PAL Worksheet Maps CSC20

Topics: Map Interface
Ref: CSC20 Sets
CSC20 PAL Polymorphism and Interfaces

Map

Imagine you built a personalized phonebook using a linear list, where each element stores the person's name and phone number. However, you'd run into two issues: searching for a person by name requires going over multiple existing elements until the search item is found. Second, it's possible to accidentally enter a person more than once, which would be confusing.

Another way to organize the phonebook would be to separate a person's data into two parts: their name, and their number. If we do this, we could rely on a data structure which instantly returns the number based on the name – an instant lookup is the most efficient search.

The Map collection separates data into two parts: a key and value. This is called a *key-value pair*. We can think of a map as a two-column table with the first column holding the key and the second holding the value. The value of a map can typically be any object data type, even an array or collection. This is important because the value in our phonebook can then store not only the phone number, but also the person's address, email, etc. A key is a unique identifier and duplication of keys is not allowed. The same value can be associated with different keys much like a many-one function.

The Map Interface of Java has two implementations: *HashMap* and *TreeMap*. A HashMap is based on a *hash table*, which uses a mathematical function to take in a key and compute the storage location of the associated value (it doesn't matter if the key is a number or a string). We will stay with HashMaps in this course.

Let's create a simple contact list and put in some contacts, using the Hashmap collection:

```
Map<String, String> contactList = new HashMap<>();

contactList.put('Jane', '123-456-7891');
contactList.put('Chris', '987-654-4321');
```

We can also use an arraylist to store the contact info.

```
Map<String, ArrayList<String>> phoneBook = new HashMap<>();

// build arraylist with jane's relevant info
ArrayList<String> jane_info = new ArrayList<>();
jane_info.add('number: 123-456-7891');
jane_info.add('address: 123 Lombard St');
jane_info.add('email: jane@gmail.com');

// place arraylist as the value for this key
phoneBook.put('Jane', jane_info);
```

Let's say we have a Hashmap-based phonebook with multiple key-value pairs. We can now do a fast lookup by simply providing the key, i.e. 'Chris', and by computing a function on the key, the Hashmap will instantly return Chris's value. There is no need to sequentially go through the list until the key 'Chris' is found.

For the same reason, removing key-value pairs is also a fast operation, via 'remove(key)'.

Ex.1) Write a Java method which inputs a string and then uses a HashMap to count the frequency of each unique character in the string. For example, it takes in "A cat chased some clever mice" and records 4 occurrences of the letter 'c'. It returns the HashMap that was created to contain letter frequencies.

Ex.2) The price of commodities is constantly changing, and a linked list or array structure takes too long to update the price of current commodities in an investment portfolio.

Write a method which takes in an existing HashMap commodity portfolio as a parameter, then continually reads in user input from the keyboard in the format ("gold 117.85") to update its prices. Assume the only keys in the provided hashmap are: ['gold', 'silver', 'oil', 'water', 'palladium, and 'wheat']. The return type of this method is void.

Use the following code to help you take in user input:

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);

while(true){ // loop takes input as: "itemA itemB"

     String itemA = scanner.next(); // take in itemA

     if(commodity.equals("exit")){
          break;
     }
     else{
          String itemB = scanner.nextDouble(); //price
     }
}
```

Let's practice removing items from a HashMap. Its important to know the best practice for solving problems like the ones ahead.

For example, consider the following problem:

At the end of each fiscal year, a manager must remove all employees from the company database, who have scored below a certain level on their annual performance review. Performance scores are awarded out of 100, and a score below 20 means the employee must be removed. The employee database is a HashMap with String employee names as the key and Integer scores as the corresponding value.

Ex.3) Write a Java method called removeKeyByValue(), which takes in a HashMap and a criteria value, then removes all key-value pairs for all values which are less than the criteria value. Use keySet() to get a Set of all keys in the hashmap. It stores these key-values in a new hashmap and returns this map.

Ex.4) Do Practice It 11.15 (max occurrances) and 11.18 (reverse). Use the fact that you can store values in a Hashset using map.values().

Now that we've gone over common Map operations, let's try to understand how Maps compare to other Collections for certain applications.

Ex.5) A human resources system wants to implement a feature that allows quick lookups of employee salary based on their name. They wish to store employee information as String name and Double salary in a Collection

database and want to compare two different Collections: an **ArrayList**, and a **HashMap**.

However, ArrayLists in Java are designed to store elements of a single specified data type. How would you design the ArrayList to store employee information?

Would it be easier to store the information in a HashMap? Which Collection which have the fastest lookup of salary, based on employee name, and why?