Structured Data in Java HashMaps

Arrays and ArrayLists

- Arrays and ArrayLists:
- Are very convenient ways to store data
- Data elements can be directly accessed by their index
- Another way to view an array or ArrayList:
  - You give it an integer key (or index) and it returns the associated value
- This is fast and efficient, but not necessarily applicable to all problems

Mapping between sets

- Sometimes we want to create a mapping between elements of one set and another set
  - Example 1: map people to their phone numbers
    - "Coursera" --> "386-5525"
    - "Jenny" --> "867-5309"
  - Example 2: map state abbreviations to state names
    - "TN" --> "Tennessee"
    - "WA" --> "Washington"
  - Example 3: map student ids to a student records
    - 908264739 --> {name:"John Doe", class:Senior, gpa:3.37,…}
    - 724995613 --> {name:"Sally Jane", class:Junior, gpa:3.85,…}

The **`HashMap`** Class

- The **`HashMap`** class is one of the most valuable tools in the Java Collections Framework and comes up in a surprising number of applications
  - A part of the **`java.util`** package, along with the other Collections members
- The **`HashMap`** implements the `Map` interface**,** which is an associative relationship between keys and values
- A ***key*** is an object that never appears more than once in a map and can therefore be used to uniquely identify a ***value***, which is the object associated with a particular key

Keys and Values

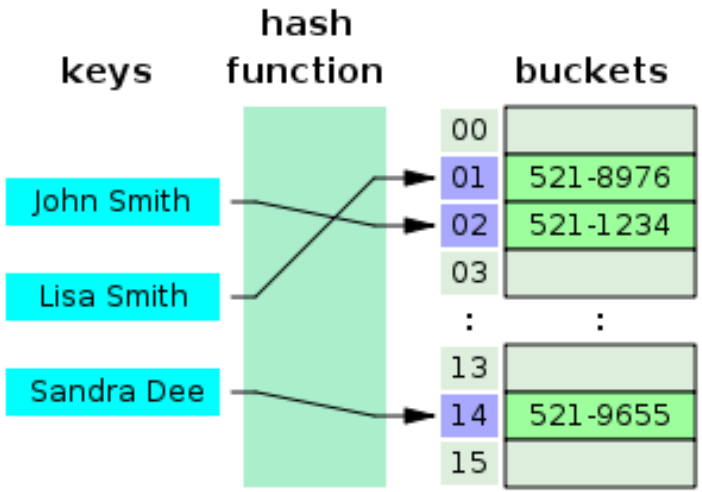
- Dictionary Analogy:
  - The ***key*** in a dictionary is a word:
    android
  - The associated ***value*** is the definition:
    an open-source operating system used for smartphones and tablet computers
- A key and its associated value form a pair that is stored in a map
- To retrieve a value, the key for that value must be supplied
  - An array can be viewed as a Map with integer keys

The `HashMap` Class

- **`HashMap`** are also called:
  - hash tables or hashes
  - dictionaries
  - associative arrays

`HashMap` Properties

- Employs a function to distribute elements evenly over a table



- Need to handle collisions when two elements map to the same table entry

Advantages of **`HashMap`**

- **`HashMaps`** provide us with the following abilities:
  - Ability to use any object type as keys, whereas arrays and **`ArrayLists`** only allow integers as keys
  - Fast insert: O(1) constant time
  - Fast lookup: O(1) constant time
  - Fast removal: O(1) constant time

Disadvantages of `HashMap`

- To get fast insert and lookup, we have to give up some desirable features
  - Values cannot be accessed via an index; must only use a key
  - `HashMaps` are unordered
    - Iteration over the key set will not process them in sorted order as for many other containers
  - Iteration over collection views will take time proportional to the "capacity" of the `HashMap` (the number of buckets) plus its size (the number of key-value mappings)
    - Many other containers have runtimes proportional to their size

Adding to a `HashMap`

- The `put` method is used to place a key-value pair in the `HashMap`
  - You specify both a key and a value
  - This creates an association between the key and the value
- If the key is a new one, the association is added to the `HashMap`
- If the key previously existed, its old value is replaced by the new value
  - The old value is returned as the result of the method

Retrieving from a **`HashMap`**

- The **`get`** method is used to retrieve a value given a key
  - You specify a key and its associated value is returned
- This is similar to getting a value from an **`ArrayList`**, except that we use a key instead of an index
- If the **`HashMap`** does not contain a mapping for the specified key, the value `null` is returned

Deleting from a **`HashMap`**

- The **`remove`** method is used to delete a mapping in an **`HashMap`**
- You specify a key whose mapping is to be deleted
- The method returns the value that had been associated with the key
  - The value `null` is returned if there was no mapping for the key

`HashMap` methods

`put(`key, value`)`	Sets the association for *key* in the map to *value*
`get(`key`)`	Returns the *value* associated with *key*, or *null* if none
`remove(`key`)`	Removes the mapping for *key* if present
`size()`	Returns the number of *key-value* mappings in the map
`isEmpty()`	Returns true if the map is empty, otherwise false
`toString()`	Returns a string representation of the map such as `"{key1=val1, key2=val2,...}"`

`HashMap` methods

`equals(`map`)`	Returns true if the given other *map* contains the same mappings
`clear()`	Removes all mappings from the map
`containsKey(`key`)`	Returns true if this map contains a mapping for the specified *key*
`containsValue(`value`)`	Returns true if this map maps one or more keys to the specified *value*

Plus more…

See the reference documents on the web for a complete list

Collection views

- A **`HashMap`** itself is not regarded as a collection
  - **`HashMap`** does not implement the `Collection` interface
- As such, a **`HashMap`** does not provide for an iterator
- Instead collection *views* of a **`HashMap`** may be obtained
  - Get a *Set* of its keys
  - Get a *Collection* of its values (why not a set?)

Iterators and Maps

- A **`HashMap`** has two methods:
  1. `keySet()`

     Returns a *set view* of the keys contained in the map
  2. `values()`

     Returns a *collection view* of the values contained in the map

- And you can iterate over these collections

- A view is *dynamic access* into the **`HashMap`**
  - If you change the **`HashMap`** , the view changes
  - If you change the view, the **`HashMap`** changes

Examining all elements

- To process all elements of a **`HashMap`**, one usually gets the set view of keys, and then iterates over that set

```
// given a HashMap<String, Integer> named myMap
Set<String> keys = myMap.keySet();
Iterator<String> itr = keys.iterator();
while (itr.hasNext()) {
    String key = itr.next();
    int value = myMap.get(key);
    Out.println(key + " => " + value);
}
```

Choosing the datatype

- When you declare a **`HashMap`** object, you *should* use the **`Map`** interface as the datatype instead of **`HashMap`**
  - That will allow you to change to a **`TreeMap`** implementation in the future, if needed, by changing a single line of code!

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

```
HashMap<String,Integer> myMap =
                    new HashMap<String,Integer>();
```

- The latter is not necessary wrong, just not recommended

`HashMap` Requirements

- The keys and values must be class type
  - Must use wrapper classes for primitive types
- The class type you use for the keys must have the methods `equals` and `hashCode` properly defined
  - This is not a problem if you use `Strings` for the keys (this is *extremely* common)
- You should use *immutable objects* (like `Strings`) as keys
- If you put a value into a `HashMap` with a mutable key, and you later change the key, what happens?
  - Answer: *Nothing good!*

Sample `HashMap` Problem

- To count things, we can make an association between the things to be counted and an integer count
  - We'll use the wrapper class Integer rather than primitive int
- For example, to count the occurrences of words in a file, we will map type String to type Integer
  - The String is the word we are counting
  - The integer represents the number of occurrences
- Since file processing on Android is challenging, we'll assume we are given an ArrayList of the words in the file

Sample **HashMap** Problem

Determine the frequency of words in an array

```
// Given ArrayList<String> words, an array list
// of words to be counted
Map<String, Integer> counts =
       new HashMap<String, Integer>();
for (String word : words){
  word = word.toLowerCase();
  if(!counts.containsKey(word)) {
    counts.put(word, 1);
  } else {
    counts.put(word, counts.get(word)+1);
  }
}
```