Sets and Maps

Bruce Nan

The Set interface

- A Set is unordered and has no duplicates
- Operations are exactly those for Collection

```
int size( );
boolean isEmpty( );
boolean contains(Object e);
boolean add(Object e);
boolean remove(Object e);
Iterator iterator( );
```

```
boolean containsAll(Collection c);
boolean addAll(Collection c);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
void clear( );

Object[ ] toArray( );
Object[ ] toArray(Object a[ ]);
```

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()`
- Since sets have iterators, you can also use Java 5's "enhanced `for` loop"
- `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`

Count the number of distinct words

```
import java.io.*;
import java.util.*;
public class test {
   public static void main(String [] args){
   String [] words = {"hello", "world", "hello", "hi", "java",
"hi", "java", "world"};
   Set<String> myset = new HashSet();
   for(int i=0; i<words.length; i++)
   myset.add(words[i]);
   System.out.println("# of distinct words: " + myset.size());
   for(String x: myset)
   System.out.println(x);
   }
}
```

Set implementations

- Set is an interface; you can't say new Set( )
- There are four implementations:
  - HashSet is best for most purposes
  - TreeSet guarantees that an iterator will return elements in sorted order
  - LinkedHashSet guarantees that guarantees that an iterator will return elements in the order they were inserted
  - AbstractSet is a "helper" abstract class for new implementations
- It's poor style to expose the implementation, so:
- Good: `Set s = new HashSet( );`
  Fair: `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)
  ```

Set equality

- Object.equals(Object), inherited by all objects, really is an *identity* comparison
- Implementations of Set override equals so that sets are equal if they contain the same elements
- equals even works if two sets have different implementations
- equals is a test on entire sets; you have to be sure you have a working equals on individual set elements
- hashCode has been extended similarly
  - This is for *sets*, not *elements* of a collection!

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
    - Hash codes are discussed in a separate lecture
    - 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 boolean equals(Object) and 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 ascending order
- SortedSet is 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 either:
  - The int compareTo(Object) method of the Comparable interface, or
  - The int compare(Object, Object) method of the Comparator interface
- Which method to use is determined *when the TreeSet is constructed*

Comparisons for TreeSets

- new TreeSet()
  - Uses the elements "natural order," that is, it uses compareTo(Object) from Comparable
  - All elements added to this TreeSet must implement Comparable, or you will get a ClassCastException
- new TreeSet(Comparator)
  - Uses compare(Object, Object) from the given Comparator
  - The Comparator specified in the constructor must be applicable to all elements added to this TreeSet, or you will get a ClassCastException
- **Moral:** to use a TreeSet properly, you must provide the equals method and implement either Comparable or Comparator for the *elements* of the set

How hard is it to use a Set?

- You must have a working equals(Object) and a working hashCode() or comparison method
- If you don't really care about iteration order, *every* object inherits equals(Object) and hashCode() from Object, and this is usually good enough
  - That is, assuming you are happy with the == test
- Strings do all this for you (they implement equals, hashCode, and Comparable)
- **Bottom line:** If you don't care about order, and == is good enough, just use HashSet

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 an an element
- **Danger:** The behavior of a set is *undefined* if you change an element to be equal to another element
- **Danger:** A TreeSet *may* throw a ConcurrentModificationException if you change an element in the TreeSet

The Map interface

- A Map is an object that maps keys to values
- A map cannot contain duplicate keys
- Each key can map to at most one value
- Examples: dictionary, phone book, etc.

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 unnecessarily, so:
- Good: Map map = new HashMap( );
  Fair: HashMap map = new HashMap( );

Map: Basic operations

```
Object put(Object key, Object value);
Object get(Object key);
Object remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
int size( );
boolean isEmpty( );
```

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
  - Example: newMap.putAll(oldMap);
- void clear();
  - Example: oldMap.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 set of entries, that is, key-value pairs)
- The above views provide the *only* way to iterate over a Map

Map example: Count frequency of word

```
import java.io.*;
import java.util.*;
public class test {
   public static void main(String [] args){
   String [] words = {"hello", "world", "hello", "hi", "java", "hi",
"java", "world", "hello"};
   Map<String, Integer> mymap = new HashMap();
   for(int i=0; i<words.length; i++)
   mymap.put(words[i], 1+mymap.getOrDefault(words[i], 0));
   System.out.println("# of distinct words: " + mymap.size());
   for(String x: mymap.keySet())
   System.out.println(x + " occurs " + mymap.get(x) + " times ");
   }
}
```

Map.Entry
Interface for **entrySet** elements

- public interface Entry { // Inner interface of Map
  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

The End