1. [Caching with Spring:](#_Caching_with_Spring)
2. [Ehcache](#Ehcache)

[EHCache calling in java :](#Ehcache)

<http://stackoverflow.com/questions/16211420/ehcache-with-tomcat-simple-example>

CacheManager.getInstance().addCache("xyz"); // creates a cache called xyz.

Cache xyz = CacheManager.getInstance().getCache("xyz");

xyz.put(new Element("key", new Person()));

Element e = xyz.get("key");

Person p = (Person) e.getObjectValue();

Java/Ehcache: A simple example of use of Ehcache 2.6.2

<http://www.javablog.fr/javaehcache-a-simple-example-of-use-of-ehcache-2-6-2.html>

**Download**  
First, we need to download the **ehcache-core-2.6.2.jar**, **slf4j-api-1.6.1.jar** and **slf4j-jdk14-1.6.1.jar** in **ehcache-core-2.6.2-distribution.tar.gz** from [**http://ehcache.org/downloads**](http://ehcache.org/downloads).

Then, add this jar in the librairies folder of your project **\war\WEB-INF\** or in your pom.xml (MAVEN);

**Configuration ehcache.xml**  
Create (this file could be copied from downloaded package) an **ehcache.xml** file in the classpath of your project. This file contains a default Cache configuration with an implicit name “default” which is a reserved cache name. This cache will be applied to caches created programmatically (using CacheManager.add(String cacheName)).

So, we will add a sample cache named “myCache1″ wich will contain a maximum in memory of 10000 elements, and will expire an element if it is idle for more than 5 minutes (300 sec) and lives for more than 10 minutes (600 sec). If there are more than 10000 elements it will overflow to the disk cache, which in this configuration will go to wherever java.io.tmp is defined on your system. On a standard Linux system this will be ‘/tmp’, for Windows7 it could be ‘C:\Users\username\AppData\Local\Temp’.



**Use / Utilization**  
To prove the effectiveness of Ehcache, we need create several classes.

* First, we will create an utilitary class named **CacheUtil** in order to manipulate and solicit the Ehcache:  
  \* Get the cache instance of Ehcache via the method **getCache**. This method could be synchronized. More, to specify the ehcache configuration file, we could use an environment or a VM variable:

|  |  |  |
| --- | --- | --- |
| 01 | public static CacheManager cacheMgr = null; | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | private static Ehcache getCache(String cacheName){ | |
| 04 | if(cacheMgr == null){ |

|  |  |
| --- | --- |
| 05 | // We could use an environment or a VM variable |
| 06 | cacheMgr = CacheManager.create("...\\config\\ehcache.xml"); | |

|  |  |
| --- | --- |
| 07 | } |
| 08 |  | |

|  |  |  |
| --- | --- | --- |
| 09 | Ehcache cache = null; | |
| 10 | if(cacheMgr!=null){ |

|  |  |  |
| --- | --- | --- |
| 11 | //cache = cacheMgr.addCacheIfAbsent(name); | |
| 12 | cache = cacheMgr.getEhcache(cacheName); |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| 14 |  |

|  |  |  |
| --- | --- | --- |
| 15 | return cache; | |
| 16 | } |

* \* Get data from the cache via the method **getListFromCache**. This method could be synchronized. In anticipation of its use, this method is waiting a **threadName** argument.

|  |  |
| --- | --- |
| 01 | @SuppressWarnings("unchecked") |
| 02 | public static <T> List<T> getListFromCache(String threadName, String cacheName, String key, CacheCreation<T> cacheCreation){ | |

|  |  |  |
| --- | --- | --- |
| 03 | List<T> all = new ArrayList<T>(); | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | Ehcache cache = getCache(cacheName); | |
| 06 | Element element = null; |

|  |  |
| --- | --- |
| 07 | if(cache!=null){ |
| 08 | element = cache.get(key); | |

|  |  |
| --- | --- |
| 09 | } |
| 10 |  | |

|  |  |
| --- | --- |
| 11 | if(element==null){ |
| 12 | System.out.println(threadName+" : CacheUtil.getListFromCache() : the element '"+key+"' has not been found in the cache ---> get the original data."); | |

|  |  |
| --- | --- |
| 13 |  |
| 14 | all = cacheCreation.getAll(); | |

|  |  |
| --- | --- |
| 15 | cache.put(new Element(key, all)); |
| 16 | System.out.println(threadName+" : CacheUtil.getListFromCache() : the original data for the element '"+key+"' has been added in the cache."); | |

|  |  |
| --- | --- |
| 17 |  |
| 18 |  |

|  |  |
| --- | --- |
| 19 | }else{ |
| 20 | System.out.println(threadName+" : CacheUtil.getListFromCache() : the element '"+key+"' has been found in the cache."); | |

|  |  |
| --- | --- |
| 21 |  |
| 22 | //all = (List<T>) element.getValue(); | |

|  |  |  |
| --- | --- | --- |
| 23 | all = (List<T>) element.getObjectValue(); | |
| 24 | } |

|  |  |  |
| --- | --- | --- |
| 25 | return all; | |
| 26 |  |

|  |  |
| --- | --- |
| 27 | } |

* We have also created an abstract class **CacheCreation** to in anticipation of the use of cache:

|  |  |  |
| --- | --- | --- |
| 1 | public abstract class CacheCreation<T> { | |
| 2 | public abstract List<T> getAll(); |

|  |  |
| --- | --- |
| 3 | } |

* Then, we will create an class **UseCaseClass** to use and check the Ehcache containing:  
  \* main method to create several threads soliciting the Ehcache:

|  |  |  |
| --- | --- | --- |
| 01 | public static void main(String[] args) { | |
| 02 | int nbThreads = 3; |

|  |  |  |
| --- | --- | --- |
| 03 | ExecutorService execService = Executors.newFixedThreadPool(nbThreads); | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | // Create several threads which solicit the Ehcache | |
| 06 | for (int i = 0; i < nbThreads; i++) { |

|  |  |  |
| --- | --- | --- |
| 07 | final int indexFinal = i; | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | execService.submit(new Runnable(){ | |
| 10 | String threadName= null; |

|  |  |  |
| --- | --- | --- |
| 11 | UseCaseClass useCaseClass = null; | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | public void run(){ | |
| 14 | try { |

|  |  |
| --- | --- |
| 15 | useCaseClass = new UseCaseClass(); |
| 16 | threadName = "thread\_"+indexFinal; | |

|  |  |  |
| --- | --- | --- |
| 17 | useCaseClass.getAllData1(threadName); | |
| 18 | { |

|  |  |
| --- | --- |
| 19 | int sleepTime = getRandomSleepTime(1000, 5000); |
| 20 | System.out.println(threadName+" will sleep during "+sleepTime+"ms."); |

|  |  |
| --- | --- |
| 21 | Thread.currentThread().sleep(sleepTime); |
| 22 | System.out.println(threadName+" wakes up"); | |

|  |  |
| --- | --- |
| 23 | } |
| 24 | useCaseClass.getAllData2(threadName); | |

|  |  |
| --- | --- |
| 25 | { |
| 26 | int sleepTime = getRandomSleepTime(1000, 5000); | |

|  |  |  |
| --- | --- | --- |
| 27 | System.out.println(threadName+" will sleep during "+sleepTime+"ms."); | |
| 28 | Thread.currentThread().sleep(sleepTime); |

|  |  |  |
| --- | --- | --- |
| 29 | System.out.println(threadName+" wakes up"); | |
| 30 | } |

|  |  |
| --- | --- |
| 31 | useCaseClass.getAllData1(threadName); |
| 32 | useCaseClass.getAllData2(threadName); |

|  |  |
| --- | --- |
| 33 | useCaseClass.getAllData1(threadName); |
| 34 | useCaseClass.getAllData2(threadName); |

|  |  |
| --- | --- |
| 35 | } catch (Throwable e) { |
| 36 | e.printStackTrace(); | |

|  |  |
| --- | --- |
| 37 | } |
| 38 |  | |

|  |  |  |
| --- | --- | --- |
| 39 | }//end-run | |
| 40 |  |

|  |  |
| --- | --- |
| 41 | private int getRandomSleepTime(int min, int max){ |
| 42 | return min + (int)(Math.random() \* ((max - min) + 1)); | |

|  |  |  |
| --- | --- | --- |
| 43 | } | |
| 44 |  |

|  |  |  |
| --- | --- | --- |
| 45 | }//end-runnable | |
| 46 |  |

|  |  |  |
| --- | --- | --- |
| 47 | );//end-submit | |
| 48 |  |

|  |  |  |
| --- | --- | --- |
| 49 | }//end-for | |
| 50 | } |

* \* The Ehcache will be sollicited by 2 methods **getAllData1** and **getAllData2**:

|  |  |  |
| --- | --- | --- |
| 01 | private static final String CACHE\_NAME = "myCache1"; | |
| 02 |  |

|  |  |
| --- | --- |
| 03 |  |
| 04 | public List<String> getAllData1(final String threadName){ | |

|  |  |  |
| --- | --- | --- |
| 05 | return CacheUtil.getListFromCache(threadName, CACHE\_NAME, "data1", newCacheCreation<String>(){ | |
| 06 | @Override |

|  |  |
| --- | --- |
| 07 | public List<String> getAll(){ |
| 08 | System.out.println(threadName+" : UseCaseClass.getAllData1() : the target original method is called to get the values."); | |

|  |  |  |
| --- | --- | --- |
| 09 | List<String> list = new ArrayList<String>(); | |
| 10 | list.add("data1-value1"); |

|  |  |
| --- | --- |
| 11 | list.add("data1-value2"); |
| 12 | list.add("data1-value3"); |

|  |  |  |
| --- | --- | --- |
| 13 | list.add("data1-value4"); | |
| 14 | return list; |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 | }); |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 |  |
| 20 | public List<String> getAllData2(final String threadName){ | |

|  |  |  |
| --- | --- | --- |
| 21 | return CacheUtil.getListFromCache(threadName, CACHE\_NAME, "data2", newCacheCreation<String>(){ | |
| 22 | @Override |

|  |  |
| --- | --- |
| 23 | public List<String> getAll(){ |
| 24 | System.out.println(threadName+" : UseCaseClass.getAllData2() : the target original method is called to get the values."); | |

|  |  |  |
| --- | --- | --- |
| 25 | List<String> list = new ArrayList<String>(); | |
| 26 | list.add("data2-value1"); |

|  |  |
| --- | --- |
| 27 | list.add("data2-value2"); |
| 28 | list.add("data2-value3"); |

|  |  |  |
| --- | --- | --- |
| 29 | list.add("data2-value4"); | |
| 30 | return list; |

|  |  |  |
| --- | --- | --- |
| 31 | } | |
| 32 | }); |

|  |  |
| --- | --- |
| 33 | } |

…So, if we execute the main method:

* with the following parameters in **ehcache.xml** (the cache will expire an element in memory if it is idle for more than 5 minutes and lives for more than 10 minutes):

|  |  |
| --- | --- |
| 01 | <cache name="myCache1" |
| 02 | maxEntriesLocalHeap="10000" | |

|  |  |  |
| --- | --- | --- |
| 03 | maxEntriesLocalDisk="1000" | |
| 04 | eternal="false" |

|  |  |  |
| --- | --- | --- |
| 05 | diskSpoolBufferSizeMB="20" | |
| 06 | timeToIdleSeconds="300" |

|  |  |
| --- | --- |
| 07 | timeToLiveSeconds="600" |
| 08 | memoryStoreEvictionPolicy="LFU" | |

|  |  |
| --- | --- |
| 09 | transactionalMode="off"> |
| 10 | <persistence strategy="localTempSwap"/> | |

|  |  |
| --- | --- |
| 11 | </cache> |

* …we obtain the below results – at the end, the elements ‘data1′ and ‘data2′ have been found in the cache-:

|  |  |
| --- | --- |
| 01 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 02 | thread\_0 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |  |
| --- | --- | --- |
| 03 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. | |
| 04 | thread\_1 will sleep during 3820ms. |

|  |  |  |
| --- | --- | --- |
| 05 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. | |
| 06 | thread\_0 will sleep during 3252ms. |

|  |  |  |
| --- | --- | --- |
| 07 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. | |
| 08 | thread\_2 will sleep during 2626ms. |

|  |  |
| --- | --- |
| 09 | thread\_2 wakes up |
| 10 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has not been found in the cache ---> get the original data. | |

|  |  |
| --- | --- |
| 11 | thread\_2 : UseCaseClass.getAllData2() : the target original method is called to get the values. |
| 12 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element'data2' has been added in the cache. |

|  |  |  |
| --- | --- | --- |
| 13 | thread\_2 will sleep during 3622ms. | |
| 14 | thread\_0 wakes up |

|  |  |  |
| --- | --- | --- |
| 15 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. | |
| 16 | thread\_0 will sleep during 1956ms. |

|  |  |
| --- | --- |
| 17 | thread\_1 wakes up |
| 18 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. | |

|  |  |  |
| --- | --- | --- |
| 19 | thread\_1 will sleep during 2747ms. | |
| 20 | thread\_0 wakes up |

|  |  |
| --- | --- |
| 21 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 22 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

|  |  |
| --- | --- |
| 23 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 24 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

|  |  |
| --- | --- |
| 25 | thread\_2 wakes up |
| 26 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. | |

|  |  |
| --- | --- |
| 27 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |
| 28 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |

|  |  |  |
| --- | --- | --- |
| 29 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. | |
| 30 | thread\_1 wakes up |

|  |  |
| --- | --- |
| 31 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 32 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

|  |  |
| --- | --- |
| 33 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 34 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

* with the following parameters in **ehcache.xml** (the cache will expire an element in memory if it is idle for more than 1 seconds and lives for more than 2 seconds):

|  |  |
| --- | --- |
| 01 | <cache name="myCache1" |
| 02 | maxEntriesLocalHeap="10000" | |

|  |  |  |
| --- | --- | --- |
| 03 | maxEntriesLocalDisk="1000" | |
| 04 | eternal="false" |

|  |  |  |
| --- | --- | --- |
| 05 | diskSpoolBufferSizeMB="20" | |
| 06 | timeToIdleSeconds="1" |

|  |  |
| --- | --- |
| 07 | timeToLiveSeconds="2" |
| 08 | memoryStoreEvictionPolicy="LFU" | |

|  |  |
| --- | --- |
| 09 | transactionalMode="off"> |
| 10 | <persistence strategy="localTempSwap"/> | |

|  |  |
| --- | --- |
| 11 | </cache> |

* …we obtain the below results – at the end, the elements ‘data1′ and ‘data2′ have been found in the cache, but they expire frequently-:

|  |  |
| --- | --- |
| 01 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 02 | thread\_2 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 03 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 04 | thread\_1 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 05 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 06 | thread\_0 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |  |
| --- | --- | --- |
| 07 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. | |
| 08 | thread\_2 will sleep during 3449ms. |

|  |  |  |
| --- | --- | --- |
| 09 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. | |
| 10 | thread\_1 will sleep during 1335ms. |

|  |  |  |
| --- | --- | --- |
| 11 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. | |
| 12 | thread\_0 will sleep during 2558ms. |

|  |  |
| --- | --- |
| 13 | thread\_1 wakes up |
| 14 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has not been found in the cache ---> get the original data. | |

|  |  |
| --- | --- |
| 15 | thread\_1 : UseCaseClass.getAllData2() : the target original method is called to get the values. |
| 16 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element'data2' has been added in the cache. |

|  |  |  |
| --- | --- | --- |
| 17 | thread\_1 will sleep during 3509ms. | |
| 18 | thread\_0 wakes up |

|  |  |
| --- | --- |
| 19 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has not been found in the cache ---> get the original data. |
| 20 | thread\_0 : UseCaseClass.getAllData2() : the target original method is called to get the values. |

|  |  |  |
| --- | --- | --- |
| 21 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element'data2' has been added in the cache. | |
| 22 | thread\_0 will sleep during 1451ms. |

|  |  |
| --- | --- |
| 23 | thread\_2 wakes up |
| 24 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. | |

|  |  |  |
| --- | --- | --- |
| 25 | thread\_2 will sleep during 3111ms. | |
| 26 | thread\_0 wakes up |

|  |  |
| --- | --- |
| 27 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 28 | thread\_0 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 29 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. |
| 30 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

|  |  |
| --- | --- |
| 31 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 32 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

|  |  |
| --- | --- |
| 33 | thread\_1 wakes up |
| 34 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. | |

|  |  |
| --- | --- |
| 35 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has not been found in the cache ---> get the original data. |
| 36 | thread\_1 : UseCaseClass.getAllData2() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 37 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element'data2' has been added in the cache. |
| 38 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |

|  |  |  |
| --- | --- | --- |
| 39 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. | |
| 40 | thread\_2 wakes up |

|  |  |
| --- | --- |
| 41 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has not been found in the cache ---> get the original data. |
| 42 | thread\_2 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 43 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element'data1' has been added in the cache. |
| 44 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has not been found in the cache ---> get the original data. |

|  |  |
| --- | --- |
| 45 | thread\_2 : UseCaseClass.getAllData2() : the target original method is called to get the values. |
| 46 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element'data2' has been added in the cache. |

|  |  |
| --- | --- |
| 47 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has been foundin the cache. |
| 48 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been foundin the cache. |

We change the parameters in **ehcache.xml** (the cache will expire an element in memory if it is idle for more than 5 minutes and lives for more than 10 minutes):

|  |  |
| --- | --- |
| 01 | <cache name="myCache1" |
| 02 | maxEntriesLocalHeap="10000" | |

|  |  |  |
| --- | --- | --- |
| 03 | maxEntriesLocalDisk="1000" | |
| 04 | eternal="false" |

|  |  |  |
| --- | --- | --- |
| 05 | diskSpoolBufferSizeMB="20" | |
| 06 | timeToIdleSeconds="300" |

|  |  |
| --- | --- |
| 07 | timeToLiveSeconds="600" |
| 08 | memoryStoreEvictionPolicy="LFU" | |

|  |  |
| --- | --- |
| 09 | transactionalMode="off"> |
| 10 | <persistence strategy="localTempSwap"/> | |

|  |  |
| --- | --- |
| 11 | </cache> |

… and we change the method **getCache** to override the parameters from **ehcache.xml**:

|  |  |  |
| --- | --- | --- |
| 1 | Ehcache cache = null; | |
| 2 | if(cacheMgr!=null){ |

|  |  |  |
| --- | --- | --- |
| 3 | //cache = cacheMgr.addCacheIfAbsent(name); | |
| 4 | cache = cacheMgr.getEhcache(cacheName); |

|  |  |
| --- | --- |
| 5 |  |
| 6 | //It is possible to override the parameters from ehcache.xml | |

|  |  |
| --- | --- |
| 7 | cache.getCacheConfiguration().setTimeToIdleSeconds(1); |
| 8 | cache.getCacheConfiguration().setTimeToLiveSeconds(2); |

|  |  |
| --- | --- |
| 9 | } |

…we obtain the below results – at the end, the elements ‘data1′ and ‘data2′ have been found in the cache, but they expire frequently-:

|  |  |
| --- | --- |
| 01 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. |
| 02 | thread\_1 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 03 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. |
| 04 | thread\_2 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 05 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. |
| 06 | thread\_0 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |  |
| --- | --- | --- |
| 07 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. | |
| 08 | thread\_0 will sleep during 3295ms. |

|  |  |  |
| --- | --- | --- |
| 09 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. | |
| 10 | thread\_1 will sleep during 1603ms. |

|  |  |  |
| --- | --- | --- |
| 11 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. | |
| 12 | thread\_2 will sleep during 4515ms. |

|  |  |
| --- | --- |
| 13 | thread\_1 wakes up |
| 14 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has not been found inthe cache ---> get the original data. | |

|  |  |
| --- | --- |
| 15 | thread\_1 : UseCaseClass.getAllData2() : the target original method is called to get the values. |
| 16 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element 'data2'has been added in the cache. |

|  |  |  |
| --- | --- | --- |
| 17 | thread\_1 will sleep during 4186ms. | |
| 18 | thread\_0 wakes up |

|  |  |
| --- | --- |
| 19 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has not been found inthe cache ---> get the original data. |
| 20 | thread\_0 : UseCaseClass.getAllData2() : the target original method is called to get the values. |

|  |  |  |
| --- | --- | --- |
| 21 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element 'data2'has been added in the cache. | |
| 22 | thread\_0 will sleep during 1049ms. |

|  |  |
| --- | --- |
| 23 | thread\_0 wakes up |
| 24 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. | |

|  |  |
| --- | --- |
| 25 | thread\_0 : UseCaseClass.getAllData1() : the target original method is called to get the values. |
| 26 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. |

|  |  |
| --- | --- |
| 27 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has not been found inthe cache ---> get the original data. |
| 28 | thread\_0 : UseCaseClass.getAllData2() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 29 | thread\_0 : CacheUtil.getListFromCache() : the original data for the element 'data2'has been added in the cache. |
| 30 | thread\_0 : CacheUtil.getListFromCache() : the element 'data1' has been found in the cache. |

|  |  |  |
| --- | --- | --- |
| 31 | thread\_0 : CacheUtil.getListFromCache() : the element 'data2' has been found in the cache. | |
| 32 | thread\_2 wakes up |

|  |  |  |
| --- | --- | --- |
| 33 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been found in the cache. | |
| 34 | thread\_2 will sleep during 4864ms. |

|  |  |
| --- | --- |
| 35 | thread\_1 wakes up |
| 36 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. | |

|  |  |
| --- | --- |
| 37 | thread\_1 : UseCaseClass.getAllData1() : the target original method is called to get the values. |
| 38 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. |

|  |  |
| --- | --- |
| 39 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has not been found inthe cache ---> get the original data. |
| 40 | thread\_1 : UseCaseClass.getAllData2() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 41 | thread\_1 : CacheUtil.getListFromCache() : the original data for the element 'data2'has been added in the cache. |
| 42 | thread\_1 : CacheUtil.getListFromCache() : the element 'data1' has been found in the cache. |

|  |  |  |
| --- | --- | --- |
| 43 | thread\_1 : CacheUtil.getListFromCache() : the element 'data2' has been found in the cache. | |
| 44 | thread\_2 wakes up |

|  |  |
| --- | --- |
| 45 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has not been found inthe cache ---> get the original data. |
| 46 | thread\_2 : UseCaseClass.getAllData1() : the target original method is called to get the values. |

|  |  |
| --- | --- |
| 47 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element 'data1'has been added in the cache. |
| 48 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has not been found inthe cache ---> get the original data. |

|  |  |
| --- | --- |
| 49 | thread\_2 : UseCaseClass.getAllData2() : the target original method is called to get the values. |
| 50 | thread\_2 : CacheUtil.getListFromCache() : the original data for the element 'data2'has been added in the cache. |

|  |  |
| --- | --- |
| 51 | thread\_2 : CacheUtil.getListFromCache() : the element 'data1' has been found in the cache. |
| 52 | thread\_2 : CacheUtil.getListFromCache() : the element 'data2' has been found in the cache. |

<http://www.javacreed.com/caching-made-easy-with-spring/>

## Caching with Spring

Consider the following class.

package com.javacreed.examples.sc.part1;

import org.springframework.stereotype.Component;

@Component

public class Worker {

public String longTask(final long id) {

System.out.printf("Running long task for id: %d...%n", id);

return "Long task for id " + id + " is done";

}

public String shortTask(final long id) {

System.out.printf("Running short task for id: %d...%n", id);

return "Short task for id " + id + " is done";

}

}

Here we have a simple Spring component class that has two methods. One method, named longTask(), carries a fictitious long task, while the second method, named shortTask(), runs quickly. The output of both methods is only determined by the input of these methods. Therefore, for the same input we will always get the same output. **This is very important, as otherwise we cannot apply caching**.

Now consider the following class.

package com.javacreed.examples.sc.part1;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(final String[] args) {

final String xmlFile = "META-INF/spring/app-context.xml";

try (ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlFile)) {

final Worker worker = context.getBean(Worker.class);

worker.longTask(1);

worker.longTask(1);

worker.longTask(1);

worker.longTask(2);

worker.longTask(2);

}

}

}

Here we are creating the Spring environment and retrieving an instance of our Worker from Spring. Then we invoke the methodlongTask() five times. This will produce the following output to the command prompt.

Running long task for id: 1...

Running long task for id: 1...

Running long task for id: 1...

Running long task for id: 2...

Running long task for id: 2...

Note that the method longTask() runs five times, one for every request. Also note that this method only received two different inputs. This method is invoked three times with the parameter value: 1 and twice with the parameter value: 2. Since the output of this fictitious long task method is only determined by its input, we can cache the output and on the next request for this input we use the cached value instead of re-running this fictitious long task.

In order to apply caching we need to do the following three things.

1. **Mark the methods (or classes) which output will be cached**.  
   Spring 3.1 added new annotations that enable method caching as shown next.
2. package com.javacreed.examples.sc.part1;
3. **import org.springframework.cache.annotation.Cacheable;**
4. import org.springframework.stereotype.Component;
5. @Component
6. public class Worker {
7. **@Cacheable("task")**
8. public String longTask(final long id) {
9. System.out.printf("Running long task for id: %d...%n", id);
10. return "Long task for id " + id + " is done";
11. }
12. public String shortTask(final long id) {
13. System.out.printf("Running short task for id: %d...%n", id);
14. return "Short task for id " + id + " is done";
15. }
16. }

By simply adding the @Cacheable annotation ([Java Doc](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/cache/annotation/Cacheable.html)) to the method signature, repetitive requests to this method, with the same parameter value, will simply return the cached value. Spring allows us to cache values without having to write the boilerplate-code that handles that. Note that this annotation also takes a value, which is the name of the cache repository. We will talk about the cache repository in a minute.

#### Observation

1. **Enabled Spring caching**.  
   Before Spring can start caching our values, we need add the following declaration.
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:context="http://www.springframework.org/schema/context"
6. xmlns:cache="http://www.springframework.org/schema/cache"
7. xmlns:p="http://www.springframework.org/schema/p"
8. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
9. http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache-3.2.xsd
10. http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd">
11. <context:annotation-config />
12. <context:component-scan base-package="com.javacreed.examples.sc" />
13. <!-- Enables the caching through annotations -->
14. **<cache:annotation-driven />**
15. </beans>

With this declaration, Spring will look for any classes or methods that are marked cacheable and will take all necessary actions in order to provide caching.

1. **Configure the caching repository to be used**.  
   Before this code can work, we need to define the cache repository as shown next.
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:context="http://www.springframework.org/schema/context"
6. xmlns:cache="http://www.springframework.org/schema/cache"
7. xmlns:p="http://www.springframework.org/schema/p"
8. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
9. http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache-3.2.xsd
10. http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd">
11. <context:annotation-config />
12. <context:component-scan base-package="com.javacreed.examples.sc" />
13. <!-- Enables the caching through annotations -->
14. <cache:annotation-driven />
15. <!-- Generic cache manager based on the JDK ConcurrentMap -->
16. **<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">**
17. **<property name="caches">**
18. **<set>**
19. **<bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="task" />**
20. **</set>**
21. **</property>**
22. **</bean>**
23. </beans>

The cache repository is where the actual objects are saved. Spring supports two types of repositories: one based on JDK ConcurrentMap and the other on [ehcache](http://ehcache.org/" \t "_blank) popular library. More can be added. Here we are using the JDK ConcurrentMap as our cache repository. The repository plays little (if any) effect on the code and switching between repositories should be very easy. Our objects will be cached within a ConcurrentMap. In this example we added one cache repository, named task. We can have more than one repository. Note that the name of this repository is the same as the name showed in the annotation before.

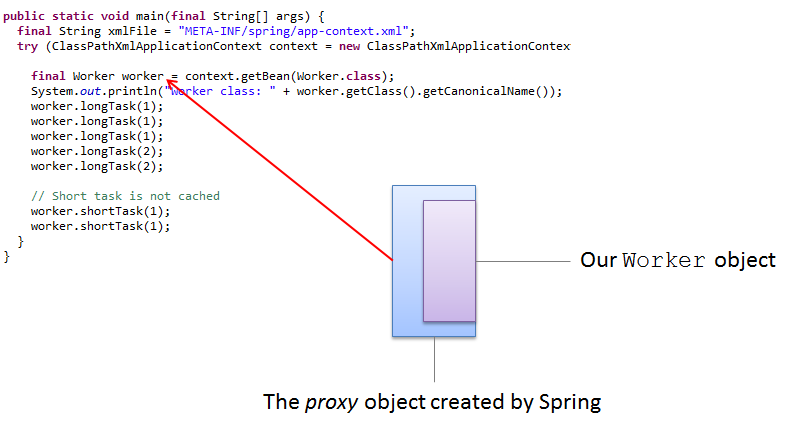
#### Observation

Please note that the JDK ConcurrentMap class shown in the above declaration varies between versions of Spring 3.1 and 3.2. Here we are using Spring 3.2. In version 3.1 the class name is as shown below.

org.springframework.cache.concurrent.ConcurrentCacheFactoryBean

**How does this work?**

Spring, when configured to use caching, will wrap the objects that are marked to be cached in to a proxy. The caller will not work with our object but with the proxy instead as shown in the following figure.

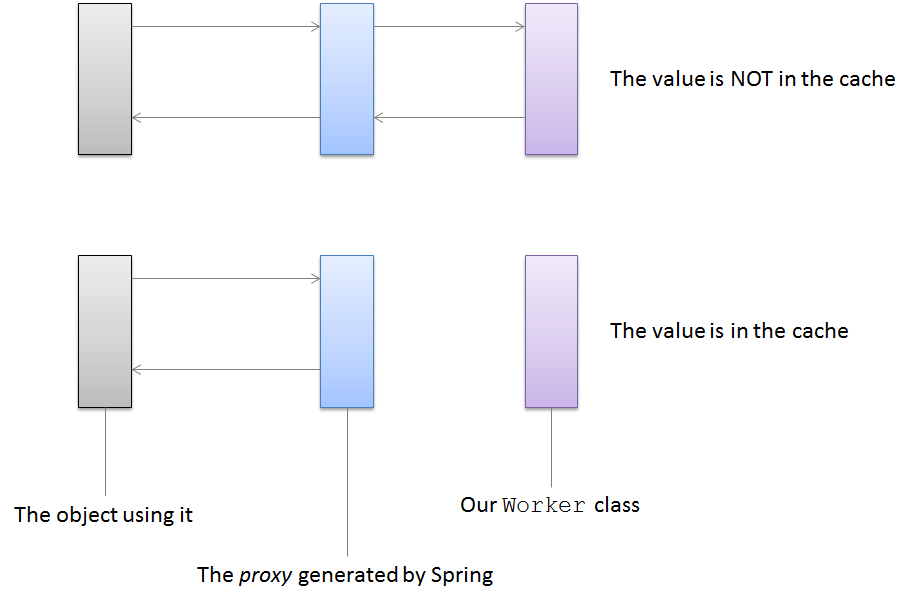
[](http://www.javacreed.com/caching-made-easy-with-spring/spring-cache-example-1/)

*Caching Proxy*

If we had to print the class’s canonical name of the object returned by the Spring environment, we will see the following.

Worker class: com.javacreed.examples.sc.part1.Worker$$EnhancerByCGLIB$$4fa6f80b

Note that this is not the Worker class (canonical name: com.javacreed.examples.sc.part1.Worker) we created but some other class. In fact, this class was generated by Spring using code generation techniques, which are not discussed here. When we invoke any methods from the Worker class, instead we are calling the methods in the generated proxy. This proxy holds an instance of our Workerclass. It will forward any request to our object and return its response as shown in the following figure. If the method is marked as cacheable, then the proxy will bypass the request and will return the cached value instead. If the proxy does not have a cached value yet for the given input, it makes the request and saves the response for future use.

[](http://www.javacreed.com/caching-made-easy-with-spring/spring-cache-example-2/)

*Proxy Flow of Control*

If we run again our Main class, we will get the following output.

Running long task for id: 1...

Running long task for id: 2...

Here the fictitious long task method (longTask()) is actually invoked twice. The proxy returned the cached result for the other times. This concludes our first section about Spring caching. As we saw, it is quite easy to enable. All we need to do is follow the three steps listed before, and we have caching. In the next section we will see how we can apply caching with recursion.

## Caching a recursive method

Consider the following class.

package com.javacreed.examples.sc.part2\_1;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Component;

@Component("fibonacci")

public class Fibonacci {

private int executions = 0;

public int getExecutions() {

return executions;

}

public void resetExecutions() {

this.executions = 0;

}

@Cacheable("fibonacci")

public long valueAt(final long index) {

executions++;

if (index < 2) {

return 1;

}

return valueAt(index - 1) + valueAt(index - 2);

}

}

This class implements [Fibonacci](http://en.wikipedia.org/wiki/Fibonacci_number) sequence and returns the Fibonacci number at a given index. The Fibonacci number is calculated recursively using the function: fib(n) = fib(n-1) + fib(n-2). The base case for this recursive function is that the first two Fibonacci numbers are 1.

Note that this class also keeps track of how many times the valueAt() method is invoked. We can obtain this value through the getter method. The Fibonacci class also enabled resetting of this value so that the counter starts from 0 once more.

Here everything seems to be in place and the class is enabled for caching. So let us execute it.

package com.javacreed.examples.sc.part2\_1;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(final String[] args) {

final String xmlFile = "META-INF/spring/app-context.xml";

try (ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlFile)) {

final long start = System.nanoTime();

final Fibonacci sequence = context.getBean("fibonacci", Fibonacci.class);

final long fibNumber = sequence.valueAt(5);

final int executions = sequence.getExecutions();

final long timeTaken = System.nanoTime() - start;

System.out.printf("The 5th Fibonacci number is: %d (%,d executions in %,d NS)%n", fibNumber, executions,

timeTaken);

}

}

}

If we run the above code, we will get the following output.

The 5th Fibonacci number is: 8 (15 executions in 17,762,022 NS)

The cacheable method valueAt() was invoked a total of 15 times. This does not sound right. The valueAt() method should have been executed only 6 times and not 15 times. The other times 9 times, the cached value should have been returned instead.

**What went wrong?**  
In the main() method we obtained an instance of the Fibonacci class through Spring. In turn, Spring wrapped our object into a proxy. Therefore within the main() method, we only have access to the proxy. But the valueAt() method within the Fibonacci class, calls itself (recursion). This is not calling the valueAt() method through the proxy, but directly from Fibonacci class. Therefore the proxy is bypassed. That is why we are not caching the value at the recursion level.

#### Observation

Note that if we had to invoke the sequence.valueAt(5); (with the same value) again, the cached value will be returned as the variablesequence is an instance of the proxied Fibonacci.

**How can we fix this?**  
In order to fix this we need to modify the Fibonacci class and pass a reference of our proxy as shown next.

package com.javacreed.examples.sc.part2\_2;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Component;

@Component("fibonacci2")

public class Fibonacci {

private int executions = 0;

public int getExecutions() {

return executions;

}

public void resetExecutions() {

this.executions = 0;

}

@Cacheable("fibonacci")

public long valueAt(final long index**, final Fibonacci callback**) {

executions++;

if (index < 2) {

return 1;

}

return **callback.**valueAt(index - 1**, callback**) + **callback.**valueAt(index - 2**, callback**);

}

}

Note that now our valueAt() method takes two parameters and not one. It takes an instance of the Fibonacci class, referred to ascallback. Furthermore, instead of invoking the valueAt() on itself, it invokes the callback‘s valueAt(). From the main() method, we need to also pass an instance of the proxied Fibonacci class as shown in the following example.

package com.javacreed.examples.sc.part2\_2;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(final String[] args) {

final String xmlFile = "META-INF/spring/app-context.xml";

try (ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlFile)) {

final long start = System.nanoTime();

final Fibonacci sequence = context.getBean("fibonacci2", Fibonacci.class);

final long fibNumber = sequence.valueAt(5**, sequence**);

final int executions = sequence.getExecutions();

final long timeTaken = System.nanoTime() - start;

System.out.printf("The 5th Fibonacci number is: %d (%,d executions in %,d NS)%n", fibNumber, executions,

timeTaken);

}

}

}

This class will produce the following output.

The 5th Fibonacci number is: 8 (6 executions in 18,320,003 NS)

#### Observation

This concludes our example with caching and using recursion. The key point is that recursion will bypass the proxy unless a callback is specified. In the following section we will see a simple, real world like, example.

## Real-world like example

Consider the following domain class.

package com.javacreed.examples.sc.part3;

public class Member {

private final int memberId;

private final String memberName;

public Member(final int memberId, final String memberName) {

this.memberId = memberId;

this.memberName = memberName;

}

// Getters removed for brevity

@Override

public String toString() {

return String.format("[%d] %s", memberId, memberName);

}

}

This is a simple class that represents a member having only an id, which uniquely identifies a member and a name. For simplicity, the members are persisted in a text file of the format shown next.

1,Albert Attard

2,Mary Borg

3,Tony White

4,Jane Black

Now consider the following service interface.

package com.javacreed.examples.sc.part3;

public interface MembersService {

Member getMemberWithId(int id);

void saveMember(Member member);

}

This interface exposed two methods one used to retrieve the member with the given id and the other to persist any modifications made to file. The following class starts the whole thing and makes several request to the MembersService implementation. Note that the implementation is cacheable and therefore we are working with the proxy version of the MembersService.

package com.javacreed.examples.sc.part3;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(final String[] args) {

final String xmlFile = "META-INF/spring/app-context.xml";

try (ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(xmlFile)) {

final MembersService service = context.getBean(MembersService.class);

// Load member with id 1

Member member = service.getMemberWithId(1);

System.out.println(member);

// Load member with id 1 again

member = service.getMemberWithId(1);

System.out.println(member);

// Edit member with id 1

member = new Member(1, "Joe Vella");

service.saveMember(member);

// Load member with id 1 after it was modified

member = service.getMemberWithId(1);

System.out.println(member);

}

}

}

If we execute the above, the following is produced to the command prompt.

Retrieving the member with id: [1] from file: C:\javacreed\spring-cache\members.txt

[1] Albert Attard

[1] Albert Attard

Retrieving the member with id: [1] from file: C:\javacreed\spring-cache\members.txt

[1] Joe Vella

Here we made two requests to retrieve the member with id 1, but the method was actually invoked once. In the second request, the cached value was returned. Then we modified the member with the same id. Since the member was modified, the cache was invalidated (we will see how shortly). Thus when retrieving the member with the same id, we invoked the actual method again and load it from file. This value will be cached until invalidated again.

Now let us see how this is implemented. The getMemberWithId() is similar to the other methods we saw already. It is annotated with the@Cacheable annotation.

@Override

@Cacheable("members")

public Member getMemberWithId(final int id) {

System.out.printf("Retrieving the member with id: [%d] from file: %s%n", id, dataFile.getAbsolutePath());

// code removed for brevity

}

The saveMember() needs to invalidate the cache. In order to achieve this, Spring provides another annotation named: @CacheEvict([Java Doc](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/cache/annotation/CacheEvict.html)) as shown next.

@Override

**@CacheEvict(value = "members", allEntries = true)**

public void saveMember(final Member member) {

// code removed for brevity

}

Whenever this method is invoked, the cache repository named members is cleared from all members (as instructed by the: allEntries = true annotation optional parameter). Therefore, next time the getMemberWithId() method is invoked it will have to load the member from file, thus reading the new changes. Without this, the getMemberWithId() method will still return the old version of the member with id 1.

This concludes our article about Caching with Spring. In this article we saw how we can leverage existing API in order to cache results without having to write any boilerplate code.