CS2103 2018 B-Term -- Project 2 -- LRU Cache

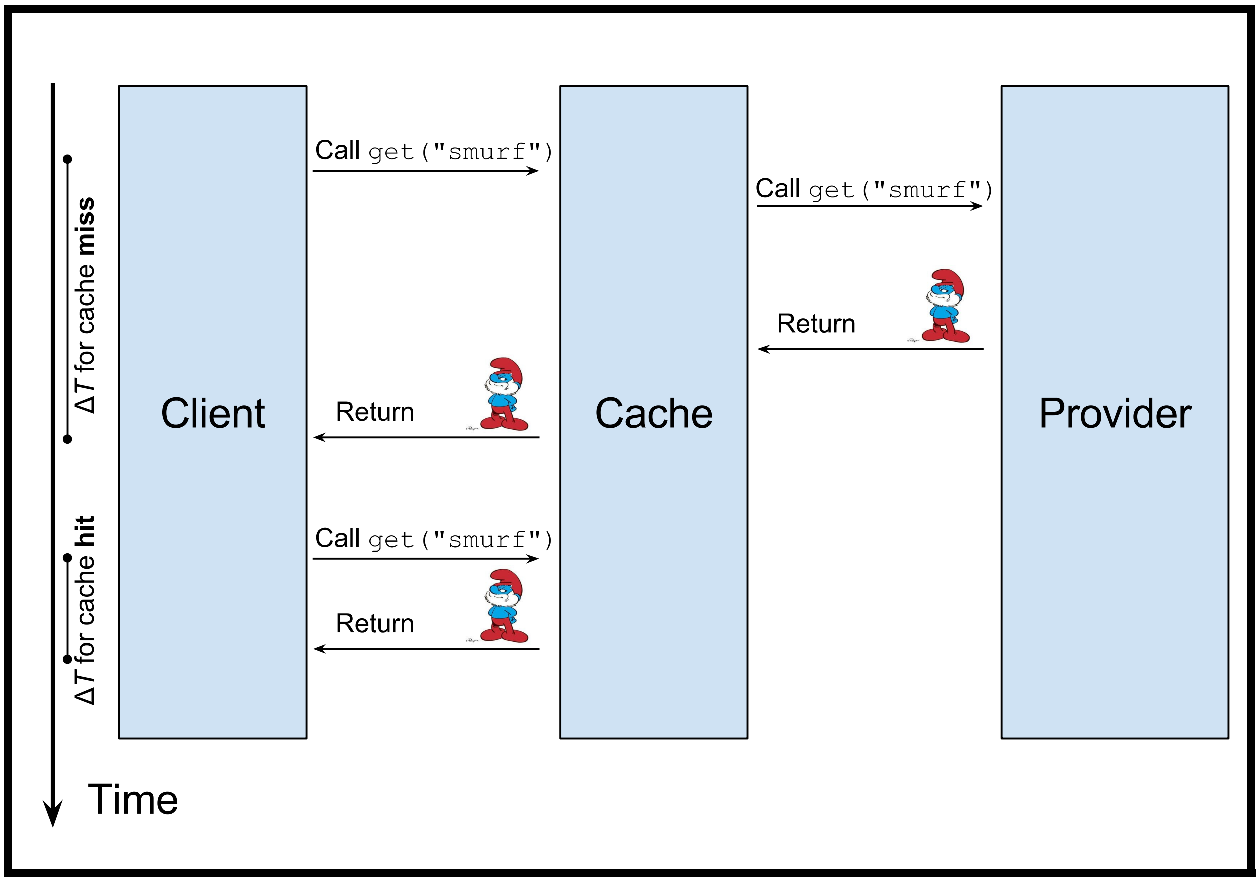
Prof. Jacob Whitehill

Introduction

In this project you will use object-oriented design techniques to implement and encapsulate an abstract data type (ADT) known as a **cache**. A cache is a data structure that provides **accelerated access** to a relatively small **subset** of data that normally take a relatively long time to access. Caches are **associative** such that the user/client (we will use both names interchangeably) requests the **value** associated with a certain **key**. For example, in a web cache, a key might be the URL of a page on a web server, and the associated value might be the contents of the web page itself. For an image cache that allows the client to access frequently used images more quickly than having to load them from disk, the key might be the name of the image (e.g., "smurf"), and the value would be the image itself (e.g., the bytes contents of a PNG file). A cache serves as an **intermediary** between the user/client and a **data provider** -- the web server or file system, in our two examples above. By using a cache to store a small subset of data in faster storage, the client application can often be accelerated significantly. Caches are ubiquitous in modern computer design; CPU caches, disk caches, web caches, etc., are all commonplace.

By definition, caches can only store a **subset** of all possible (key,value) combinations that a user might request. Why? The reason is that caches provide **accelerated** access to data -- if they could contain an infinite number of data and provide them faster than the data provider (the web server in our example above), then there would be no need for the data provider at all -- you would just store everything in the cache. The fact that caches can only store a relatively small number of (key,value) pairs compared to the set of all possible such pairs that the user might request, means that the implementer of the cache has to think carefully about **which** data should be stored. In addition, the implementer of the cache must ensure that access to data is as fast as possible -- determining whether or not a (key,value) pair is already stored in the cache should happen in **constant time**. Recall that **constant time** means that the amount of time **does not depend on the number of data in the cache**.

How caches work



The figure above provides a schematic for how caches work. Suppose you wish to cache a large collection of images in RAM so that your program doesn't have to load them from disk every time it needs to display them. The "data provider" in this example would thus be the disk (the definitive location where all the images are stored). Whenever the client needs to access a particular image, it calls the `get` method of the cache and specifies the name of the image that it wants -- "smurf", in this example.

Cache miss

In the cache's implementation of the `get` method, the first thing it does is check whether an object with a key of "smurf" is already stored in the cache. Let's suppose the cache is empty, so that "smurf" is not found. In this case, we have a **cache miss** -- a key requested by the client was not found in the cache. Whenever the cache has a miss, it must forward the `get` request to (i.e., call the `get` method of) the data provider. While the data provider has access to all possible images the client might ever request, it is also relatively **slow** compared to the cache -- disk (even SSD) is typically slower than RAM, for example. Once the data provider has determined the value that is associated with the key -- i.e., in our example, loaded the image with the specified name from disk -- it returns the result back to the cache. The cache then stores the (key,value) pair in its own internal memory -- which, by assumption, is much faster than whatever storage is used by the data provider -- and then returns the result back to the client. All in all, the amount of time taken by the cache to execute the `get` method is represented by "Delta T for cache miss" in the figure.

Cache hit

Now, let's suppose that, at some time in the near future, the client requests the "smurf" image **again** from the cache. Since not much time has passed (and, more to the point, not many other other images have been requested from the cache), it is likely that the "smurf" image is still stored in the cache. In this case, the cache can return the value associated with the key "smurf" **directly -- without needing to consult the data provider**. The total time needed to service the client's `get` request is now much reduced -- represented in the figure by "Delta T for cache hit". **This is the whole point of the cache -- hopefully, for a large fraction of requested keys, the associated values can be retrieved more quickly than by consulting the data provider itself.**

Eviction

Whenever the cache has a miss, it needs to consult the data provider, store the (key,value) pair in its own storage, and then return the value to the client. Keep in mind that the cache's internal storage is finite (and usually much smaller compared to the data provider's). If the cache's storage is full, some **existing** (key,value) pair must be **evicted** from the cache and replaced with the new pair. But which (key,value) pair should be replaced?

There is no definitive answer to this question, and different programmers have different strategies for trying to determine which will give the highest performance. In general, you want to pick your **eviction strategy** so as to minimize the expected number of cache misses. In this assignment, you will implement a commonly used heuristic known as **least-recently used (LRU)** -- it assumes that the (key,value) pair that was requested **least recently** will likely not be requested in the near future. In other words, when you need to evict an element stored in the cache's internal memory, you should evict the item that was least recently used.

Here's an example of how the LRU eviction strategy works. Suppose the cache has enough internal memory to store just 2 (key,value) pairs, and suppose that the following sequence of keys is requested by the client: `smurf garfield marge snoopy garfield snoopy`. How would the cache's internal memory look when serving each `get` request, and does the cache hit or miss for each request? The answers are shown in the table below:

Requested key	Cache's memory before request	Hit or miss	Eviction
smurf	-, -	miss	-
garfield	smurf, -	miss	-
marge	smurf, garfield	miss	smurf
snoopy	marge, garfield	miss	garfield
garfield	marge,snoopy	miss	marge
snoopy	garfield,snoopy	hit	-

Implementing an LRU cache in Java

In this project, you will implement and encapsulate an LRU cache in a Java class called `LRUCache`. You will also write test code both to verify that **your own** cache is written correctly **and** to catch errors in other people's possibly incorrectly written (buggy) caches. You will be graded on the correctness of both the cache and the cache tester.

The `LRUCache` class you create must have a public constructor that takes two parameters. The first is the `DataProvider` that the cache will consult for every cache miss. How exactly is the data provider defined? We want to be flexible, so we will define a data provider as any Java class that implements a single method: `U get (T key)`. Notice that, in contrast to `ArrayList`, which takes only one type parameter, our cache takes two type parameters: Here, `T` is the type of the key, and `U` is the type of the value associated with the key. For example, `T` might be `String` (the name of the image), and `U` might be `Image`. See the `DataProvider` interface in the project Zip file.

In addition to specifying the data provider, you also have to specify the **capacity** of the cache, which is equal to the maximum number of elements (i.e., (key,value) pairs) that the cache can store at one time. It is **crucial** that your cache allocate exactly enough memory to accomodate `capacity` memory slots -- **no more and no less** -- so that our automatic test code will evaluate your cache correctly.

The only methods that your LRU cache needs to implement are `U get (T key)` and `int getNumMisses ()`. The `get` method has the exact same signature as in the `DataProvider` interface -- indeed the `Cache` interface is actually a subinterface of `DataProvider`. The `getNumMisses` reports the number of cache "misses" that have occurred **since the Cache was instantiated**.

Usage example

Here is an example of how the cache might be used: First, there needs to be a data provider. In our image cache example, this might look something like this:

```
public class ImageLoader implements DataProvider<String, Image> {
        private final String _imageRootDirectory;
        public ImageLoader (String imageRootDirectory) {
                _imageRootDirectory = imageRootDirectory;
        }
        public Image get (String key) {
                return loadImageFromDiskAtDirectory(key, imageRootDirectory);
        }
}
```

Next, we instantiate an `LRUCache` object and pass it an `ImageLoader` as the data provider along with an initial capacity (I chose 128 arbitrarily):

```
final Cache<String, Image> cache = new LRUCache<String, Image>(new ImageLoader("pathToMyImages"), 128);
```

Now that the cache has been instantiated, we can use it:

```
final Image smurf = cache.get("smurf");
final Image garfield = cache.get("garfield");
// ...
```

Note that this was just an example. While you are free to test your cache with whatever objects you like, I would suggest something simpler than actually loading images from disk.

LRU eviction policy

The hardest part of this assignment is implementing the LRU eviction policy so that it executes in **constant time** in the **average case**. In particular, your `LRUCache` should **not** just do a linear search through the cache's internal memory looking for (and updating) the least-recently used item -- you must devise a faster procedure. We suggest that you creatively use both a **linked list** and a **hashtable** to do so. Accordingly, **you are welcome to use Java's built-in `HashMap` class**, which provides an off-the-shelf, high-performance hashtable. See the Javadoc on `HashMap` for more information.

Important: In this assignment you **may not** use the Java SDK `LinkedHashMap` class (as that would ruin all the fun).

Testing an LRU cache in Java

In this project, you will be graded not only on your *implementation* of the `LRUCache`, but also on how thoroughly you *test* it. Your test code should hopefully help you debug your own cache implementation; in addition, we will also **test your tester** by using it to validate *other* students' cache implementations. In particular, we will use your submitted `CacheTest` to test a variety of `LRUCache` implementations -- some of which are buggy, some of which are fully correct -- and give you credit according to how accurately your tester reports errors.

During your testing, you should create your own `DataProvider` -- whose inner workings you can observe and control completely since you are writing it -- that you should instantiate and pass to the `LRUCache` constructor.

As noted in class, it is not mathematically possible to test for all possible errors. We will test your tester against a set of errors that commonly occur when students try to implement an (but fail). Also: your tester is **not** required to verify that the asymptotic time-costs of the `LRUCache` methods are constant. As long as the public methods of the `LRUCache` you are testing operate correctly, then your tester should consider them "correct".

Requirements

1. **R1 (35 points)**: Implement the `Cache` interface in an `LRUCache` class that implements a least-recently-used (LRU) eviction policy:
   - The `LRUCache` class should contain a public constructor with the following signature: `public LRUCache (DataProvider provider, int capacity)`.
   - Your `LRUCache` must contain **exactly** the number of (key,value) pairs specified in the constructor's `capacity` parameter. Our automatic testing code will verify that the (key,value) pairs that are evicted -- and thus must be reloaded from the `DataProvider` -- are exactly correct according to the LRU policy described above.

   - The `get` method of your `LRUCache` must execute in **constant time** in the **average case**. This is actually the crux of the assignment -- devising an algorithm that can find the LRU item, update it, and return the value associated with the key, all within a constant amount of time.
   - All of the gritty details of your cache implementation should be **hidden** -- i.e., **encapsulated** -- from the client/user. In addition, you should not make any unnecessary variables, methods, or utility classes visible (public) to the client.
   - Your cache must not only return the correct value associated with a key, but it must also do so in constant time. **Submissions that do not achieve constant time in the average case for the `get` method will receive at most 50% of the total points for this part of the assignment.**
2. **R2 (15 points)**: Implement a class called `CacheTest` that instantiates one or more `LRUCache` objects and conducts `junit` unit tests them for correctness. Your `CacheTest` must satisfy the following criteria:
   - **All** tests must pass on a correctly implemented `LRUCache` implementation.
   - **At least 1** unit test must fail on an incorrectly implemented `LRUCache` implementation.

Teamwork

You may work as a team on this project; the maximum team size is 2.

Style

Your code must adhere to reasonable Java style. In particular, please adhere to the following guidelines:

- Each class name should be a singular noun that can be easily pluralized.
- Class names should be in `CamelCase`; variables should be in `mixedCase`.
- Avoid "magic numbers" in your code (e.g., `for (int i = 0; i < 999 /*magic number*/; i++)`). Instead, use **constants**, e.g., `private static final int NUM_ELEPHANTS_IN_THE_ROOM = 999;`, defined at the top of your class file.
- Use whitespace consistently.
- No method should exceed 50 lines of code (for a "reasonable" maximum line length, e.g., 100 characters). If your method is larger than that, it's probably a sign it should be decomposed into a few helper methods.
- Use comments to explain non-trivial aspects of code.
- Use a comment to explain what each method does, what parameters it takes, and what it returns.
- Use the `final` keyword whenever possible.
- Use the **most restrictive** access modifiers (e.g., `private`, default, `protected`>, `public`), for both variables and methods, that you can.
- Declare variables using the **weakest type** (e.g., an interface rather than a specific class implementation) you can; ithen instantiate new objects according to the actual class you need. This will help to ensure **maximum flexibility** of your code. For example, instead of
  ```
  final ArrayList<String> list = new ArrayList();
  ```
  use
  ```
  final List<String> list = new ArrayList<String>();.
  ```

Getting started

Download the Project2 starter file.

Unit testing

When testing your cache, it will be necessary to implement a `DataProvider`. We recommend that you endow your provider with some additional methods that will enable you to verify that the cache is working correctly. For instance, if you know that a (key,value) pair should already be stored in the cache, and the client calls `get` on that same key (resulting in a cache hit), then the cache should **not** call the `get` method of your data provider -- the cache should just return the value associated with the key immediately. By implementing a "test" data provider that offers some auxiliary methods (e.g., `getNumFetches`), you can verify the correctness (or incorrectness) of the `LRUCache` you are testing.

What to Submit

Create a Zip file containing `LRUCache.java` and `CacheTest.java`. Submit the Zip file you created to Canvas. **Submission deadline**: Wednesday, November 7, at 11:59pm EDT.