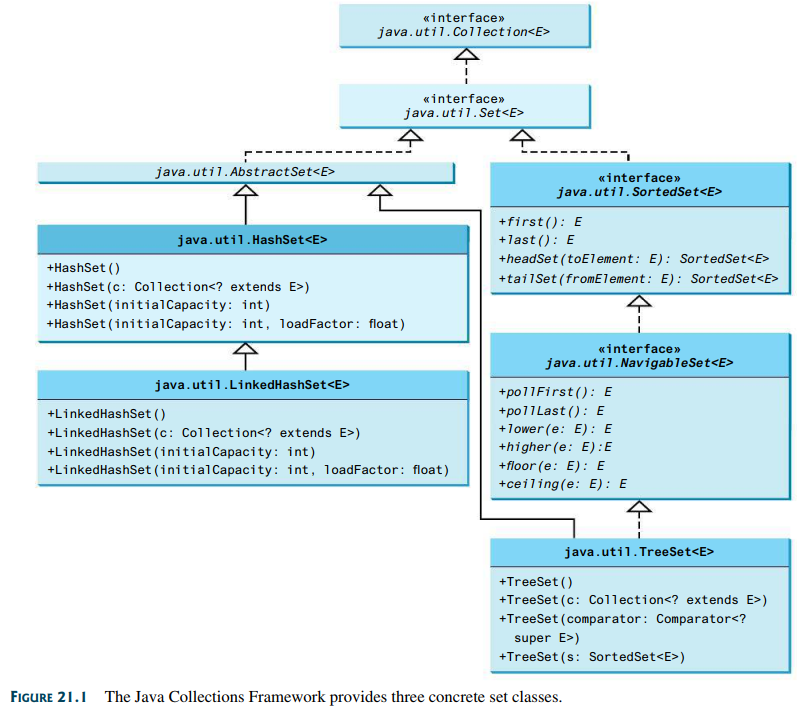
COP 2251 – Java Programming II : Chapter 21 – Sets and Maps

**Sets**

* The Set interface extends the Collection interface.
* A set cannot contain duplicate objects.
* There are 3 concrete classes of Set. See Listing 21.1 on page 817 or below.



1. **HashSet**

* Used for a non-duplicate group of elements stored in no particular order.

Try **TestHashSet.java** from the zipped examples.

Try **HashSetOne.java** and observe the output.

* Note that this example creates a set of Strings (but could be any **reference** type).
* Note the output order differs from the order in which elements were inserted.
* You have no control over the order of a HashSet. The order is determined by a hashing algorithm.
* If you need order, use a LinkedHashSet.
* The elements in a set can be traversed with an iterator. A foreach loop or the forEach method can also be used.

Read “What is a hash code” back in mycourses.

* All Java classes inherit or override a hashcode() method from java.lang.Object.
* The hash code of an object points at its address in memory.
* The hash code of a Java collection is equal to the sum of the hash codes of its elements.

**Exercise**:

Add code to TestHashSet.java to verify the last statement above.

Try **TestMethodsInCollection.java**.

* Note how sets can be combined and manipulated with methods inherited from **Collection**.

1. **LinkedHashSet**

* Use for a non-duplicate set retrievable in the same order in which it is created.
* To impose a different order, use TreeSet.
* Figure 21.1 on page 817 shows that this class extends class **HashSet**.

Try **TestLinkedHashSet.java**.

* Note that the cities are displayed in the order in which they were added to the set.

Try **LinkedHashSetOne.java**.

* Note that the latter example also illustrates **autoboxing** of ints to Integers.

1. **TreeSet**.

* For a non-duplicate set that is sorted.
* The elements of a TreeSet must be capable of being compared to each other for sorting.
* That means the elements must implement the Comparable interface or the Comparator interface.
* TreeSet extends **NavigableSet** which extends the **SortedSet** interface. Refer again to Figure 21.1. These super classes provide a number of methods for manipulating a TreeSet. Review these methods in the UML diagrams of Figure 21.1 on page 817.

Try **TestTreeSet.java**.

* Note that this TreeSet is created from a HashSet by specifying the HashSet instance inside the parentheses of the TreeSet constructor.
* SortedSet Methods of Interest
  + first() and last() have obvious purposes.
  + headSet(toElement) returns elements of set **before** toElement argument.
  + tailSet(fromElement) returns elements **equal to** or **after** fromElement argument.
* NavigableSet Methods of Interest
  + ceiling(arg) returns least (smallest) element greater than or equal to arg.
  + floor(arg) returns the highest element less than or equal to arg.
  + higher(arg) returns the least (smallest) element greater than (but not equal to) arg.
  + lower(arg) is the opposite of higher().
  + pollFirst() returns and removes the first element, or returns null if set is empty.
  + pollLast() returns and removes the last element, or returns null if set is empty.

Try **TreeSetOnejava**.

* The latter example creates a **HashSet** of Double objects, and then a **TreeSet** from it.
* Recall that there is no control of order in the HashSet.
* Note the TreeSet is sorted, and resorted after adding each element.
* The TreeSet is “automagically” sorted because Double implements the **Comparable** interface.
* The **compareTo( )** method in this interface sorts for us in “natural order”.

Issue: what if you are using **custom objects** of your own design that don’t implement Comparable, or what if you want to sort in another manner?

Solution: You have to create your own “comparator” that implements the **Comparator** interface. We do this next.

**The Comparator Interface**

* Review the Comparator interface on page 787 in Chapter 20.
* Recall that the **compare( )** method must be capable of returning a negative, a positive, and zero.

Study **GeometricObject.java** and **GeometricObjectComparator.java** on page 787.

Try **TestTreeSetWithComparator.java**.

* Note that the objects are sorted by area using the compare() method.

Another Example

* Assume that you are making a retail inventory application. Firstly, create a custom class named **InventoryValue.java.** It’s in the project.
* Assume that you want to sort by the retail value of the inventory items, and you obtain the retail value with the double returned by the **getValue( )** method.
* Next, we need a class that implements the **Comparator** interface.
* **InventoryComparator.java** is supplied in the project for this purpose.
* Note how value1 and value2 are obtained by running the **getValue( )** method of the two **InventoryValue** objects passed into the **compare( )** method.
* Note also how the if…else blocks will return either -1, 0, or 1 as the compare() method in all comparator classes should do. This method will be used when performing the sort.
* The executable class **TestInventoryComparator.java** file completes this example.

Try **TestInventoryComparator.java**.

**Try also TestInventoryComparator2.java**.

* This example sorts an array of InventoryValue objects by value using Comparator.comparing() and a lambda expression.

Comparing the Performance of Sets and Lists

* Sets are more efficient than lists for determining if a specified element is in the collection.

Try **SetListPerformanceTest.java** to see a demonstration of this.

* Note how a set can be created from an array with the **Arrays.asList()** method.

Case Study: Counting Keywords

* This case study stores Java keywords in a HashSet.

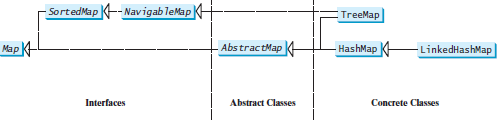
Try **CountKeywords.java**. Enter Welcome.java as the file to be examined. It’s in the project folder.

**The Map Interface**

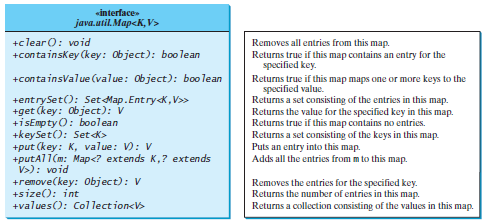
* A Map is a collection that maps keys to object values. It stores **key-value pairs**.
* Think of maps as arrays except that elements are accessed with keys, instead of indexes.

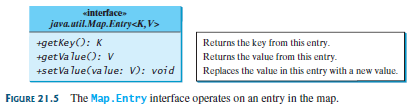
NOTE: in other languages, a map might be called a dictionary, associative array, or hash.

* The key is used to retrieve the object, so duplicate keys are not permitted. Each key can map to only one value. See Figure 21.2 on page 828.
* The UML in Figure 21.3 shows two interfaces, one abstract class, and three concrete classes.



* See the UML diagram in Figure 21.4 on page 829 for the methods of the Map interface.



* Note that two generics are required (one for the key and one for the value) for some methods.
* The three concrete map classes: HashMap, LinkedHashMap, and TreeMap.
* The **keySet()** method of Map can return a set (no duplicate keys) of key objects 
* The forEach() method provides a new and easy way to process a Map.

Try **TestMap.java**.

1. **HashMap**

* Doesn’t support ordering but is good for quickly locating a particular value by its key.
* Also good for inserting and deleting a key-value pair.

Try **HashMapOne.java**. It’s not in the book.

* NOTE the use of the forEach() method with a lambda expression.

1. **LinkedHashMap**

* Supports ordering in several orders, depending on the constructor used.
* Using the no-arg constructor makes a LinkedHashMap in insertion order.
* Other constructors create with access order, based on most recent access.

1. **TreeMap**

* This class stores key=value pairs sorted by keys.
* TreeMap is good for seamlessly processing the keys in a sorted order using **Comparable.**
* You can also sort with an implementation of the **Comparator** interface that you design (as was done earlier with TreeSet).

Try **TreeMapper.java**.

* The cabinet TreeMap is sorted by the keys (positions).
* Usually, you will want to retrieve values by their keys, but you can use an iterator as well.

Try **TestMap.java**.

* This example demonstrates all three concrete Map classes. Note that Lewis, the most recently accessed entry, is placed at the end of the map because variable linkedHashMap was NOT created with the no-arg constructor.

**Keys and Values**

* The keyset() method returns all Map keys as a Set, and the values() method returns the values as a set.

Try **KeysValues.java**. It’s not in the book.

Case Study: Occurrences of Words

* A TreeMap is used to store key=value pairs of words and their counts in a string.

Try **CountOccurrenceOfWords.java**.

* In each pair, the word is the key and the count is the value.

Singleton and Unmodifiable Collections and Maps

* Static methods of the Collections interface can be used to create a singleton set, singleton list or singleton map.
* A singleton class is limited to only one instance. Singletons are useful when it is important to control resources, such as access to a database.
* In programming, a **singleton** is a design pattern that restricts the instantiation of a class to **one object**. A singleton “collection” has only one entry.
* There are also static methods that return read-only views of collections.
* See UML and discussion on page 835.

**Next and final topic – MySQL (or Maria) Databases and Java**