Oracle9*i* Application Server

Oracle9*i*AS Object Caching Service for Java Developer's Guide

Release 1 (v1.0.2.2)

May 2001

Part No. A88852-01

ORACLE

Oracle9*i* Application Server Oracle9*i*AS Object Caching Service for Java Developer's Guide, Release 1 (v1.0.2.2)

Part No. A88852-01

Primary Author: Thomas Van Raalte

Contributors: Jerry Bortvedt, Qihong Chen, Kyle Jan, Jun X. Wang, Yongwen Xu

Editor: Kay Kaufmann

Contents

3 Getting Started with Object Caching Service for Java

4 Disk Cache and StreamAccess Objects

5 Local and Distributed Caching

6 Advanced Cache Usage

Index

Send Us Your Comments

Oracle9*i* Application Server Oracle9*i*AS Object Caching Service for Java Developer's Guide, Release 1 (v1.0.2.2)

Part No. A88852-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: iasdocs_us@oracle.com
- Postal service:
  Oracle Corporation
  Oracle9*i* Application Server Object Caching Service for Java
  500 Oracle Parkway M/S 6op4
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

The *Oracle9iAS Object Caching Service for Java Developer's Guide* describes how to use the Oracle9*i* Object Caching Service for Java.

This preface contains the following topics:

Audience

The *Oracle9iAS Object Caching Service for Java Developer's Guide* is intended for application programmers, system administrators, and other Oracle users who perform the following tasks:

- Configure software installed on Oracle9*i* Application Server
- Create Java programs that run as servlets in a servlet engine on Oracle9*i* Application Server
- Create Java programs that benefit from the features of the Object Caching Service for Java.

To use this document, you need a working knowledge of Java programming language fundamentals.

Organization

This document contains:

Chapter 1, "Object Caching Service for Java Concepts"

This chapter introduces the basic concepts for the Object Caching Service for Java and provides a description of the caching service architecture.

Chapter 2, "Working with Objects and Attributes"

This chapter describes the Object Caching Service for Java constructs used to work with the cache, including the supported object types and the attributes that apply to the different object types.

Chapter 3, "Getting Started with Object Caching Service for Java"

This chapter provides an introduction to the procedures used to setup and start working with the Object Caching Service for Java.

Chapter 4, "Disk Cache and StreamAccess Objects"

This chapter shows you how to work with the disk cache feature of the Object Caching Service for Java, and includes descriptions of disk objects and StreamAccess objects.

Chapter 5, "Local and Distributed Caching"

This chapter describes the local and distributed modes of operation that the Object Caching Service for Java supports. In local mode, objects are isolated to a single Java VM process and are not shared. In distributed mode, the Object Caching Service for Java can propagate object changes, including invalidations, destroys, and replaces, through the cache messaging system to other caches running on a single system or across a network.

Chapter 6, "Advanced Cache Usage"

This chapter covers advanced Object Caching Service for Java features, including Pool objects and Event Listeners.

Related Documentation

For more information, see these Oracle resources:

The Oracle9*i* Application Server Documentation Library

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://technet.oracle.com/membership/index.htm`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://technet.oracle.com/docs/index.htm`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
...	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading

technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

1

Object Caching Service for Java Concepts

This chapter introduces the basic concepts for the Object Caching Service for Java and its architecture.

This chapter covers the following topics:

Where Does the Object Caching Service for Java Fit?

Electronic business (e-business) creates new performance requirements for Web sites. To carry out e-business successfully, Web sites must protect against poor response time and system outages caused by peak loads. Slow performance translates into lost revenue. High-volume Web sites try to counter this problem by adding more application servers to their existing architecture. As more users access these Web sites, more application servers need to be added. The manageability costs associated with adding application servers often outweigh the benefits.

Static caches and content distribution services can provide some relief. However, these solutions are unable to serve content that is dynamically generated.

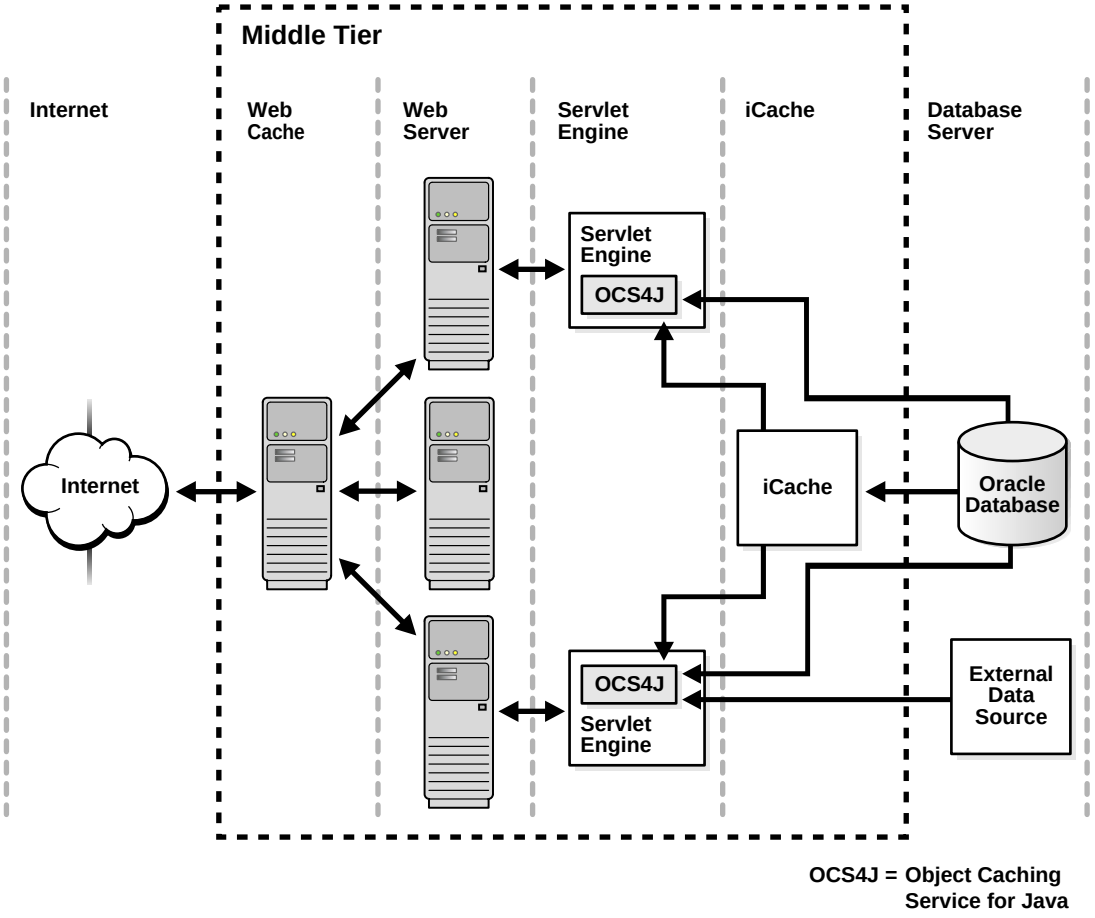
Faced with these performance challenges, e-businesses need to invest in more cost-effective technologies and services to improve the performance of their Web sites. Oracle offers the Object Caching Service for Java to help e-businesses manage Web-site performance issues for dynamically generated content. The Object Caching Service for Java improves the performance, scalability, and availability of Web sites that run on Oracle9*i* Application Server.

By storing frequently accessed or expensive-to-create objects in memory or on disk, the Object Caching Service for Java eliminates the need to repeatedly create and load information within a Java program. The Object Caching Service for Java retrieves content faster and greatly reduces the load on application servers.

Figure 1–1 shows the Oracle9*i* Application Server cache architecture, including the following cache components:

- Oracle Web Cache, The Web cache sits in front of the application servers (Web servers), caching their content and providing that content to Web browsers that request it. When browsers access the Web site, they send HTTP requests to Oracle Web Cache. The Web cache, in turn, acts as a virtual server to the application servers. If the requested content has changed, Oracle Web Cache retrieves the new content from the application servers.
- Object Caching Service for Java, The caching service provides caching for expensive or frequently used Java objects when the application servers use a Java program to supply their content. Cached Java objects may contain generated pages or may provide support objects within the program to assist in creating new content. The Object Caching Service for Java automatically loads and updates objects as specified by the Java application.
- Oracle iCache Data source. You can use the Object Caching Service for Java to generate content from within the application, from an Oracle9*i* Database Server, from the Oracle9*i* iCache, or from some other external data source.

Figure 1–1 Architecture for Object Caching Service for Java and the Oracle9i Application Server



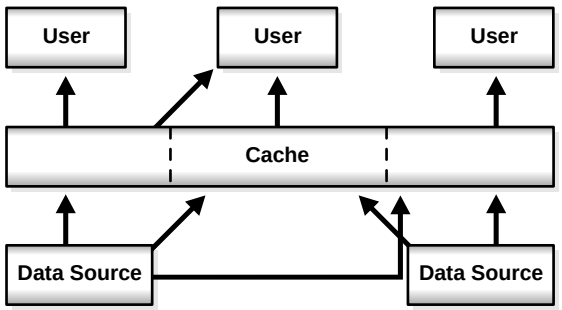
Object Caching Service for Java Basic Architecture

For a programmer using the Object Caching Service for Java, information has one of three characteristics:

1. Static information that never changes. The programmer handles the data efficiently using a Java `Hashtable`.
2. Dynamic information that is unique. The programmer must generate data each time the information is requested.
3. Variable information that is sometimes static and sometimes is generated. The programmer uses the Object Caching Service for Java.

   Figure 1–2 shows the basic architecture for the Object Caching Service for Java. The cache delivers information to a user process. The process could be a servlet application that generates HTML pages or any other Java application.

Figure 1–2 Object Caching Service for Java Basic Architecture

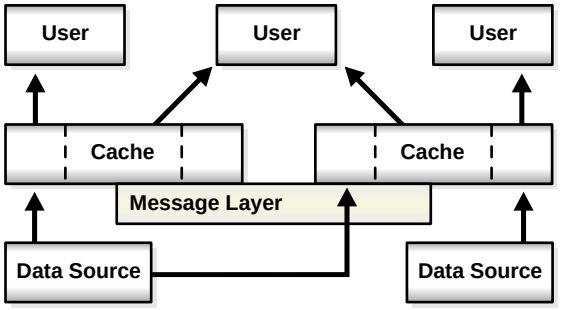


Distributed Object Management

For simplicity, availability, and performance, the Java object cache is specific to each process (object creation is not centrally controlled). However, using distributed object management, the Object Caching Service for Java provides coordination of updates and invalidations between processes. If an object is updated or invalidated in one process, it is also updated or invalidated in all other associated processes. This distributed management allows a system of processes to stay synchronized, without the overhead of centralized control.

Figure 1–3 shows the architecture for the Object Caching Service for Java, using distributed object management. The cache delivers information to a user process. The user process could be a servlet application that generates HTML pages or any other Java application. Using the distributed object management message layer, the application uses the Object Caching Service for Java to share the information across processes and between caches.

Figure 1–3 Object Caching Service for Java Distributed Architecture



How the Object Caching Service for Java Works

The Object Caching Service for Java manages Java objects within a process, across processes, or on a local disk. The Object Caching Service for Java provides a powerful, flexible, and easy-to-use service that significantly improves Java performance by managing local copies of Java objects. There are very few restrictions on the types of Java objects that can be cached or on the original source of the objects. Programmers use the Object Caching Service for Java to manage objects that, without cache access, are expensive to retrieve or to create.

The Object Caching Service for Java is easy to integrate into new and existing applications. Objects can be loaded into the object cache, using a user-defined object, the `CacheLoader`, and can be accessed through a `CacheAccess` object. The `CacheAccess` object supports local and distributed object management. Most of the functionality of the Object Caching Service for Java does not require administration or configuration. Advanced features support configuration using administration application programming interfaces (APIs) in the `Cache` class. Administration includes setting configuration options, such as naming local disk

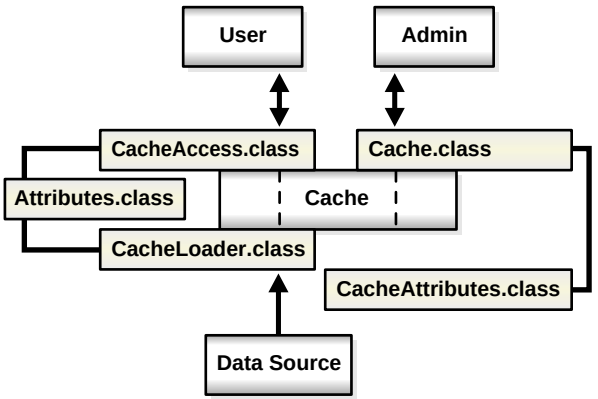
space or defining network ports. The administration features allow applications to fully integrate the Object Caching Service for Java.

Each cached Java object has a set of associated attributes that control how the object is loaded into the cache, where the object is stored, and how the object is invalidated. Cached objects are invalidated based on time or an explicit request (notification can be provided when the object is invalidated). Objects can be invalidated by group or individually.

See Also: Chapter 2, "Working with Objects and Attributes"

Figure 1–4 shows the basic Object Caching Service for Java APIs. Figure 1–4 does not show distributed cache management.

Figure 1–4 Object Caching Service for Java Basic APIs



Cache Organization

The Object Caching Service for Java is organized as follows:

- **Cache Environment**. The cache environment includes cache regions, subregions, groups, and attributes. Cache regions, subregions, and groups associate objects and collections of objects. Attributes are associated with cache regions, subregions, groups, and individual objects. Attributes affect how the Object Caching Service for Java manages objects.
- **Cache Object Types**. The cache object types include memory objects, disk objects, pooled objects, and StreamAccess objects.

Table 1–1 provides a summary of the constructs in the cache environment and the cache object types.

See Also: Chapter 2, "Working with Objects and Attributes"

Table 1–1 Cache Organizational Construct

Cache Construct	Description
Attributes	Functionality associated with cache regions, groups, and individual objects. Attributes affect how the Object Caching Service for Java manages objects.
Cache region	An organizational name space for holding collections of cache objects within Object Caching Service for Java.
Cache subregion	An organizational name space for holding collections of cache objects within a parent region, subregion, or group.
Cache group	An organizational construct used to define an association between objects. The objects within a region can be invalidated as a group. Common attributes can be associated with objects within a group.
Memory object	An object that is stored and accessed from memory.
Disk object	An object that is stored and accessed from disk.
Pooled object	A set of identical objects that the Object Caching Service for Java manages. The objects are checked out of the pool, used, and then returned.
StreamAccess object	An object that is loaded using a Java `OutputStream` and accessed using a Java `InputStream`. The object can be accessed from memory or disk, depending on the size of the object and the cache capacity.

Object Caching Service for Java Features

The Object Caching Service for Java provides the following features:

- Objects can be updated or invalidated.
- Objects can be invalidated either explicitly, or with an attribute specifying the expiration time or the idle time.
- Objects can be coordinated between processes.
- Object loading and creation can be automatic.
- Object loading can be coordinated between processes.
- Objects can be associated in cache regions or groups with similar characteristics.
- Cache event notification provides for event handling and special processing.
- Cache management attributes can be specified for each object or applied to cache regions or groups.

2

Working with Objects and Attributes

This chapter describes the important constructs for the Object Caching Service for Java. The constructs include:

- Object Caching Service for Java Object Types
- Object Caching Service for Java Environment

Object Caching Service for Java Object Types

This section describes the object types that the Object Caching Service for Java manages, including:

- Memory Objects
- Disk Objects
- StreamAccess Objects
- Pool Objects

Memory Objects

Memory objects are Java objects that the Object Caching Service for Java manages. Memory objects are stored in the Java VM's heap space as Java objects. Memory objects can hold HTML pages, the results of a database query, or any information that can be stored as a Java object.

Memory objects are usually loaded into the Object Caching Service for Java with an application-supplied loader. The source of the memory object may be controlled externally (for example, using data in a table on the Oracle9*i* Database Server). The application supplied loader accesses the source and either creates or updates the memory object. Without the Object Caching Service for Java, the application would be responsible for accessing the source directly, rather than using the loader.

You can update memory objects by obtaining a private copy of the memory object, applying the changes to the copy, and then placing the updated object back in the cache (using `CacheAccess.replace()`).

The `CacheAccess.defineObject()` method associates attributes with an object. If attributes are not defined, the object inherits the default attributes from its associated region, subregion, or group.

An application can request that a memory object be spooled to a local disk (using the `SPOOL` attribute). Setting this attribute allows the caching service to handle memory objects that are large, or costly to re-create and seldom updated. When the disk cache is set up to be significantly larger than the memory cache, objects on disk usually stay in the disk cache longer than objects in memory.

Combining memory objects that are spooled to a local disk with the distributed feature from the `DISTRIBUTE` attribute provides object persistence (when the Object Caching Service for Java is running in distributed mode). Object persistence allows you to re-create objects when the system or the Java VM is restarted after the process fails or shuts down.

There are very few restrictions on Object Caching Service for Java memory objects. Memory objects can contain any Java object.

> **See Also:** "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

Disk Objects

Disk objects are stored on a local disk and are accessed directly from the disk by the application using the Object Caching Service for Java. Disk objects may be shared by all Object Caching Service for Java processes, or they may be local to a particular process, depending on the setting for the `DISTRIBUTE` attribute (and whether the Object Caching Service for Java is running in distributed or local mode).

Disk objects can be invalidated explicitly or by setting the `TimeToLive` or `IdleTime` attributes. Disk objects can be updated by obtaining a private copy of the disk object (file). When the Object Caching Service for Java requires additional space, disk objects that are not being referenced may be removed from the cache.

There are very few restrictions on disk objects in the Object Caching Service for Java.

> **See Also:** "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

StreamAccess Objects

StreamAccess objects are objects that are accessed as a stream, and are automatically loaded to the disk cache. The object is loaded as an `OutputStream` and read as an `InputStream`. The Object Caching Service for Java determines how to access the StreamAccess object based on the size of the object and the capacity of the cache. Smaller objects are accessed from memory, while larger objects are streamed directly from disk.

The cache user's access to the StreamAccess object is through an `InputStream`. All the attributes that apply to memory objects and disk objects also apply to StreamAccess objects. A StreamAccess object does not provide a mechanism to manage a stream; for example, StreamAccess objects cannot manage socket endpoints. `InputStream` and `OutputStream` objects are available to access fixed sized, potentially very large objects.

The Object Caching Service for Java places some restrictions on StreamAccess objects.

> **See Also:** "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

Pool Objects

A pool object is a special class of objects that the Object Caching Service for Java manages. A pool object contains a set of identical object instances. The pool object itself is a shared object, while the objects within the pool are private objects. Individual objects within the pool can be checked out to be used and then returned to the pool when they are no longer needed.

Attributes, including `TimeToLive` or `IdleTime` may be associated with a pool object. These attributes apply to the pool object as a whole, or they can be applied to the objects within the pool individually.

The Object Caching Service for Java instantiates objects within a pool using an application-defined factory object. The size of a pool decreases or increases based on demand and on the values of the `TimeToLive` or `IdleTime` attributes. A minimum size for the pool is specified when the pool is created. The minimum-size value is interpreted as a request rather than a guaranteed minimum value. Objects within a pool object are subject to removal from the cache due to lack of space, so the pool may decrease below the requested minimum value. A maximum pool size value can be set that puts a hard limit on the number of objects available in the pool.

> **See Also:** "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

Object Caching Service for Java Environment

The Object Caching Service for Java environment includes the following:

- Cache Regions
- Cache Subregions
- Cache Groups
- Cache Object Attributes

This section describes these Object Caching Service for Java environment constructs.

Cache Regions

Objects that use the caching service are managed within a cache region. A **cache region** defines a name space within the cache. Each object within a cache region must be uniquely named, and the combination of the cache region name and the object name must uniquely identify an object. Thus, cache region names must be unique from other region names, and all objects within a region must be uniquely named relative to the region (multiple objects can have the same name if they are within different regions or subregions).

You can define as many regions as you need to support your application. However, most applications only require one region. The Object Caching Service for Java provides a **default region**; when a region is not specified, objects are placed in the default region.

Attributes may be defined for a region and are then inherited by the objects, subregions, and groups within the region.

See Also: "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

Cache Subregions

Objects that use the caching service are managed within a cache region. Specifying a subregion within a cache region defines a child hierarchy. A **cache subregion** defines a name space within a cache region, or cache subregion. Each object within a cache subregion must be uniquely named, and the combination of the cache region name, the cache subregion name, and the object name must uniquely identify an object.

You can define as many subregions as you need to support your application.

A subregion inherits its attributes from its parent region or subregion unless the attributes are defined when the subregion is defined. A subregion's attributes are inherited by the objects within the subregion. If a subregion's parent region is invalidated or destroyed, the subregion is also invalidated or destroyed.

See Also: "Cache Object Attributes" on page 2-6 and "Restrictions on Identifying Objects" on page 2-11

Cache Groups

A **cache group** creates an association between objects within the Object Caching Service for Java. Cache groups allow related objects to be manipulated together. Objects are typically associated in a cache group because they need to be invalidated together or they use common attributes. Any set of cache objects within the same region or subregion can be associated using a cache group, which may in turn, include other cache groups.

An Object Caching Service for Java object can only belong to one group at any given time. Before an object can be associated with a group, the group must be explicitly created. A group is defined with a name. A group may have its own attributes, or it may inherit its attributes from its parent region, subregion, or group.

Group names are not used to identify individual objects. A group defines a set or collection of objects that have something in common. A group does not define a hierarchical name space. Object type does not distinguish objects for naming purposes; therefore, a region cannot include a group and a memory object with the same name. Use subregions to define a hierarchical name space within a region.

Groups can contain groups, with the groups having a parent and child relationship. The child group inherits attributes from the parent group.

Cache Object Attributes

Cache object Attributes (**Attributes**) affect how the Object Caching Service for Java manages objects. Each object type, region, subregion, and group has a set of associated attributes. An object's applicable attributes contain either the default attribute values; the attribute values inherited from the object's parent region, subregion, or group; or the attribute values that you select for the object.

Attributes fall into two categories:

1. Attributes that must be defined before an object is loaded into the cache. Table 2–1 summarizes these attributes. Each of the attributes shown in Table 2–1 does not have corresponding set or get methods, except the `LOADER` attribute. Use the `Attributes.setFlags()` method to set these attributes.
2. Attributes that can be modified after an object is stored in the cache. Table 2–2 summarizes these attributes.

Note: Some attributes do not apply to certain types of objects. See Object Types sections in the descriptions in Table 2–1 and Table 2–2.

Using Attributes Defined Before Object Loading

The attributes shown in Table 2–1 must be defined on an object before the object is loaded. These attributes determine an object's basic management characteristics.

The following list shows the methods you can use to set the attributes shown in Table 2–1 (by setting the values of an `Attributes` object argument).

- `CacheAccess.defineRegion()`
- `CacheAccess.defineSubRegion()`
- `CacheAccess.defineGroup()`
- `CacheAccess.defineObject()`
- `CacheAccess.put()`
- `CacheAccess.createPool()`
- `CacheLoader.createDiskObject()`
- `CacheLoader.createStream()`
- `CacheLoader.SetAttributes()`

Note: You cannot reset the attributes shown in Table 2–1 by using the `CacheAccess.resetAttributes()` method.

Table 2–1 Object Caching Service for Java Attributes–Set at Object Creation

Attribute Name	Description
DISTRIBUTE	This attribute specifies whether an object is local or distributed. When using the Object Caching Service for Java distributed-caching feature, an object is set as a local object so that updates and invalidations are not propagated to other caches in the site. **Object Types:** When set on a region, subregion, or a group, this attribute sets the default value for the DISTRIBUTE attribute for the objects within the region, subregion, or group, unless the objects explicitly set their own DISTRIBUTE attribute. Pool objects are always local, so this attribute does not apply to pool objects. **Default Value:** All objects are local.
GROUP_TTL_ DESTROY	This attribute indicates that the associated object, group, or region should be destroyed when the TimeToLive expires. **Object Types:** When set on a region or a group, all the objects within the region or group, and the region, subregion, or group itself are destroyed when the TimeToLive expires. **Default Value:** By default only group member objects are invalidated when the TimeToLive expires.
LOADER	This attribute specifies the CacheLoader associated with the object. **Object Types:** When set on a region or a group, the specified CacheLoader becomes the default loader for the region, subregion, or group, the LOADER attribute is individually specified on objects within the region or the group. **Default Value:** By default, no LOADER is set.
ORIGINAL	This attribute indicates that the object was created by the application in the cache, rather than loaded from an external source. ORIGINAL objects are not removed from the cache when the reference count goes to zero. ORIGINAL objects must be explicitly destroyed when they are no longer useful. **Object Types:** When set on a region or a group, this attribute sets the default value for the ORIGINAL attribute for the objects within the region, subregion, or group, unless the objects set their own ORIGINAL attribute. **Default Value**: By default, this attribute is not set.
REPLY	This attribute specifies whether objects can expect to receive a reply from remote caches after a request for an object update or invalidation has completed. This attribute should be set when a high level of consistency is required between cached objects. If the DISTRIBUTE attribute is not set, or the cache is started in nondistributed mode, REPLY is ignored. **Object Types:** When set on a region or a group, this attribute sets the default value for the REPLY attribute for the objects within the region, subregion, or group, unless the objects explicitly set their own REPLY attribute. For memory, StreamAccess, and disk objects, this attribute only applies when the DISTRIBUTE attribute is set to the value DISTRIBUTE. Pool objects are always local, so this attribute does not apply for pool objects. **Default Value:** By default no reply is sent. When DISTRIBUTE is set to local the REPLY attribute is ignored.

Table 2–1 (Cont.) Object Caching Service for Java Attributes–Set at Object Creation

Attribute Name	Description
SPOOL	This attribute specifies that 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. If the object is also distributed, the object can survive the death of the process that spooled it. Local objects are only accessible by the process that spools them, so if the Object Caching Service for Java is not running in distributed mode, the spooled object is lost when the process dies. **Note**: An object must be serializable to be spooled. If this attribute is set on a region, subregion, or group, all associated objects must implement the `java.io.Serializable` interface. **Object Types:** When set on a region, subregion, or a group, this attribute sets the default value for the SPOOL attribute for the objects within the region, subregion, or group, unless the objects set their own SPOOL attribute. **Default Value:** By default, memory objects are not stored to disk.
SYNCHRONIZE	This attribute is used to synchronize updates within multiple threads or at multiple locations within a site. Updates are synchronized by obtaining ownership for objects. Use the `CacheAccess.getOwnership()` method to obtain ownership of an object. Setting the SYNCHRONIZE attribute does not prevent a user from reading or invalidating the object. **Object Types:** When set on a region, subregion, or a group, ownership is applied to the region, subregion, or group as a whole. Pool objects do not use this attribute. **Default Value:** By default updates are not synchronized.
SYNCHRONIZE_ DEFAULT	This attribute indicates that all objects in a region, subregion, or group should be synchronized. Each user object in the region, subregion, or group is marked with the SYNCHRONIZE attribute. Ownership of the object must be obtained before the object can be loaded or updated. Setting the SYNCHRONIZE_DEFAULT attribute does not prevent a user from reading or invalidating objects. Thus, ownership is not required for reads or invalidation of objects that have the SYNCHRONIZE attribute set. **Object Types:** When set on a region, subregion, or a group, ownership is applied to individual objects within the region, subregion, or group. Pool objects do not use this attribute. **Default Value:** By default updates are not synchronized.

Using Attributes Defined Before or After Object Loading

A set of Object Caching Service for Java attributes can be modified either before or after object loading. Table 2–2 lists these attributes. These attributes can be set using the methods listed in the list shown before Table 2–1, and can be reset using the `CacheAccess.resetAttributes()` method.

Table 2–2 Object Caching Service for Java Attributes

Attribute Name	Description
`DefaultTimeToLive`	The `DefaultTimeToLive` applies only to regions, subregions, and groups. This attribute establishes a default value for the `TimeToLive` that is applied to all objects individually within the region, subregion, or group. This value can be overridden be setting the `TimeToLive` on individual objects. **Object Types:** When set on a region, subregion, group, or pool, this attribute applies to all the objects within the region, subregion, group, or pool, unless the objects explicitly set their own `TimeToLive`. **Default Value:** no automatic invalidation.
`IdleTime`	The `IdleTime` attribute specifies the amount of time an object may remain idle, with a reference count of 0, in the cache before being invalidated. If the `TimeToLive` or `DefaultTimeToLive` attribute is set, the `IdleTime` attribute is ignored. **Object Types:** When set on a region, subregion, group, or pool, this attribute applies individually to each object within the region, subregion, group, or pool, unless the objects explicitly set `IdleTime`. **Default Value:** no automatic `IdleTime` invalidation.
`CacheEventListener`	This attribute specifies the `CacheEventListener` associated with the object. **Object Types:** When set on a region, subregion, or a group, the specified `CacheEventListener` becomes the default `CacheEventListener` for the region, subregion, or group, unless a `CacheEventListener` is specified individually on objects within the region, subregion, or the group. **Default Value:** By default, no `CacheEventListener` is set.
`TimeToLive`	The `TimeToLive` attribute establishes the maximum amount of time an object remains in the cache before being invalidated. If associated with a region, subregion, or group, all objects in the region, subregion, or group are invalidated when the time expires. If the region, subregion, or group is not destroyed (that is if, `GROUP_TTL_DESTROY` is not set) the `TimeToLive` value is reset. **Object Types:** When set on a region, subregion, group, or pool, this attribute applies to the region, subregion, group, or pool, as a whole, unless the objects explicitly set their own `TimeToLive`. **Default Value:** no automatic invalidation.
`Version`	An application may set a `Version` for each instance of an object in the cache. The `Version` is available for application convenience and verification. The caching system does not use this attribute. **Object Types:** When set on a region, subregion, group, or pool, this attribute applies to all the objects within the region, subregion, group, or pool, unless the objects explicitly set their own `Version`. **Default Value:** The default `Version` is `0`.

Restrictions on Identifying Objects

Objects are identified by a name that can be any Java object. Usually, the name is represented with a `String`. The Java object used for the identifying name must override the default Java object `equals` method, and the default Java object `hashcode` method. The `String` class provides implementations for both of these methods.

If you provide an object to use as the Object Caching Service for Java name, you need to provide implementations for the `equals` and `hashcode` methods for the object. If the object is distributed, then the `Serializable` interface must also be implemented.

3

Getting Started with Object Caching Service for Java

This chapter provides an introduction to the procedures for setting up and working with the Object Caching Service for Java. Very little setup is required before using the Object Caching Service for Java to place Java objects into the cache and retrieve Java objects from the cache. The topics covered in this chapter include:

- Importing the Caching Service
- Defining a Cache Region
- Defining a Cache Group
- Defining a Cache Subregion
- Defining and Using Cache Objects
- Implementing a CacheLoader
- Invalidating Cache Objects
- Destroying Cache Objects
- Setting Cache Configuration Properties

Importing the Caching Service

The Oracle installer installs the Object Caching Service for Java jar file `cache.jar` in the directory `$ORACLE_HOME/ocs4j/lib` on UNIX or in `%ORACLE_HOME%\ocs4j\lib` on Windows NT.

To use the Object Caching Service for Java, you need to import `oracle.ias.cache`.

```
import oracle.ias.cache.*;
```

Defining a Cache Region

All access to the Object Caching Service for Java is through a `CacheAccess` object. A `CacheAccess` object provides access to the cache through a cache region. You define a cache region, usually associated with the name of an application, using the `CacheAccess.defineRegion()`static method. If the cache has not been initialized, `defineRegion()` initializes the Object Caching Service for Java.

When you define the region, you can also set attributes and create a `CacheLoader` object. Attributes specify how the Object Caching Service for Java manages objects. The `Attributes.setLoader()` method sets the name of `CacheLoader`.

```
Attributes attr = new Attributes();
MyLoader mloader = new MyLoader;
attr.setLoader(mloader);
attr.setDefaultTimeToLive(10);

final static String APP_NAME_ = "Test Application";
CacheAccess.defineRegion(APP_NAME_, attr);
```

The first argument for `defineRegion` uses a `String` to set the region name. This static method creates a private region name within the Object Caching Service for Java. The second argument defines the attributes for the new region.

> **See Also:** "Cache Object Attributes" on page 2-6 and "Implementing a CacheLoader" on page 3-5

Defining a Cache Group

When you want to create an association between two or more objects within the cache, create a cache group. Objects are typically associated in a cache group because they need to be invalidated together or because they have a common set of attributes.

Any set of cache objects within the same region or subregion can be associated using a cache group, including other cache groups. Before an object can be associated with a cache group, the cache group must be defined. A cache group is defined with a name and can use its own attributes, or it can inherit attributes from its parent cache group, subregion, or region. The following code defines a cache group within the region named "`Test Application`".

```
final static String APP_NAME_ = "Test Application";
final static String GROUP_NAME_ = "Test Group";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a group
caccess.defineGroup(GROUP_NAME_);
// Close the CacheAccess object
caccess.close();
```

Defining a Cache Subregion

Define a subregion when you want to create a private name space within a region or within a previously defined subregion. A subregion's name space is independent of the parent name space. A region can contain two objects with the same name, as long as the objects are within different subregions.

A subregion can contain anything that a region can contain, including cache objects, groups, or additional subregions. Before an object can be associated with a subregion, the subregion must be defined. A cache subregion is defined with a name and can use its own attributes, or it can inherit attributes from its parent cache region or subregion. Use the `getParent()` method to obtain a subregion's parent.

In the following example, cache subregion is defined within the region named "`Test Application`".

```
final static String APP_NAME_ = "Test Application";
final static String SUBREGION_NAME_ = "Test SubRegion";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
```

```
// Create a SubRegion
caccess.defineSubRegion(SUBREGION_NAME_);
// Close the CacheAccess object
caccess.close();
```

Defining and Using Cache Objects

You may sometimes want to describe to the Object Caching Service for Java how an individual object should be managed within the cache before the object is loaded. Management options can be specified when the object is loaded, by setting attributes within the `CacheLoader.load()` method. However, you can also associate attributes with an object by using the `CacheAccess.defineObject()` method. If attributes are not defined for an object, the Object Caching Service for Java uses the default attributes set for the region, subregion, or group with which the object is associated.

Example 3–1 shows how to set attributes for a cache object.

Example 3–1 Setting Cache Attributes

```
import oracle.ias.cache.*;
final static String APP_NAME_ = "Test Application";
CacheAccess cacc = null;
try
{
   cacc = CacheAccess.getAccess(APP_NAME_);
// set the default IdleTime for an object using attributes
   Attributes attr = new Attributes();
// set IdleTime to 2 minutes
   attr.setIdleTime(120);

// define an object and set its attributes
   cacc.defineObject("Test Object", attr);

// object is loaded using the loader previously defined on the region
// if not already in the cache.
   result = (String)cacc.get("Test Object");
}  catch (CacheException ex){
     // handle exception
 } finally {
      if (cacc!= null)
         cacc.close();
}
```

Implementing a CacheLoader

Generally, you should use the Object Caching Service for Java to load objects automatically, as needed rather than using the application to directly manage objects in the cache. When an application directly manages objects, it uses the `CacheAccess.put()` method to insert objects into the cache. To take advantage of automatic loading, you use a `CacheLoader` object and implement a `load()` method to insert objects into the cache.

A `CacheLoader` can be associated with a region, subregion, a group, or an object. Using a `CacheLoader` allows the Object Caching Service for Java to schedule and manage object loading, and handle the logic for, "if the object is not in cache then load."

When an object is not in the cache, when an application calls `CacheAccess.get()` or `CacheAccess.preLoad()`, the `CacheLoader` executes the `load` method. When the `load` method returns, the Object Caching Service for Java inserts the returned object into the cache. Using `CacheAccess.get()`, if the cache is full the object is returned from the loader and the object is immediately invalidated in the cache (therefore, using `CacheAccess.get()` with a full cache does not generate a `CacheFullException`).

When a `CacheLoader` is defined for a region, subregion, or group, it is taken to be the default loader for all objects associated with the region, subregion, or group. A `CacheLoader` that is defined for an individual object is used only to load the object.

Note: A `CacheLoader` that is defined for a region, subregion, or group or for more than one cache object needs to be written with concurrent access in mind. The implementation should be thread-safe, since the `CacheLoader` object is shared.

Using CacheLoader Methods Within the Load Method

The Object Caching Service for Java supports several `CacheLoader` methods that you can use within a `load()` method implementation. Table 3–1 summarizes the available `CacheLoader` methods.

Table 3–1 CacheLoader Methods for Use in a Load Method

Method	Description
`setAttributes()`	Sets the attributes for the object being loaded.
`netSearch()`	Searches other available caches for the object to load. Objects are uniquely identified by the region name, subregion name, and the object name.
`getName()`	Returns the name of the object being loaded.
`getRegion()`	Returns the name of the region associated with the object being loaded
`createStream()`	Creates a StreamAccess object
`createDiskObject()`	Creates a disk object
`exceptionHandler()`	Converts noncache exceptions into `CacheExceptions`, with the base set to the original exception
`log()`	Records a messages in the cache service log

Example 3–2 shows a `CacheLoader` using the `cacheLoader.netSearch()` method to check if the object being loaded is available in distributed Object Caching Service for Java caches. If the object is not found using `netSearch()`, the load method uses a more expensive call to retrieve the object (an expensive call might involve an HTTP connection to a remote Web site or a connection to the Oracle9*i* Database Server). For this example, the Object Caching Service for Java stores the result as a `String`.

Example 3–2 Implementing a CacheLoader

```
import oracle.ias.cache.*;
class YourObjectLoader extends CacheLoader{
      public YourObjectLoader () {
      }
      public Object load(Object handle, Object args) {
         String contents;
         // check if this object is loaded in another cache
         try {
            contents = (String)netSearch(5000); // wait for up to 5 seconds
            return new String(contents);
         } catch(ObjectNotFoundException ex){}

         try {
            contents =  expensiveCall(args);
```

```
            return new String(contents);
          } catch (Exception ex) {throw exceptionHandler("Loadfailed", ex);}
            }

    private String expensiveCall(Object args) {
        String str = null;
        // your implementation to retrieve the information.
        // str = ...
        return str;
    }
}
```

Invalidating Cache Objects

An object can be removed from the cache either by setting the `TimeToLive` attribute for the object, group, subregion, or region; or by explicitly invalidating or destroying the object.

Invalidating an object marks the object for removal from the cache. Invalidating a region, subregion, or a group invalidates all the individual objects from the region, subregion, or group, leaving the environment, including all groups, loaders, and attributes available in the cache. Invalidating an object does not undefine the object. The object loader remains associated with the name. To completely remove an object from the cache, destroy the object using the `CacheAccess.destroy()` method.

An object may be invalidated automatically based on the `TimeToLive` or `IdleTime` attributes. When the `TimeToLive` or `IdleTime` expires, objects are by default, invalidated and not destroyed (see `GROUP_TTL_DESTROY` in Table 2–1 for information on changing this default behavior).

If an object, group, subregion, or region is defined as distributed, the invalidate request is propagated to all caches in the distributed environment.

To invalidate an object, group, subregion, or region use `CacheAccess.invalidate()`.

```
CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.invalidate("Test Object");  // invalidate an individual object
cacc.invalidate("Test Group"); // invalidate all objects associated with a group
cacc.invalidate();     // invalidate all objects associated with the region cacc
cacc.close();          // close the CacheAccess access
```

Destroying Cache Objects

An object can be removed from the cache either by setting the `TimeToLive` attribute for the object, group, subregion, or region; or by explicitly invalidating or destroying the object.

Destroying an object marks the object and the associated environment, including any associated loaders, event handlers, and attributes for removal from the cache. Destroying a region, subregion, or a group marks all objects associated with the region, subregion, or group for removal, including the associated environment.

An object may be destroyed automatically based on the `TimeToLive` or `IdleTime` attributes. By default, objects are invalidated and are not destroyed. If the objects need to be destroyed, set the attribute `GROUP_TTL_DESTROY`. Destroying a region also closes the `CacheAccess` object used to access the region.

To destroy an object, group, subregion, or region use the `CacheAccess.destroy()` method.

```
CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.destroy("Test Object"); // destroy an individual object
cacc.destroy("Test Group");  // destroy all objects associated with
                             // the group "Test Group"

cacc.destroy();       // destroy all objects associated with the region
                      // including groups and loaders
```

Setting Cache Configuration Properties

During initialization, the Object Caching Service for Java sets values for configuration properties. Table 3–2 lists the configuration properties for Object Caching Service for Java. By default, the first time a region is created, or the default region is accessed, the Object Caching Service for Java initializes the configuration properties. When the Object Caching Service for Java is installed, the installer updates values for certain administrative properties and places the updated values in the `OCS4J.properties` configuration file, in the directory `$ORACLE_HOME/ocs4j/admin` on UNIX or in `%ORACLE_HOME\ocs4j\admin` on Windows NT.

You can modify the `OCS4J.properties` file to use values other than the default configuration property values. For configuration property values that are not specified in `OCS4J.properties`, the Object Caching Service for Java uses the default values included in Table 3–2.

When the Object Caching Service for Java is initialized, it uses either the default administration property values, or values specified in `OCS4J.properties`. No explicit method calls are required to configure the administrative properties using this initialization technique. The Object Caching Service for Java also supports other initialization techniques (see the `Cache` object methods in the Javadoc for details).

The format for the values in the properties `OCS4J.properties` file is:

property=value

A # character in a configuration file starts a comment. When the # is in the first column, the entire line is a comment. When the # is occurs after a property value specification, it applies to the remainder of the line.

Table 3–2 lists the valid property names and lists the valid types for each property.

Table 3–2 Object Caching Service for Java Configuration Properties

Configuration Property	Description	Type
`cleanInterval`	Specifies the time, in seconds, between each cache cleaning. At the cache-cleaning interval, the Object Caching Service for Java checks for objects that have been invalidated by the `TimeToLive` or `IdleTime` attributes associated with the object. **Default value:** `60`	`int`
`discoveryAddress`	Specifies the address that the Object Caching Service for Java initially contacts to join the caching system, when using distributed caching. The value is in the form, *`hostname`*:*`port`*. If the *`hostname`* is omitted, `localhost` is used. If the caching service spans systems, a comma separated list of hostnames and ports should be included, with one *`hostname`*:*`port`* pair specified for each node. **Default Value:** `:12345` (this is equivalent to `localhost:12345`).	`String`
`diskPath`	Specifies the absolute path to the root for the disk cache (a directory). If this attribute is not set, disk caching is not available. **Default value:** `null`	`String`
`distribute`	Indicates whether the cache is distributed. Updates and invalidation for objects that have the `distribute` property set are propagated to other caches known to the Object Caching Service for Java. If the `distribute` property is set to `false`, all objects are treated as local, even when the attributes set on objects are set to distribute. **Default value:** `false`	`boolean`

Table 3–2 (Cont.) Object Caching Service for Java Configuration Properties

Configuration Property	Description	Type
`logFileName`	Specifies the log file name for the default logger implementation. **Default value:** `$ORACLE_HOME/ocs4j/admin/logs/ocs4j.log` on UNIX or `%ORACLE_HOME%\ocs4j\admin\logs\ocs4j.log` on Windows NT	`String`
`logger`	Specifies the class name for the object that implements the `CacheLogger` interface. The object is instantiated when the Object Caching Service for Java is initialized. **Default value:** `oracle.ias.cache.DefaultCacheLogger`	`String`
`logSeverity`	Specifies the logging severity level used for initializing the logger. The valid values are: ▪ `-1 CacheLogger.OFF` ▪ `0 CacheLogger.FATAL` ▪ `3 CacheLogger.ERROR` ▪ `4 CacheLogger.DEFAULT` ▪ `6 CacheLogger.WARNING` ▪ `7 CacheLogger.TRACE` ▪ `10 CacheLogger.INFO` ▪ `15 CacheLogger.DEBUG` **Default value:** `CacheLogger.DEFAULT`	`int`
`maxObjects`	Specifies the maximum number of in-memory objects that are allowed in the cache. The count does not include group objects, or objects that have been spooled to disk and are not currently in memory. **Default value:** `5000`	`int`
`maxSize`	Specifies the maximum size of the memory, in megabytes, available to the Object Caching Service for Java. **Default value:** `10`	`int`

Note: Configuration properties are distinct from the Object Caching Service for Java attributes that you specify using the `Attributes` class.

Restrictions and Programming Pointers

This section covers restrictions and programming pointers to keep in mind when using the Object Caching Service for Java.

1. The `CacheAccess` object should not be shared between threads. This object represents a user to the caching system. The `CacheAccess` object contains the current state of the user's access to the cache: what object is currently being accessed, what objects are currently owned, and so on. Trying to share the `CacheAccess` object is unnecessary and can result in nondeterministic behavior.
2. A `CacheAccess` object only holds a reference to one cached object at a time. If multiple cached objects are being accessed concurrently, multiple `CacheAccess` objects should be used. For objects stored in memory, the consequences of not doing this are minor since Java prevents the cached object from being garbage collected even if the cache believes it is not being referenced. For disk objects, if the cache reference is not maintained, the underlying file could be removed by another user or by time-based invalidation, causing unexpected exceptions. To optimize resource management, you should keep the cache reference open as long as the cached object is being used.
3. A `CacheAccess` object should always be closed when it is no longer being used. The `CacheAccess` objects are pooled. They acquire other cache resources on behalf of the user. If the access object is not closed when it is not being used, these resources are not returned to the pool and are not cleaned up until they are garbage collected by the Java VM. If `CacheAccess` objects are continually allocated and not closed, available resources and a consequent degradation in performance may occur.
4. When local objects (objects that do not set the `Attributes.DISTRIBUTE` attribute) are saved to disk using the `CacheAccess.save()` method they do not survive the termination of the process. By definition, local objects are only visible to the cache instance where they were loaded. If that cache instance goes away for any reason, the objects it manages, including on disk, are lost. If an object needs to survive process termination, both the object and the cache need to be defined `DISTRIBUTE`.
5. The cache configuration, also called the cache environment, is local to a cache, this includes the region, subregion, group, and object definitions. The cache configuration is not saved to disk or propagated to other caches. The cache configuration should be defined during the initialization of the application.

6. If a `CacheAccess.waitForResponse()` or `CacheAccess.releaseOwnership()` method call times out, it must be called again until it returns successfully. Call these methods with a `-1` timeout value to free up resources, and eliminate waits.
7. When a group is destroyed or invalidated, distributed definitions take precedence over local definitions. That is, if the group is distributed, all objects in the group will be invalidated or destroyed across the entire cache system even if the individual objects or associated groups are defined as local. If the group is defined as local, local objects within the group are invalidated locally, while distributed objects are invalidated throughout the entire cache system.
8. When an object or group is defined with the `SYNCHRONIZE` attribute set, ownership is required to load or replace the object. However, ownership is not required for general access to the object or to invalidate the object.
9. In general, objects stored in the cache should be loaded by the system class loader defined in the `CLASSPATH` when the Java VM is initialized, rather than by a user defined class loader. Specifically, any objects that are shared between applications or may be saved or spooled to disk need to be defined in the system `CLASSPATH`. Failure to do so may result in `ClassNotFoundExceptions` or `ClassCastExceptions`.
10. On some systems, the open file descriptors may be limited by default. On these systems, you may need to change system parameters to improve performance. On UNIX systems, for example, a value of `1024` or greater may be an appropriate value for the number of open file descriptors.
11. When configured in either local or distributed mode, at startup, one active Object Caching Service for Java cache is created in a Java VM process (that is, in the program running in the Java VM that uses the Object Caching Service for Java API).

4

Disk Cache and StreamAccess Objects

This chapter shows you how to work with the disk cache for Object Caching Service for Java, and includes descriptions of disk objects and StreamAccess objects. Disk objects are objects that are stored on a local disk in a file, and the application using the Object Caching Service for Java accesses the file directly. StreamAccess objects are objects that are accessed as a stream (loaded as an `OutputStream` and read as an `InputStream`).

This chapter covers the following topics:

Working with Disk Objects

The Object Caching Service for Java can manage objects on disk as well as in memory.

This section covers the following topics:

- Configuring Properties for Using the Disk Cache
- Local and Distributed Disk Cache Objects
- Adding Objects to the Disk Cache

Configuring Properties for Using the Disk Cache

To configure the Object Caching Service for Java to use a disk cache, set the value of the `diskPath` configuration property in the `OCS4J.properties` file.

Setting the diskPath Configuration Property

To configure the Object Caching Service for Java to use a disk cache, the `diskPath` property in the configuration properties file should be set to the path of the root directory for the disk cache. The default value for `diskPath` is null, which specifies that the Object Caching Service for Java should not enable the disk cache.

Note: when operating in distributed mode. To share disk cache files, all caches cooperating in the same cache system must specify values for the `diskPath` property that represent the same physical disk. However, the values specified for the `diskPath` do not need to be the same.

If you configure the `diskPath` properties to represent different locations on the same or different physical disks, the disk cache objects are not shared.

See Also: "Setting Cache Configuration Properties" on page 3-8

Local and Distributed Disk Cache Objects

This section covers the following topics:

Local Objects

When operating in local mode, all objects are treated as local objects (even when the `DISTRIBUTE` attribute is set for an object). In local mode, all objects in the disk cache are only visible to the Object Caching Service for Java cache that loaded them, and they do not survive after process termination. In local mode, objects stored in the disk cache are lost when the process using the cache dies.

Distributed Objects

When operating in distributed mode, disk cache objects are shared by all caches that have access to the file system hosting the disk cache. This configuration allows for better utilization of disk resources and allows disk objects to persist beyond the life of the Object Caching Service for Java process. Distributed memory objects are not shared by all caches since individual copies of each memory object reside in the individual caches across the system.

Objects stored in the disk cache are identified using the concatenation of the path specified in the `diskPath` configuration property and an internally generated `String` representing the remaining path to the file. Thus, caches that share a disk cache can have a different directory structure, as long as the `diskPath` represents the same directory on the physical disk and is accessible to the Object Caching Service for Java processes.

If a memory object that is saved to disk is also distributed, the memory object can survive the death of the process that spooled it.

> **See Also:** "Automatically Adding Objects" on page 4-4 for information on using the `SPOOL` attribute

Adding Objects to the Disk Cache

There are several ways to use the disk cache with the Object Caching Service for Java, including:

- Automatically Adding Objects
- Explicitly Adding Objects
- Using Objects That Only Reside on Disk Cache

Automatically Adding Objects

The Object Caching Service for Java automatically adds certain objects to the disk cache. Such objects may reside either in the memory cache or in the disk cache. If an object in the disk cache is needed, it is copied back to the memory cache. The action of spooling to disk occurs when the Object Caching Service for Java determines that it requires free space in the memory cache. The Object Caching Service for Java automatically moves objects from the memory cache to the disk cache in two cases.

- When space is running out in the memory cache, the Object Caching Service for Java searches through the cache, looking for memory objects that are not currently accessed. These memory objects may be removed from the cache. If the memory object is defined with the SPOOL attribute set, the memory object is written to disk before it is removed. Spooling saves the memory object to the disk cache, and avoids re-creating the object when or if it is needed again. You should set the SPOOL attribute for objects that are expensive to create, especially if the time required to create the object is greater than the cost of loading the object from disk.
- StreamAccess objects are automatically loaded to disk cache. StreamAccess objects give the Object Caching Service for Java latitude as to how the object is accessed. Smaller StreamAccess objects can be accessed from memory or the disk cache, while larger StreamAccess objects are streamed directly from disk. The Object Caching Service for Java determines how to store the StreamAccess object based on the size of the object and the capacity of the cache.

See Also: "Cache Object Attributes" on page 2-6 and "Working with StreamAccess Objects" on page 4-7

Explicitly Adding Objects

In some situations, you may want to force one or more objects to be written to the Object Caching Service for Java disk cache. Using the `CacheAccess.save()` method, a region, subregion, group, or object is synchronously written to the disk cache (if the object or objects are already in the disk cache, they are not written again).

> **Note:** Using `CacheAccess.save()` saves an object to disk even when the `SPOOL` attribute is not set for the object.

Calling `CacheAccess.save()` on a region, subregion, or group saves all the objects within the region, subregion, or group to the disk cache. During a `CacheAccess.save()` method call, if an object is encountered that cannot be written to disk, either because it is not serializable, or for other reasons, the event is recorded in the Object Caching Service for Java log and the save operation continues with the next object.

Using Objects That Only Reside on Disk Cache

Objects that you only access directly from disk cache are loaded into the disk cache by calling `CacheLoader.createDiskObject()` from the `CacheLoader.load()` method. The `createDiskObject()` method returns a `File` object that the application can use to load the disk object. If the disk object's attributes are not defined for the disk object, set them using the `createDiskObject()` method. The system manages local and distributed disk objects differently; the determination of local or distributed is made when the system creates the object, based on the specified attributes.

> **Note:** If you want to share a disk cache object between distributed caches in the same cache system, you must define the `DISTRIBUTE` attribute when the disk cache object is created. This attribute cannot be changed for the disk cache object after the object is created.

When `CacheAccess.get()` is called on a disk object, the full path name to the file is returned, and the application can open the file, appropriate to its needs.

Disk objects are stored on a local disk and accessed directly from the disk by the application using the Object Caching Service for Java. Disk objects may be shared by all Object Caching Service for Java processes, or they may be local to a particular

process, depending on the setting for the DISTRIBUTE attribute (and the mode the Object Caching Service for Java is running in, either distributed, or local).

Example 4–1 shows a loader object that loads a disk object into the cache.

> **See Also:** "Implementing a CacheLoader" on page 3-5 and "Cache Object Attributes" on page 2-6

Example 4–1 Creating a Disk Object in a CacheLoader

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
   public Object load(Object handle, Object args) {
      File file;
      FileOutputStream = out;
      Attributes attr = new Attributes();

      attr.setFlags(Attributes.DISTRIBUTE);
      try
      {
         file = createDiskObject(handle, attr);
         out = new FileOutputStream(file);

         out.write((byte[])getInfofromsomewhere());
         out.close();
     }
     catch (Exception ex) {
       // translate exception to CacheException, and log exception
         throw exceptionHandler("exception in file handling", ex)
      }
      return file;
      }
   }
```

Example 4–2 shows application code that uses an Object Caching Service for Java disk object. This example assumes the region named "Stock-Market" is already defined with the "YourObjectLoader" loader set up in Example 4–1 as the default loader for the region.

Example 4–2 Application Code That Uses a Disk Object

```
import oracle.ias.cache.*;

try
{
   FileInputStream in;
   File file;
   String filePath;
   CacheAccess cacc = CacheAccess.getAccess("Stock-Market");

   filePath = (String)cacc.get("file object");
   file = new File(filePath);
   in = new FileInputStream(filePath);
   in.read(buf);

// do something interesting with the data
   in.close();
   cacc.close();
}
catch (Exception ex)
{
// handle exception
}
```

Working with StreamAccess Objects

StreamAccess objects are objects that are accessed as a stream and are automatically loaded to the disk cache. The object is loaded as an `OutputStream` and read as an `InputStream`. Smaller StreamAccess objects can be accessed from memory or from the disk cache, while larger StreamAccess objects are streamed directly from disk. The Object Caching Service for Java automatically determines where to access the StreamAccess object based on the size of the object and the capacity of the cache.

The user is always presented with a stream object, an `InputStream` for reading and an `OutputStream` for writing, regardless of whether the object is in a file or in memory. The StreamAccess object allows the Object Caching Service for Java user to always access the object in a uniform manner, without regard to object size or resource availability.

Creating a StreamAccess Object

To create a StreamAccess object, call the `CacheLoader.createStream()` method from the `CacheLoader.load()` method when the object is loaded into the cache. The `createStream()` method returns an `OutputStream` object. The `OutputStream` object can be used to load the object into the cache.

If the attributes have not already been defined for the object, they should be set using the `createStream()` method. The system manages local and distributed disk objects differently; the determination of local or distributed is made when the system creates the object, based on the attributes.

> **Note:** If you want to share a StreamAccess object between distributed caches in the same cache system, you must define the `DISTRIBUTE` attribute when the StreamAccess object is created. This attribute cannot be changed after the object is created.

Example 4–3 shows a loader object that loads a StreamAccess object into the cache.

Example 4–3 Creating a StreamAccess Object in a Cache Loader

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
   public Object load(Object handle, Object args) {
     OutputStream = out;
     Attributes attr = new Attributes();
     attr.setFlags(Attributes.DISTRIBUTE);

     try
     {
        out = createStream(handle, attr);
        out.write((byte[])getInfofromsomewhere());
     }
     catch (Exception ex) {
        // translate exception to CacheException, and log exception
        throw exceptionHandler("exception in write", ex)
     }
     return out;
     }
}
```

5

Local and Distributed Caching

The Object Caching Service for Java supports two modes of operation, local mode and distributed mode. Using local mode, objects are isolated to a single Java VM process and are not shared. Using distributed mode, the Object Caching Service for Java can propagate object changes – including invalidations, destroys, and replaces– through the cache's messaging system to other communicating caches running either on a single system or across a network (the Object Caching Service for Java messaging system is built on top of TCP/IP).

This chapter covers the following topics:

Running in Local Mode

When running in local mode, the Object Caching Service for Java does not share objects or communicate with any other caches running locally on the same machine or remotely across the network. Local mode provides a decentralized architecture that supports a very efficient cache system, with very limited overhead. Object persistence across system shutdowns or program failures is not supported when running in local mode.

By default, the Object Caching Service for Java runs in local mode and all objects in the cache are treated as local objects. When the Object Caching Service for Java is configured in local mode, the cache ignores the `DISTRIBUTE` attribute for all objects.

Running in Distributed Mode

In distributed mode, the Object Caching Service for Java can share objects and communicate with other caches running either locally on the same machine or remotely across the network. Object updates and invalidations are propagated between communicating caches. Distributed mode supports object persistence across system shutdowns and program failures. Running in distributed mode has possible disadvantages. Specifically, significant system resources may be required when a large number of distributed objects need to be invalidated, when very large objects are updated, or when updates must be performed rapidly.

This section covers the following topics:

- Configuring Properties for Distributed Mode
- Using Distributed Objects, Regions, Subregions, and Groups
- Cached Object Consistency Levels

Configuring Properties for Distributed Mode

To configure the Object Caching Service for Java to run in distributed mode, set the value of the `distribute` and `discoveryAddress` configuration properties in the `OCS4J.properties` file.

Setting the Distribute Configuration Property

To start the Object Caching Service for Java in distributed mode, the `distribute` property should be set to `true` in the configuration file.

See Also: "Setting Cache Configuration Properties" on page 3-8

Setting the DiscoveryAddress Configuration Property

In distributed mode, invalidations, destroys, and replaces are propagated through the cache's messaging system. The messaging system requires a known hostname and port address to allow a cache to join the cache system when it is first initialized. Use the `discoveryAddress` property in the `OCS4J.properties` file to specify a list of hostname and port addresses.

By default, Object Caching Service for Java sets the `discoveryAddress` to the value `:12345` (this is equivalent to `localhost:12345`). To eliminate conflicts with other software on the site, you should have your system administrator set the `discoveryAddress`.

If the caching service spans systems, a comma separated list of hostname and port pairs should be included as the value for `discoveryAddress`, with one *`hostname:port`* pair specified for each node. This avoids any dependency on a particular machine being available or on the order the processes are started.

See Also: "Setting Cache Configuration Properties" on page 3-8

Note: All caches cooperating in the same cache system must specify the same set of hostname and port addresses. The address list, set with the `discoveryAddress` property defines the caches that make up a particular cache system. If the address lists vary, the cache system could be partitioned into distinct groups resulting in inconsistencies between caches.

Using Distributed Objects, Regions, Subregions, and Groups

When the Object Caching Service for Java runs in distributed mode, individual regions, subregions, groups, and objects can be either local, or distributed. By default, objects, regions, subregions, and groups are defined as local. To change the default local value, set the `DISTRIBUTE` attribute when the object, region, or group is defined.

A distributed cache may contain both local and distributed objects.

Several attributes and methods in the Object Caching Service for Java allow you to work with distributed objects and control the level of consistency of object data across the caches.

See Also: "Cached Object Consistency Levels" on page 5-8

Using the REPLY Attribute with Distributed Objects

When updating, invalidating, or destroying objects across multiple caches, it is useful to know when the action has completed at all the participating sites. Setting the REPLY attribute causes all participating caches to send a reply to the sender when a requested action has completed for the object with the REPLY attribute set. This also enables the wait for response feature for object updates, invalidates, or destroys, and requires the use of the blocking method `CacheAcces.waitForResponse()`.

To wait for a distributed action to complete across multiple caches, use `CacheAccess.waitForResponse()`. To ignore responses, use the `CacheAccess.cancelResponse()` method, which frees the cache resources used to collect the responses.

Both `CacheAccess.waitForResponse()` and `CacheAccess.cancelResponse()` apply to all objects accessed by the `CacheAccess` object. This allows the application to update a number of objects, then wait for all the replies.

Example 5–1 illustrates how to set an object as distributed and handle replies when the REPLY attribute is set. In this example, the attributes may also be set for the entire region. Attributes could also be set for a group or individual object, as appropriate for your application.

Example 5–1 Distributed Caching Using Reply

```
import oracle.ias.cache.*;

CacheAccess cacc;
String     obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and have a reply generated
// by the remote caches when the change is completed

attr.setFlags(Attributes.DISTRIBUTE|Attributes.REPLY);
attr.setLoader(loader);
```

```
CacheAccess.defineRegion("testRegion",attr);
cacc = CacheAccess.getAccess("testRegion"); // create region with distributed attributes

obj = (String)cacc.get("testObject");
cacc.replace("testObject", obj + "new version"); // change will be propagated to other caches

cacc.invalidate("invalidObject"); // invalidation is propagated to other caches

try
{
// wait for up to a second,1000 milliseconds, for both the update and the invalidate to complete
    cacc.waitForResponse(1000);

catch (TimeoutException ex)
{
   // tired of waiting so cancel the response
   cacc.cancelResponse();
}
cacc.close();
}
```

Using SYNCRONIZE and SYNCHRONIZE_DEFAULT

When updating objects across multiple caches, or when multiple threads access a single object, you may coordinate the update action. Setting the SYNCHRONIZE attribute enables synchronized updates and requires an application to obtain ownership of an object before the object is loaded or updated.

The SYNCHRONIZE attribute also applies to regions, subregions, and groups. When the SYNCHRONIZE attribute is applied to a region, subregion, or group, ownership of the region, subregion, or group must be obtained before an object can be loaded or replaced in the region, subregion, or group.

Setting the SYNCHRONIZE_DEFAULT attribute on a region, subregion, or group applies the SYNCHRONIZE attribute to all of the objects within the region, subregion, or group. Ownership must be obtained for the individual objects within the region, subregion, or group before they can be loaded or replaced.

Note: You can also use the SYNCHRONIZE and SYNCHRONIZE_DEFAULT attributes with objects that are not distributed to control updates for the objects from multiple threads, where each thread uses the Object Caching Service for Java.

To obtain ownership of an object, use `CacheAccess.getOwnership()`. Once ownership is obtained, no other `CacheAccess` instance is allowed to load or replace the object. Reads and invalidation of objects are not affected by synchronization.

Once ownership has been obtained and the modification to the object is completed, call `CacheAccess.releaseOwnership()` to release the object. `CacheAccess.releaseOwnership()` waits up to the specified time for the updates to complete at the remote caches. If the updates complete within the specified time, ownership is released, otherwise a `TimeoutException` is thrown. If the method times out, call `CacheAccess.releaseOwnership()` again. `CacheAccess.releaseOwnership()`must return successfully for ownership to be released. If the time out value is `-1`, ownership is released immediately without waiting for the responses from the other caches.

Example 5–2 Distributed Caching Using SYNCRHONIZE and SYNCHRONIZE_DEFAULT

```
import oracle.ias.cache.*;

CacheAccess cacc;
String     obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and set synchronize attribute
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE);
attr.setLoader(loader);

//create region
CacheAccess.defineRegion("testRegion");
cacc = CacheAccess.getAccess("testRegion");
cacc.defineGroup("syncGroup", attr); //define a distributed synchronized group
cacc.defineObject("syncObject", attr); // define a distributed synchronized object
attr.setFlagsToDefaults()  // reset attribute flags

// define a group where SYNCHRONIZE is the default for all objects in the group
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE_DEFAULT);
cacc.defineGroup("syncGroup2", attr);
try
{
// try to get the ownership for the group don't wait more than 5 seconds
   cacc.getOwnership("syncGroup", 5000);
   obj = (String)cacc.get("testObject", "syncGroup"); // get latest object
   // replace the object with a new version
```

```
   cacc.replace("testObject", "syncGroup", obj + "new version");
   obj = (String)cacc.get("testObject2", "syncGroup"); // get a second object
   // replace the object with a new version
   cacc.replace("testObject2", "syncGroup", obj + "new version");
}

catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for group");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("syncGroup",5000);
}
catch (TimeoutException ex)
{
   // tired of waiting so just release ownership
   cacc.releaseOwnership("syncGroup", -1));
}
try
{
   cacc.getOwnership("syncObject", 5000); // try to get the ownership for the object
   // don't wait more than 5 seconds
   obj = (String)cacc.get("syncObject");  // get latest object
   cacc.replace("syncObject", obj + "new version"); // replace the object with a new version
}
catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for object");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("syncObject", 5000);
}
catch (TimeoutException ex)
{
   cacc.releaseOwnership("syncObject", -1)); // tired of waiting so just release ownership
}
try
{
   cacc.getOwnership("Object2", "syncGroup2", 5000); // try to get the ownership for the object
```

```
   // where the ownership is defined as the default for the group don't wait more than 5 seconds
   obj = (String)cacc.get("Object2", "syncGroup2"); // get latest object
   // replace the object with new version
   cacc.replace("Object2", "syncGroup2", obj + "new version");
}

catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for object");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("Object2", 5000);
}
catch (TimeoutException ex)
{
   cacc.releaseOwnership("Object2", -1)); // tired of waiting so just release ownership
}
   cacc.close();
}
```

Cached Object Consistency Levels

Within the Object Caching Service for Java, each cache manages its own objects locally within its Java VM process. In distributed mode, when using multiple processes or when the system is running on multiple sites, a copy of an object may exist in more than one cache.

The Object Caching Service for Java allows you to specify the consistency level required between copies of objects that are available in multiple caches. The consistency level you specify depends on the application and the objects being cached. The supported levels of consistency vary, from none, to all copies of objects being consistent across all communicating caches.

Setting object attributes specifies the level of consistency. The consistency between objects in different caches is categorized into the following four levels:

- No consistency requirements – Using Local Objects
- Propagating Changes Without Waiting for a Reply
- Propagating Changes and Waiting for a Reply
- Serializing Changes Across Multiple Caches

Using Local Objects

If there are no consistency requirements between objects in distributed caches, an object should be defined as a local object (when `Attributes.DISTRIBUTE` is unset, this specifies a local object). Local is the default setting for objects. For local objects, all updates and invalidation are only visible to the local cache.

Propagating Changes Without Waiting for a Reply

To distribute object updates across distributed caches, an object should be defined as distributed by setting the `DISTRIBUTE` attribute. All modifications to distributed objects are broadcast to other caches in the system. Using this level of consistency does not control or specify when an object is loaded into the cache or updated, and does not provide notification as to when the modification has completed in all caches.

Propagating Changes and Waiting for a Reply

To distribute object updates across distributed caches and wait for the change to complete before continuing, set the object's `DISTRIBUTE` and `REPLY` attributes. Using these attributes, notification occurs when a modification has completed in all caches. When `Attributes.REPLY` is set for an object, replies are sent back to the modifying cache when the modification has been completed at the remote site. These replies are returned asynchronously; that is, the `CacheAccess.replace()` and `CacheAccess.invalidate()` methods do not block. Use the `CacheAccess.waitForResponse()` method to wait for replies and block.

Serializing Changes Across Multiple Caches

To use Object Caching Service for Java's highest level of consistency set the appropriate attributes on the region, subregion, group, or object to make objects act as synchronized objects.

On a region, subregion, or group, setting `Attributes.SYNCHRONIZE_DEFAULT` sets the `SYNCHRONIZE` attribute for all of the objects within the region, subregion, or group.

On an object, setting `Attributes.SYNCHRONIZE` forces applications to obtain ownership of the object before the object can be loaded or modified. Setting this attribute effectively serializes write access to objects. To obtain ownership of an object, use the `CacheAccess.getOwnership()` method. Using the `Attributes.SYNCHRONIZE` attribute, notification is sent to the owner when the update is completed. Use `CacheAccess.releaseOwnership()` to block until

any outstanding updates have completed, and the replies are received. This releases ownership of the object so that other caches can update or load the object.

> **Note:** Setting `Attributes.SYNCHRONIZE` for an object does not effectively synchronize. With `Attributes.SYNCHRONIZE` set, the Object Caching Service for Java forces the cache to synchronize its updates of the object, but does not prevent the Java programmer from obtaining a reference to the object and then modifying the object.

When using this level of consistency, with `Attributes.SYNCHRONIZE`, the `CacheLoader.load()` method should call `CacheLoader.netSearch()` before loading the object from an external source. Calling `CacheLoader.netSearch()` in the load method tells the Object Caching Service for Java to search all other caches for a copy of the object. This prevents different versions of the object from being loaded into the cache from an external source.

6

Advanced Cache Usage

This chapter covers advanced Object Caching Service for Java features, including pool objects and event listeners.

This chapter includes the following sections:

- Working with Pool Objects
- Implementing a Cache Event Listener

Working with Pool Objects

A pool object is a special cache object that the Object Caching Service for Java manages. A pool object contains a set of identical object instances. The pool object itself is a shared object, stored as a static across the entire cache instance, while the objects within the pool object are private objects that the Object Caching Service for Java manages. Users access individual objects within the pool with a check out, using a pool access object, and then return the objects to the pool when they are no longer needed.

This section covers the following topics:

Creating Pool Objects

To create a pool object, use `CacheAccess.createPool()`. The `CreatePool()` method takes as arguments a `PoolInstanceFactory`, and an `Attributes` object, plus two integer arguments. The integer arguments specify the maximum pool size and the minimum pool size. By supplying a group name as an argument to `CreatePool()`, a pool object is associated with a group.

Attributes, including `TimeToLive` or `IdleTime` may be associated with a pool object. These attributes can be applied to the pool object itself, when specified in the attributes set with `CacheAccess.createPool()`, or they can be applied to the objects within the pool individually.

Using `CacheAccess.createPool()`, specify minimum and maximum sizes with the integer arguments. The minimum is specified first. It sets the minimum number of objects to create within the pool. The minimum size is interpreted as a request rather than a guaranteed minimum. Objects within a pool object are subject to removal from the cache due to lack of resources, so the pool may decrease the number of objects below the requested minimum value. The maximum pool size puts a hard limit on the number of objects available in the pool.

Note: Pool objects, and the objects within a pool object are always treated as local objects.

See Also:

- "Implementing a Pool Object Instance Factory" on page 6-4
- "Cache Object Attributes" on page 2-6

Example 6–1 shows how to create a pool object.

Example 6–1 Creating a Pool Object

```
import oracle.ias.cache.*;

   try
   {
      CacheAccess cacc = CacheAccess.getAccess("Stock-Market");
      Attributes  attr = new Attributes();
      QuoteFactory poolFac = new QuoteFactory();

      // set IdleTime for an object in the pool to three minutes
      attr.setIdleTime(180);
      // create a pool in the "Stock-Market" region with a minimum of
      // 5 and a maximum of 10 object instances in the pool
      cacc.createPool("get Quote", poolFac, attr, 5, 10);
      cacc.close();
   }
   catch(CacheException ex)
   {
          // handle exception
   }
}
```

Using Objects from a Pool

To access objects in a pool, use a `PoolAccess` object. The `PoolAccess.getPool()` static method returns a handle to a specified pool. The `PoolAccess.get()` method returns an instance of an object from within the pool (this checks out an object from the pool). When an object is no longer needed, return it to the pool, using the `PoolAccess.returnToPool()` method, which checks the object back into the pool. Finally, call the `PoolAccess.close()` method when the pool handle is no longer needed.

Example 6–2 shows the calls required to create a `PoolAccess` object, check an object out of the pool, and then check the object back in and close the `PoolAccess` object.

Example 6–2 Using a PoolAccess Object

```
PoolAccess pacc = PoolAccess.getPool("Stock-Market", "get Quote");
//get an object from the pool
GetQuote  gq = (GetQuote)pacc.get();
// do something useful with the gq object
// return the object to the pool
pacc.returnToPool(gq);
pacc.close();
```

Implementing a Pool Object Instance Factory

The Object Caching Service for Java instantiates and removes objects within a pool, using an application-defined factory object, a `PoolInstanceFactory`. The `PoolInstanceFactory` is an abstract class with two methods that you must implement, `createInstance()` and `destroyInstance()`.

The Object Caching Service for Java calls `createInstance()` to create instances of objects being accumulated within the pool. The Object Caching Service for Java calls `destroyInstance()` when an instance of an object is being removed from the pool (object instances from within the pool are passed into `destroyInstance()`).

The size of a pool object, that is the number of objects within the pool, is managed using these `PoolInstanceFactory()` methods. The system decreases or increases the size and number of objects in the pool, based on demand, and based on the values of the `TimeToLive` or `IdleTime` attributes. Example 6–3 shows the calls required when implementing a `PoolInstanceFactory`.

Example 6–3 Implementing Pool Instance Factory Methods

```
import oracle.ias.cache.*;
   public class MyPoolFactory implements PoolInstanceFactory
   {
       public Object createInstance()
      {
         MyObject obj = new MyObject();
         obj.init();
         return obj;
       }
       public void destroyInstance(Object obj)
       {
           ((MyObject)obj).cleanup();
       }
   }
```

Implementing a Cache Event Listener

There are a number of events that can occur in the life cycle of a cached object, including object creation and object invalidation. This sections shows how an application can be notified when cache events occur.

To receive notification of an object's creation, implement event notification as part of the `cacheLoader`. For notification of invalidation or updates, implement a `CacheEventListener` and associate the `CacheEventListener` with an object, group, region, or subregion using `Attributes.setCacheEventListener()`.

`CacheEventListener` is an interface that extends `java.util.EventListener`. The cache event listener provides a mechanism to establish a callback method that is registered, and then executes when the event occurs. In the Object Caching Service for Java, the event listener executes when a cached object is invalidated or updated.

An event listener is associated with a cached object, group, region, or subregion. If an event listener is associated with a group, region, or subregion, the listener only runs when the group, region, or subregion itself is invalidated. Invalidating a member does not trigger the event. `Attributes.setCacheEventListener()` takes a boolean argument, that if `true`, applies the event listener to each member of the region, subregion, or group, rather than to the region, subregion, or group itself. In this case, the invalidation of an object within the region, subregion, or group triggers the event.

The `CacheEventListener` interface has one method, `handleEvent()`. This method takes a single argument, a `CacheEvent` object that extends `java.util.EventObject`. This object has two methods `getID()`,which returns the type of event (`OBJECT_INVALIDATION` or `OBJECT_UPDATED`), and `getSource()`, which returns the object being invalidated. For group objects, the `getSource()` method returns the name of the group.

The `handleEvent()` method is executed in the context of a background thread that the Object Caching Service for Java manages. Avoid using JNI code in this method, as the expected thread context may not be available.

Example 6–4 shows how a `CacheEventListener` is implemented and associated with an object or a group.

Example 6–4 Implementing a CacheEventListener

```
import oracle.ias.cache.*;
   // A CacheEventListener for a cache object
   class MyEventListener implements
   CacheEventListener  {

       public void handleEvent(CacheEvent ev)
       {
          MyObject obj = (MyObject)ev.getSource();
          obj.cleanup();
        }

       // A CacheEventListener for a group object
       class MyGroupEventListener implements CacheEventListener {
       public void handleEvent(CacheEvent ev)
       {
          String groupName = (String)ev.getSource();
          app.notify("group " + groupName + " has been invalidated");

       }
   }
```

Use the `Attributes.listener` attribute to specify the `CacheEventListener` for a region, subregion, group, or object.

Example 6–5 shows how to set a cache event listener on an object. Example 6–6 shows how to set a cache event listener on a group.

> **See Also:** "Defining and Using Cache Objects" on page 3-4 for information on working with attributes

Example 6–5 Setting a Cache Event Listener on an Object

```
import oracle.ias.cache.*;

   class YourObjectLoader extends CacheLoader
   {
      public YourObjectLoader () {
      }

      public Object load(Object handle, Object args) {
         Object obj = null;
         Attributes attr = new Attributes();
         MyEventListener el = new MyEventListener();
         attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, el);

         // your implementation to retrieve or create your object

         setAttributes(handle, attr);
         return obj;
    }
}
```

Example 6–6 Setting a Cache Event Listener on a Group

```
import oracle.ias.cache.*;
try
{
   CacheAccess cacc = CacheAccess.getAccess(myRegion);
   Attributes attr = new Attributes ();

   MyGroupEventListener listener = new MyGroupEventListener();
   attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, listener);

   cacc.defineGroup("myGroup", attr);
   //....
   cacc.close();

}catch(CacheException ex)
{
   // handle exception
}
```

Index

E

G

H

I

L

M

N

O

P

R

S

T

V