Recitation 2 : Maps and Sets

CIT 594

Attendance

https://qr-codes.io/ZYEoEY

Reminders

- HW1 due 2/7 @11:59PM ET
- HW2 was released. Due on 2/14 @11:59PM ET
- Recitation Assignment this week due 2/3 @ 11:45 PM ET
  - Should be able to finish it in this recitation period!
- Java Review Session
  - Recordings and slides were posted

Maps

The Map Interface

- A **map** is an object that maps keys to values.
- A map cannot contain duplicate keys
- Each key can only map to at most one value
- Subinterfaces and implementations:
  - **SortedMap**
    - TreeMap
  - **HashMap**

Map implementations

- **Map** is an interface; you can't say **new Map ( )**
- Here are two implementations:
  - **HashMap** is the faster
  - **TreeMap** guarantees the order of iteration
- It's poor style to expose the implementation, so:
- Good: **Map map = new HashMap ( );**
- Bad: **HashMap map = new HashMap ( );**

The Map Interface: Operations

- **V put(K key, V value):** Associates the specified value with the specified key in this map
- **V get(Object key):** Returns the value to which the specified key is mapped, or null
- **boolean containsKey(Object key):** Returns true if this map contains a mapping for the specified key
- **V remove(Object key):** Removes the mapping for the specified key from this map if present
- **boolean remove(Object key, Object value):** Removes the entry for the specified key only if it is currently mapped to the specified value

Operations relying on comparing elements using the **equals()** or **hashCode()** methods take an object as parameter

More about **put**

- If the map already contains a given key, **put(key, value)** replaces the value associated with that key
- This means Java has to do equality testing on keys
- With a **HashMap** implementation, you need to define **equals** and **hashCode** for all your keys
- With a **TreeMap** implementation, you need to define **equals** and implement the **Comparable** interface for all your keys

Map: Bulk operations

- **void putAll(Map t);**
  - copies one Map into another
- **void clear();**

Map: Collection views

- public Set keySet( );
- public Collection values( );
- public Set entrySet( );
  - returns a set of Map.Entry (key-value) pairs
- You can create iterators for the key set, the value set, or the entry set
- The above views provide the only way to iterate over a Map

Map.Entry: Interface for entrySet elements

```
public interface Entry {
    Object getKey( );
    Object getValue( );
    Object setValue(Object value);
}
```

- This is a small interface for working with the Collection returned by **entrySet( )**
- Can get elements *only* from the **Iterator**, and they are only valid during the iteration

Sets

The Set Interface

- A set is unordered and has no duplicates
- Operations are exactly those for Collections
  - i.e.- size(), isEmpty(), contains(), add(), remove(), iterator(), containsAll(), addAll(), removeAll(), retainAll(), clear(), toArray()

Iterators for sets

- A set has a method **Iterator iterator()** to create an iterator over the set
- The iterator has the usual methods:
  - Boolean hasNext()
  - Object next()
  - void remove()
- **remove()** allows you to remove elements as you iterate over the set
- If you change the set in any other way during iteration, the iterator will throw a ConcurrentModificationException

Set implementations

- Set is an interface; you can’t say **new Set()**
- There are two implementations:
  - **HashSet** is best for most purposes
  - **TreeSet** guarantees the order of iteration
- It’s poor style to expose the implementation, so:
- Good: **Set s = new HashSet( );**
- Bad: **HashSet s = new HashSet( );**

Typical set operations

- Testing if **s2** is a *subset* of **s1**

  s1.containsAll(s2)

- Setting **s1** to the *union* of **s1** and **s2**

  s1.addAll(s2)

- Setting **s1** to the *intersection* of **s1** and **s2**

  s1.retainAll(s2)

- Setting **s1** to the *set difference* of **s1** and **s2**

  s1.removeAll(s2)

Membership testing in HashSets

- When testing whether a **HashSet** contains a given object, Java does this:
    - Java computes the hash code for the given object
        - We’ll discuss **hash codes** later
        - Java compares the given object, using **equals**, only with elements in the set that have the same hash code
- Hence, an object will be considered to be in the set only if both:
    - It has the same hash code as an element in the set, and
    - The equals comparison returns true
- Moral: to use a **HashSet** properly, you must have a good **public int hashCode()** defined for the elements of the set

The SortedSet interface

- A **SortedSet** is just like a **Set**, except that an Iterator will go through it in a guaranteed order
- Implemented by **TreeSet**

Membership testing in TreeSets

- In a **TreeSet**, elements are kept in order
- That means Java must have some means of comparing elements to decide which is “larger” and which is “smaller”
- Java does this by using the **int compareTo(Object)** method of the **Comparable** interface
- For this to work properly, **compareTo** must be consistent with equals
- Moral: to use a **TreeSet** properly, you must implement both the **equals** method and the **Comparable** interface for the elements of the set

Set tips

- **add** and **remove** return **true** if they modify the set
- Here's a trick to remove duplicates from a Collection **c**:
  - **Collection noDups = new HashSet(c);**
- A Set may not contain itself as an element
- **Danger**: the behavior of a set is undefined if you change an element to be equal to another element

Recitation Coding Assignment

Problem 1:

Given an array of integers, return another array of integers containing all duplicate integers removed.

{1, 2, 3, 4, 4, 5} → {1, 2, 3, 4, 5}

Problem 2:

Given an array of Strings where there might be many null values, return a map that contains an entry for each index with a non-null String, mapping the index in the original array to the String.

This is a common technique for saving space when the array is mostly empty.

{"Voravich", "Mia", null, "Norris", null, "Harry"} →
{0=Voravich, 1=Mia, 3=Norris, 5=Harry}

Attendance

https://qr-codes.io/ZYEoEY