The Java Utility Library

Ice for Java includes a number of utility APIs in the `IceUtil` package and the `Ice.Util` class. This section summarizes the contents of these APIs for your reference.

On this page:

The `IceUtil` Package in Java

`Cache` and `Store` Classes

The `Cache` class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors for servants that store their state in a database, the `Cache` class can simplify your evictor implementation considerably.

You may also want to examine the implementation of the Freeze background save evictor in the source distribution; it uses `IceUtil.Cache` for its implementation.

The `Cache` class has the following interface:

Java

```
package IceUtil;

public class Cache {
    public Cache(Store store);

    public Object pin(Object key);
    public Object pin(Object key, Object o);
    public Object unpin(Object key);

    public Object putIfAbsent(Object key, Object newObj);
    public Object getIfPinned(Object key);

    public void clear();
    public int size();
}
```

Internally, a `Cache` maintains a map of name-value pairs. The implementation of `Cache` takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations, such as the Freeze background save evictor. The `Cache` class does not limit the number of entries in the cache — it is the job of the evictor implementation to limit the map size by calling `unpin` on elements of the map that it wants to evict.

The `Cache` class works in conjunction with a `Store` interface for which you must provide an implementation. The `Store` interface is trivial:

Java

```
package IceUtil;

public interface Store {
    Object load(Object key);
}
```

You must implement the `load` method in a class that you derive from `Store`. The `Cache` implementation calls `load` when it needs to retrieve the value for the passed key from the backing store. If `load` cannot locate a record for the given key because no such record exists, it must return null. If `load` fails for some other reason, it can throw an exception derived from `java.lang.RuntimeException`, which is propagated back to the application code.

The public member functions of `Cache` behave as follows:

`Cache(Store s)`

The constructor initializes the cache with your implementation of the `Store` interface.

`Object pin(Object key, Object val)`

To add a key-value pair to the cache, your evictor can call `pin`. The return value is null if the key and value were added; otherwise, if the map already contains an entry with the given key, the entry is unchanged and `pin` returns the original value for that key.

This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

`Object pin(Object key)`

This version of `pin` returns the value stored in the cache for the given key if the cache already contains an entry for that key. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding value (if any) from the backing store. `pin` returns the value returned by `load`, that is, the value if `load` could retrieve it, null if `load` could not retrieve it, or any exception thrown by `load`.

`Object unpin(Object key)`

`unpin` removes the entry for the given key from the cache. If the cache contained an entry for the key, the return value is the value for that key; otherwise, the return value is null.

`Object putIfAbsent(Object key, Object val)`

This function adds a key-value pair to the cache. If the cache already contains an entry for the given key, `putIfAbsent` returns the original value for that key. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry (if any) from the backing store and returns the value returned by `load`.

If the cache does not contain an entry for the given key and `load` does not retrieve a value for the key, the method adds the new entry and returns null.

`Object getIfPinned(Object key)`

This function returns the value stored for the given key. If an entry for the given key is in the map, the function returns the corresponding value; otherwise, the function returns null. `getIfPinned` does not call `load`.

`void clear()`

This function removes all entries in the map.

`int size()`

This function returns the number of entries in the map.

The `Ice.Util` Class in Java

Communicator Initialization Methods

`Ice.Util` provides a number of overloaded `initialize` methods that create a communicator.

Identity Conversion

`Ice.Util` contains two methods for converting object identities of type `Ice.Identity` to and from strings.

Per-Process Logger Methods

`Ice.Util` provides methods for getting and setting the per-process logger.

Property Creation Methods

`Ice.Util` provides a number of overloaded `createProperties` methods that create property sets.

Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account).

Stream Creation

Two methods, `createInputStream` and `createOutputStream` create streams for use with dynamic invocation.

Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

Java

```
public static String stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form `<`*`major`*`>.<`*`minor`*`>.<`*`patch`*`>`, for example, `3.4.2`. For beta releases, the version is `<`*`major`*`>.<`*`minor`*`>`b, for example, `3.4`b.

The `intVersion` method returns the Ice version in the form *`AABBCC`*, where *`AA`* is the major version number, *`BB`* is the minor version number, and *`CC`* is patch level, for example, `30402` for version 3.4.2. For beta releases, the patch level is set to `51` so, for example, for version 3.4b, the value is `30451`
.

See Also

- Background Save Evictor
- Java Mapping for Interfaces
- Command-Line Parsing and Initialization
- Setting Properties
- Object Identity
- Java Streaming Interfaces