException A problem in the message layer of the cache has occurred

NetOfflineException This cache was unable to connect to the message layer

NullObjectException A null cache object was detected.

NullObjectNameException A null object name was supplied by the application

ObjectNotSynchronizedException A method applicable only to synchronized cache objects was attempted on an object that is not marked as synchronized.

NotARetrievableObjectException An attempt was made to retrieve a group as if it were an instance of a cached object.

RegionNotFoundException A request was made to access a region that does not exist in the cache.

CacheException extend ***Exception*** by adding a class variable **base**. Base is of type ***Exception*** and will reference the original exception (if any) caught by the caching system. All other exceptions listed extend **CacheException**.

public Exception **base;** // This is the original exception caught by the system

4.3. Configuration Files

The following configuration values are available for the system administrator. The cache doesn't maintain any configuration files. It is expected the hosting server will integrate these parameters into the existing configuration files.

1. Distribute

   Distribute is a boolean. If it is set, updates and invalidations for objects that have the distribute attriube set are propagated to other processes and objects written to disk are shared between processes on a node. If this flag is false all objects are treated as local regardless of the attribute set on the object. The default value is false.

2. MaxObjects

   MaxObjects determines the maximum number of valid objects allowed in the cache. This count does not include group objects or objects that have been spooled to disk and are not currently in memory. The default value is 5000 objects.

3. MaxSize

   MaxSize is the maximum amount of memory in megabytes that is available to the cache. The cache can be sized by the number of objects, the size, or both. The default is to use only the number of objects.

4. DiskCacheSize

   DiskCacheSize is the maximum amount of disk space, in megabytes, that is available to the disk cache. The default value is 10megabytes

5. DiskPath

   DiskPath is the location on disk for the root of the disk cache. There is no default for this parameter. If diskPath is not set the disk cache is not available.

6. LogFileName

   LogFileName is the full path to the log file that is passed to the logger object when it is initialized. The default is to write log messages to the file "javacache.log" which is created in the directory the server process is started up in.

7. LogSeverity

   LogSeverity is the logging severity the logger will be initialized to. The default is CacheLogger.DEFAULT.

8. Logger

   The logger parameter is the class name of the object that implements the CacheLogger interface. This object will be instantiate when the cache is initialized. The default value is oracle.ias.cache.DefaultLogger.

9. CleanInterval

   CleanInterval determines how often the cache is checked for objects whose time to live or idle time has

expired. This will affect how quickly event handlers are called after the expiration. The time is expressed in seconds. The default value is 30 seconds

10. DiscoveryAddress

The discoveryAddress is used by the distributed cache's messaging system. This is the address initial contacted by a cache to join the caching system. The value is in the form hostname:port number. If the hostname is omitted (:port number) the local hostname is used. If the caching system spans machines a comma separated list of hosts and ports can be provided, 1 per machine. At any given time only 1 address is being used but this will avoid a dependency on any specific machine needing to be started first. The default is :12345 (localhost:12345).