**What is Cache Software?**

In computing a **cache**  is a component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere. A *cache hit* occurs when the requested data can be found in a cache, while a *cache miss* occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than re - computing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs.

To be cost-effective and to enable efficient use of data, caches are relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications exhibit the locality of reference. Moreover, access patterns exhibit temporal locality if data is requested again that has been recently requested already, while spatial locality refers to requests for data physically stored close to data that has been already requested.

**Interface/API Of A Cache:**

**Interface Cache**

public interface **Cache**

Interface used to interact with the second-level cache. If a cache is not in use, the methods of this interface have no effect, except for contains, which returns false.

**Since:**

Java Persistence 2.0

|  |  |
| --- | --- |
| **Method Summary** | |
| boolean | [**contains**](http://docs.oracle.com/javaee/6/api/javax/persistence/Cache.html#contains(java.lang.Class, java.lang.Object))(java.lang.Class cls, java.lang.Object primaryKey)            Whether the cache contains data for the given entity. |
| void | [**evict**](http://docs.oracle.com/javaee/6/api/javax/persistence/Cache.html#evict(java.lang.Class))(java.lang.Class cls)            Remove the data for entities of the specified class (and its subclasses) from the cache. |
| void | [**evict**](http://docs.oracle.com/javaee/6/api/javax/persistence/Cache.html#evict(java.lang.Class, java.lang.Object))(java.lang.Class cls, java.lang.Object primaryKey)            Remove the data for the given entity from the cache. |
| void | [**evictAll**](http://docs.oracle.com/javaee/6/api/javax/persistence/Cache.html#evictAll())()            Clear the cache. |

**Method Detail**

**contains**

boolean **contains**(java.lang.Class cls,

java.lang.Object primaryKey)

Whether the cache contains data for the given entity.

**Parameters:**

cls - entity class

primaryKey - primary key

**Returns:**

boolean indicating whether the entity is in the cache

**evict**

void **evict**(java.lang.Class cls,

java.lang.Object primaryKey)

Remove the data for the given entity from the cache.

**Parameters:**

cls - entity class

primaryKey - primary key

**evict**

void **evict**(java.lang.Class cls)

Remove the data for entities of the specified class (and its subclasses) from the cache.

**Parameters:**

cls - entity class

**evictAll**

void **evictAll**()

Clear the cache.

**What Are The Cache Eviction Algorithms?**

Bélády's Algorithm

Least Recently Used (LRU)

Most Recently Used (MRU)

Pseudo- LRU (PLRU)

Random Replacement (RR)

Segmented LRU (SLRU)

2-way set associative

Direct-mapped cache

Least-Frequently Used (LFU)

Low Inter Reference Recency Set (LIRS)

Adaptive Replacement Cache (ARC)

Clock With Adaptive Replacement (CAR)

**FIFO Cache:**

In **‘First In First Out’** approach, OS selects the page which is oldest in cache and swaps that it with new page.

**LRU Cache:**

In **‘Least Recently Used’** approach, OS selects the page which was not accessed for longest period of time.

**LFU Cache:**

In **‘Least Frequently Used’** approach, OS selects the page which is accessed least number of time till a given point of time.

**Implementation Details Of Cache Algorithms:**

**LRU**

Use a doubly-linked list in combination with a HashMap.

The HashMap should provide value lookup by key for your cache, in O(1) time.

The HashMap should also contain a pointer to the corresponding node in the doubly-linked list so that when a key is accessed, the corresponding node in the linked list can be deleted from its current position and moved to the tail, which is the position for the most recently seen node.

If you see a new element you haven't seen before (that is not in your HashMap), you add it to both the HashMap and the tail of the linked list.

The linked list allows you to see which element expires next in O(1) time, and if the list exceeds the number of elements you want to cache, you delete the next-expiring element from both the linked list and the HashMap.   
All operations are in O(1) time (expected time, in the case of the HashMap).

**Data Structures Used For Each Cache Type:**

**LRU:**

Doubly Linked List