And here is a relatively standard sample usage of the library.

public override List<Employee> GetEmployeeList()

{

string key = ConfigurationHelper.CacheKeyEmployeeList;

List<Employee> employees = CacheHelper.Get<List<Employee>>(key);

if (employees == null)

{

employees = instance.GetEmployeeList();

CacheHelper.Add(employees, key);

}

return employees;

}

Notice how I’m grabbing the cached value, storing it in a local variable and then checking if it is equal to null rather than using the CacheHelper.Exists() method.  If I used the CacheHelper.Exists() method, the cached object could expire between the time I check its existence and the time I get its value through the CacheHelper.Get() method.  Therefore, the above approach is the safest strategy to use when retrieving cached values.  CacheHelper.Exists() should really only be used for quick existence checks which are unrelated to the fetch.

But if you want to use the code CORRECTLY there’s a catch.  Did you notice the “class” constraint on the CacheHelper.Get() and CacheHelper.Add() methods?   I did this because you can’t always return null from a generic method.  If the return type were always a reference type it would be fine, but comparing a non-nullable value type to null would throw a runtime exception or would always evaluate to false.  Therefore, I’ve constrainted CacheHelper which limits its functionality but unsure safe use of the cache.  If you’re feeling dangerous, you’re welcome to remove the constraints.