Data Structures	**Programming Project 1**
CSC242 Fall 2025 (Ali Azhari)	**Due Date: Sunday Sept 28, 2025**

Please solve the problem(s) alone. Remember to test your solution thoroughly. Code that does not work correctly will lose credit. *Code that does not compile will receive 0 points of credit.* Please provide a digital printout of your source code and answers to any accompanying questions by the due date. Please also submit a digital copy, **before** the start of class, via Moodle. The digital copy of your code should be a zip file of the project folder named with your last name followed by the project number. For example, *azhari1.zip* would be my zip file for project 1. Good luck!

A browser (e.g., Firefox, Chrome, Safari, Opera, etc.) fetches a webpage by resolving the URL to an IP address, connecting to the webserver at that IP address, and issuing a GET request. Since a majority of this work happens over the internet (a comparatively slow medium), a cache (a fixed size data structure) is used to speed up the process. There are two caches in used in your browser a *DNS Cache* for speeding up the resolution of URLs to IP addresses and a webpage cache for speeding up GET requests. In this project, we will focus on the DNS cache.

The operation of a cache data structure is straightforward. Before performing a potential URL resolution, you check the cache to see if you already know the IP address, if you do (called a *cache hit*) you simply return the IP address and forgo the expensive request (which must be sent out over the internet). In the case that the URL resolution is not in the cache, called a *cache miss*, the DNS server must be consulted; the resulting resolution information is then added to the cache. Since the cache is of fixed size, you can't just add without bound. Therefore, if the cache is full when trying to add a new resolution, the oldest entry in the cache is removed and the space is used to store the new entry. This is called the *Least Recently Used* (LRU) replacement policy. More details will be provided in the description of Phase I.

In this project you will simulate the operation of LRU DNS cache by implementing an array based cache and determining the cache hits, cache misses, hit rate, and miss rate given a transcript of DNS queries.

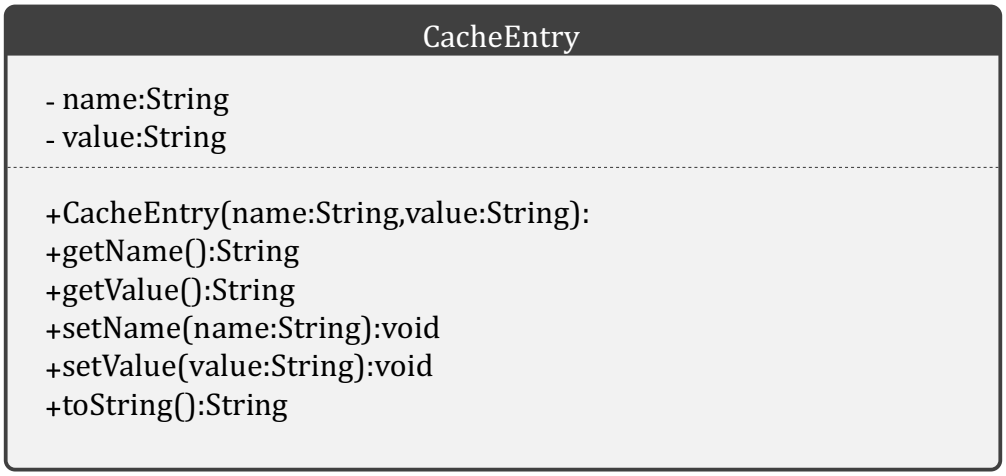
Phase I: The Cache Data Structure

Recommend Completion Date: Sunday, Sept. 28

In this part of the assignment, you will implement a cache that uses a fixed-size array of CacheEntries. Since our cache is a fixed-size data structure, we must implement a strategy to

replace old entries with new entries. The most common cache replacement strategy (used when the cache is full) is *Least Recently Used*(LRU). When using the LRU strategy, if we try to add an entry to a full cache, we remove the *oldest* entry and use the recovered space for the new entry (see the put method of ArrayCache).

The cache data structure (which should live in its own package, called cache) consists of two classes a CacheEntry class and an ArrayCache class. The UML for the CacheEntry class is

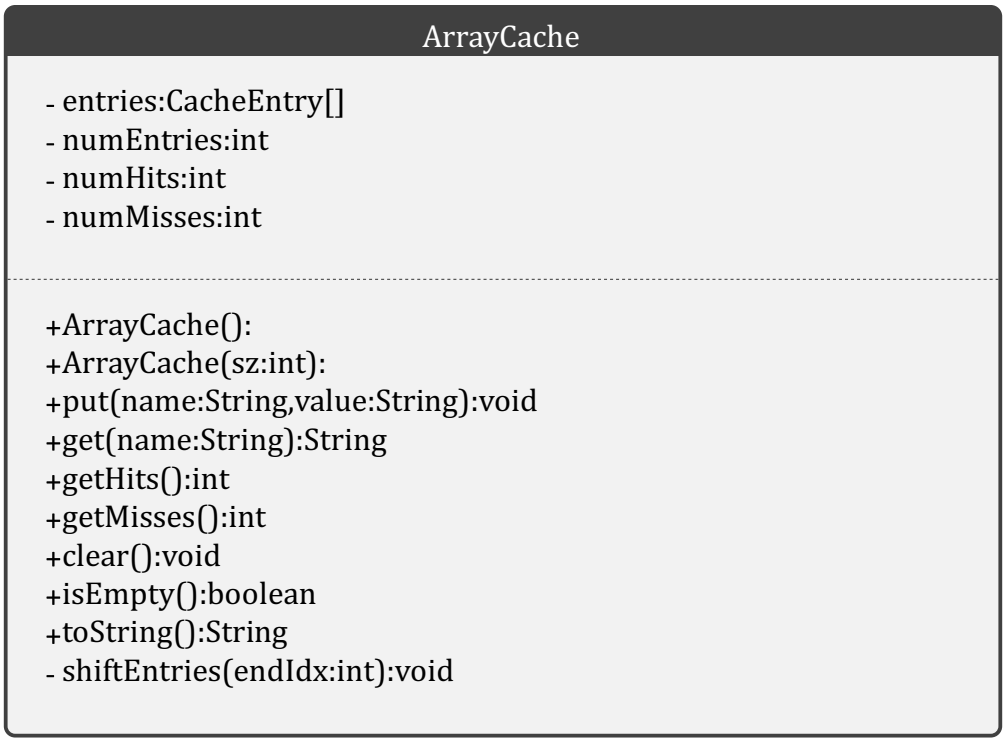


- The constructor takes the name which is a URL and the value which is an IP address and sets attributes appropriately.
- The accessor methods work in the expected way.
- The toString method should return a String that contains the name and value clearly labeled. As a specific example, I would expect the following String to be returned

Name: google.com Value: 142.250.80.110

when name = "google.com" and value = "142.250.80.110".

The UML class diagram for ArrayCache is:



- The default constructor sets entries to an array of ten (10) CacheEntrys and numEntries, numHits, and numMisses to zero (0).
- The overloaded constructor sets entries to an array of szCacheEntrys and numEntries, numHits, and numMisses to zero (0).
- put takes a name and value (URL and IP address, respectively) and adds it to the cache. To add an entry to the cache you should shift all cells in the array one cell to the right (towards higher indices) and put the new entry at index zero (0). If the array is full shifting will throw away the entry at index entries.length - 1 (i.e., overwrite it by shifting). If the number of entries is less than entries.length, increment numEntries. By handling put this way, the LRU element is always at index entries.length - 1.
- get takes a name (i.e., URL) and searches entries for a CacheEntry with the given name. If found at index idx, the CacheEntry should be saved, all entries less than idx moved right (towards higher indices), the saved entry placed at index zero(0), the numHits is incremented, and the entry's value returned to the caller. If no entry is found, the numMisses counter should be incremented and null should be returned.
- getHits and getMisses should do the natural thing.

- clear sets numEntries, numHits, and numMisses to zero.
- isEmpty returns true if the cache is empty; otherwise, false is returned.
- toString returns a String representing the caches data with one(1) CacheEntry displayed per line. For example,

```
Entries: 3
    Hits: 0
  Misses: 0
Name: google.com Value: 142.250.80.110
Name: merrimack.edu Value: 35.169.127.66
Name: xkcd.com Value: 151.101.0.67
```

- shiftEntries takes an index endIdx and shifts all entries from index zero (0) to endIdx - 1 one cell right (towards a higher index).

If you desire you may add *private methods* to help you with writing the code. You may *not* however add or subtract from the public interface or remove private methods I have asked you to write.

Phase II: The Simulator

Recommend Completion Date: Sunday, Sept. 28

The simulator's job is to use the ArrayCache to simulate the executing of the browser's DNS cache. At the end of the simulation the user should be presented with the *hit rate* and *miss rate*. The hit rate is the ratio of the number of hits to the number of cache requests (number hits and misses). We similarly define the miss rate.

To successfully create the simulator, you must write several methods:

- simulate this takes the ArrayCache and name of the query file as arguments. It then carries out the simulation of the DNS cache using the data found in the query file. To simulate the cache, the program should read each line from the file and try to get the URL from the cache. If the cache returns null, put the URL and IP address into the cache. As a note, each line of the query file contains the URL followed by a colon(:) and then the IP address. You will find the split method of the String class helpful in parsing the lines of the file. You should print the cache after each line of the query file is processed (see example output on Classroom).

- printStats this method takes an ArrayCache as an argument and prints to the screen the hit and miss ratio in the format shown below.

  Hit Rate: 26.67% Miss
  Rate: 73.33%

  Hint: You will want to use printf to achieve the nice formatting.

- main this method is responsible for prompting the user for the query file and the the size of the cache (must be a positive number, be sure to ask again if not a given a valid size), then it creates the ArrayCache and calls the simulate method followed by the printStats method.

General Reminders

- Implement the classes *exactly* as described in the UML class diagrams.
- If you desire, you may add *private methods* to help you with writing the code. You may *not* however add or subtract from any public interface or remove private methods I've asked you to write.
- **Start early**.
- You should check your simulations behavior on the query files against my examples.
- Make sure to check your code against the rubric *before* submission.
- You are encouraged to see me or the tutors for help with the project.

Submission

Submissions in this class come in two parts:

1. **Digital Submission**: Since our projects now have multiple files that need to be graded, I ask that you submit a zip file containing the project folder associated with the assignment. This zip file **must be** named with your last name followed by the project number. Failure to follow these directions will result in a deduction of up to **5 points from your grade**.
2. **Hard Copy of Code**: No hard copy is required.

The directions above are for the purpose of ensuring a timely and correct grading of your project.

Grading

Your grade on this assignment will be determined as follows:

- Program follows course style guidelines and has good comments (*10 points*).
- Program passes tests (*10 points*).
- Correctly implemented CacheEntry's accessors and constructor (*5 points*).
- Correctly implemented CacheEntry's toString (*5 points*).
- Correctly implemented ArrayCache's accessors and constructors (*5 points*).
- Correctly implemented ArrayCache's toString (*5 points*).
- Correctly implemented ArrayCache's put (*10 points*). • Correctly implemented ArrayCache's get (*10 points*).
- Correctly implemented ArrayCache's clear and isEmpty (*5 points*).
- Correctly implemented ArrayCache's shiftEntries (*10 points*).
- Correctly implemented simulate (*10 points*).
- Correctly implemented printStats (*5 points*).
- Correctly implemented main (*10 points*).