Set and Map

Introduction to the Set and Map
data structures in Java

Set - a collection of unique elements

- Set is an interface in java.util
- Elements of a set are unique - no duplicates allowed
- Three implementations:
  - HashSet (unordered)
  - TreeSet (ordered by value “sorted”)
  - LinkedHashSet (insert order)

Set - when to use

- There cannot be any duplicates
  - A deck of cards
  - You want to count unique occurrences
    - hashCode() and equals() must be correct for sets to work!
  - You collect objects but want to ignore duplicates
- Use TreeSet<T> when you want elements to be “sorted”
- Use LinkedHashSet<T> when you want to retain insert order

Set - some set operations

```
Set<SomeType> s1 = ...
Set<SomeType> s2 = ...
boolean isSuperSet = s1.containsAll(s2); // s2 is a subset of s1

s1.addAll(s2); // the union of s1 and s2 - all elements in both, uniquely

s1.retainAll(s2); // intersection - only the elements common to both sets

s1.removeAll(s2); // Set difference, keep the elements of s1 NOT also in s2

// Copy constructor - create a new set (e.g. HashSet) from a collection
s1 = new HashSet<SomeType>(s2); // s2 can be any collection removing dupes
```

Set - creating sets - from a stream

```
public static void main(String[] args) {
  Set<String> uniqeArgs =
    Arrays.asList(args)
    .stream()
    .collect(Collectors.toSet());
  // do something with uniqueArgs;
}
```

Set - creating sets - in a for-each-loop

```
public static void main(String[] args) {
  Set<String> uniqeArgs = new HashSet<>();
  for (String arg : args) {
    uniqueArgs.add(arg);
  }
  // do something with uniqueArgs;
}
```

Set - creating sets - copy constructor

```
public static void main(String[] args) {
  Set<String> uniqueArgs =
      new HashSet<>(Arrays.asList(args));
  // do something with uniqueArgs;
}
```

Further reading

- https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html
- https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
- https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html
- https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Up next: Map!

Map - a dictionary of sorts

- Key-Value pairs
- Keys are a Set! No duplicate keys!
- Sometimes called “associative array”
- Similar to Properties
- Useful for storing settings etc
  - language => English
  - os => GNU/Linux
- One object for a relation (mapping) of keys and values
  - Don’t use two parallel arrays for this for emacs sake
- Is an interface in java.util
  - Not part of the collections hierarchy, but is its ugly cousin

Map - when to use

- You have a set of related values and want to group them in one object
- You want to look up values easilly, using an object (such as String) as the key
- You want lookups to be fast
  - You DON'T want to search an array or a List for an object whose key you already have
- HashMap is a fast implementation
- TreeMap keeps the keys in a sorted set

Map - Some operations

```
Map<String, Integer> studentScore = new HashMap<>();
studentScore.put("Henrik Sandklef", 5);
studentScore.put("Rikard Fröberg", 4);

Integer henrikGrade = studentScore.get("Henrik Sandklef");

// Clear all key-values:
studentScore.clear();

// Transfer all key-values to other map:
someOtherMap.putAll(studentScore); // replacing existing keys

// copyConstructor:
Map<String, Integer> copyMap = new HashMap<>(studentScore);
```

Map - Some operations

```
// Iterate over the keys:
for (String student : studentScore. keySet()) {
  if (studentScore.get(student) > 3 ) {
    System.out.println(student + " has an OK score.");
  } else {
    System.out.println(student + " should study more.");
  }
}
```

Couting word frequencies - “random” order

```
import java.util.*;

public class WordFrequency {
  public static void main(String[] args) {
    Map<String, Integer> frequencies = new HashMap<>();
    for (String arg : args) {
      Integer frequency = frequencies.get(arg);
      if (frequency == null) { // arg didn't exist "
        frequency = 0;
      }
      frequencies.put(arg, frequency + 1);
    }
    System.out.println(frequencies);
  }
}
$ javac WordFrequency.java && java WordFrequency it put the lotion on its skin or it gets
the hose again
{the=2, or=1, again=1, skin=1, its=1, hose=1, lotion=1, it=2, gets=1, put=1, on=1}
```

Couting word frequencies - lexicographical order

```
import java.util.*;

public class WordFrequency {
  public static void main(String[] args) {
    Map<String, Integer> frequencies = new TreeMap<>();
    for (String arg : args) {
      Integer frequency = frequencies.get(arg);
      if (frequency == null) { // arg didn't exist "
        frequency = 0;
      }
      frequencies.put(arg, frequency + 1);
    }
    System.out.println(frequencies);
  }
}
$ javac WordFrequency.java && java WordFrequency it put the lotion on its skin or it gets
the hose again
{again=1, gets=1, hose=1, it=2, its=1, lotion=1, on=1, or=1, put=1, skin=1, the=2}
```

Could you spot the difference?

- Always program againt the interface type Map (same goes for Set, List etc)
- Changing from HashMap to TreeMap made the order of the keys (as shown in the toString()) be sorted

Further reading

- https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
- https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
- https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
- https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html