29. What is caching?

Brief summary: Caching is a mechanism that allows storing some data in memory, so next time it is needed, it can be served faster.

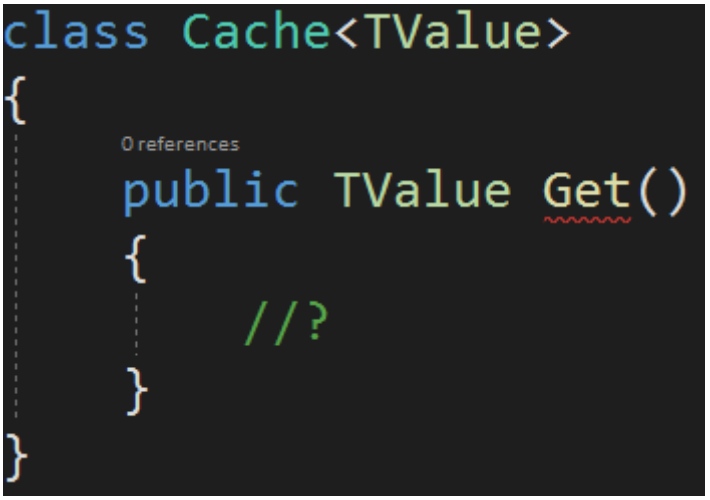
Caching is a mechanism that allows storing some data in memory, so next time it is needed, it can be served faster. To understand it better, let's consider this class:

```
class PeopleController
{
    private readonly IRepository<Person> _peopleRepository;

    1 reference
    public PeopleController(IRepository<Person> peopleRepository)
    {
        _peopleRepository = peopleRepository;
    }

    3 references
    public Person? GetByName(string firstName, string lastName)
    {
        return _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault();
    }
}
```

This class is responsible for retrieving a Person object from some repository using a person's first and last name. This code is very simple, but we must be aware of one thing - in a real-life project, accessing data from an external source may be **slow**. Maybe the class implementing the IRepository interface connects to some bulky database, or retrieves data from some API? It may be the case that every call to this external data source takes some considerable time, and if many calls are executed, the application may start to work slowly. We don't want that.

How can we make it better? Well, one solution could be this: if we already accessed the data for some particular first and last name - for example, John Smith - we could store it in the application memory and next time when we want to find John Smith, we will access his data immediately, instead of asking the external system to provide it again. This mechanism is called **caching**. We store some data in a cache, so a piece of memory of the application, making it available immediately. Let's

implement a very simple caching mechanism. We will start with defining a cache class.



I made this class generic, so it can store any type. In our case, it will store objects of type Person.

Now we need some kind of container to store the cached Person objects. What data structure would be the best for us? We will want to retrieve them by providing first and last names. So perhaps the best choice would be a Dictionary in which a tuple of two strings (for both names) would be the key, and the Person object would be the value.

```
class Cache<TValue>
{
    private readonly Dictionary<(string, string), TValue> _cachedData = new();

    0 references
    public TValue Get()
    {
        //?
    }
}
```

As you can see I used a ValueTuple for the key because for Dictionary keys we want to have a value-based equality comparison (If I used regular tuple, which is a reference type, two tuples holding the same first and last name would not be considered the same key by the Dictionary).

We want to keep the Cache generic, so the Dictionary key type should also be parameterized:

```
class Cache<TKey, TValue> where TKey: notnull
{
    private readonly Dictionary<TKey, TValue> _cachedData = new();

    0 references
    public TValue Get()
    {
        //?
    }
}
```

As you can see I also added the **notnull** constraint. The Dictionary keys should never be null.

Now about the Get method. First of all, it should take a key as a parameter:

If the Dictionary already stores the value with a given key, it should simply be returned:

```
public TValue Get(TKey key)
{
    if(_cachedData.ContainsKey(key))
    {
        return _cachedData[key];
    }
}
```

But if not, it means the value is not yet cached and we need to access it somehow. In the case of the PeopleController, it would be read from the repository. But we can't do it like this:

```
if(_cachedData.ContainsKey(key))
{
    return _cachedData[key];
}
else
{
    _cachedData[key] = _peopleRepository
        .GetByName(firstName, lastName)
        .FirstOrDefault();
}
```

The Cache class is generic and it must work not only for People read from some specific repository but for anything else too. In other words, we must also provide some generic mechanism for reading the values that will be then stored in the cache. The simplest solution is to pass a Func to the Get method. This Func will return an object of the TValue type. It will be up to the caller of the Get method to provide a specific method of retrieval. In our case, the PeopleController will be using the cache, and it will pass a function reading the Person object from the database to the Get method.

```
0 references
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if(_cachedData.ContainsKey(key))
    {
        return _cachedData[key];
    }
    else
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    return _cachedData[key];
}
```

We can now simplify this code:

```
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if (!_cachedData.ContainsKey(key))
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    return _cachedData[key];
}
```

Great. This is exactly what we wanted. The value is being read only once. It is stored in the Dictionary, and next time it will be needed it will be retrieved from it. Let's now use the Cache class in the PeopleController. First of all, I will declare and initialize the private Cache field in this class:

```
class PeopleController
{
    private readonly Cache<(string, string), Person?> _cache = new();
```

Now I can use it in the GetByName method. As a reminder: this is how this method looks without using caching:

```
public Person? GetByName(string firstName, string lastName)
{
    return _peopleRepository
        .GetByName(firstName, lastName)
        .FirstOrDefault();
}
```

And this is how it changes when the Cache is used:

```
public Person? GetByName(string firstName, string lastName)
{
    return _cache.Get(
        (firstName, lastName),
        () => _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault());
}
```

The first parameter of the Get method is the key, in our case a ValueTuple holding first and last name. The second parameter is a function that retrieves the Person with those names from the repository.

Let's test if it works as expected. If it does, the GetByName method from the repository should be called only once when I try to access the John Smith object two times.

```
var john = peopleController.GetByName("John", "Smith");
john = peopleController.GetByName("John", "Smith");
```

I've set a breakpoint in the GetByName method:

```
class PeopleRepositoryMock : IRepository<Person>
{
    2 references
    public IEnumerable<Person> GetByName(string firstName, string lastName)
    {
        if(firstName == "John" && lastName == "Smith")
        {
            return new[] { new Person("John", "Smith") };
        }
        throw new NotImplementedException();
    }
}
```

Now, when I run this program, I will see that we only hit this breakpoint once. In a real-world application, we could ask for John Smith hundreds of times and instead of asking the database or some API for his data this many times, we will only ask once.

All right. We implemented a super simple cache ourselves. Of course, we could make this cache much more complex. For example, we don't actually remove anything from the cache now, and it may happen that during the execution of the program it will grow extremely big and will finally drain our application of free memory. Usually, some kind of cleanup mechanism is added, that removes the data that is older than some specified time. Also, this cache is not thread-safe, so it shouldn't be used in multithreaded applications.

There are of course some existing libraries that provide the caching mechanisms. For example, we can use NuGet to install **Microsoft.Extensions.Caching.Memory**. It works very similarly to the cache we implemented. This is how the PeopleController would look like if we used Microsoft's implementation of the cache:

```
private readonly MemoryCache _memoryCache =
    new MemoryCache(new MemoryCacheOptions());

//alternatively we can use the Microsoft's MemotyCache
0 references
public Person? GetByNameMemoryCache(string firstName, string lastName)
{
    return _memoryCache.GetOrCreate(
        (firstName, lastName),
        cacheEntry => _peopleRepository
            .GetByName(firstName, lastName)
            .FirstOrDefault());
}
```

As you can see it's almost exactly the same as our code, and I highly recommend using this package. Nevertheless, I wanted to show you how to implement a cache on our own, so you understand how it works under the hood.

From other caching tools you should be aware of, one of the most commonly used third-party tools is called **Redis**. It provides more functionality than a regular cache and it's known for its excellent performance.

All right. Before we wrap up a word of caution. Caching is great, but for some scenarios only. If we don't retrieve the data identified by the same key repeatedly, but we keep using different keys all the time, it doesn't really give us anything. You can always check the cache success rate by adding a simple field to your cache:

```
private int _cacheSuccessRate;

1 reference
public TValue Get(TKey key, Func<TValue> getValueForTheFirstTime)
{
    if (!_cachedData.ContainsKey(key))
    {
        _cachedData[key] = getValueForTheFirstTime();
    }
    else
    {
        _cacheSuccessRate++;
    }
    return _cachedData[key];
}
```

This counter will be incremented each time we use the cached data instead of retrieving it from the data source. You can check its value during debugging and

see if your cache is successful. The higher the counter goes, the better your cache is performing in your application.

So, use caching when it really makes sense.

Caching is most often used to retrieve data from some external sources, but remember that even the data calculated locally can be cached. For example, if your program does some performance-costly mathematical operations that take a lot of time, you could consider caching the results too.

Also, remember that the underlying data can change after it has been first retrieved by the cache. It means the cache will be providing us with **stale** data. That's why caching is best when the underlying data doesn't change often. In this case, it's enough to have some mechanism that removes the piece of data from the cache after some specific time has passed.

Bonus questions:

- "**What are the benefits of using caching?**"
  Caching can give us a performance boost if we repeatedly retrieve data identified by the same key. It can help not only with data retrieved from an external data source but even calculated locally if the calculation itself is heavy (for example some complex mathematical operations).

- "**What are the downsides of using caching?**"
  Cache occupies the application's memory. It may grow over time, and some kind of cleanup mechanism should be introduced to avoid OutOfMemoryExceptions. Such mechanisms are usually based on the expiration time of the data. Also, the data in the cache may become stale, which means it changed at the source but the old version is cached and used in the application. Because of that, caching is most useful when retrieving data that doesn't change often.