UNIT V – COLLECTIONS FRAMEWORK

Collection overview – Recent changes to collection - Collection interface – Collection classes – Working with maps –Collection algorithms - The legacy classes and interfaces. Applet class: Types – Basics – Architecture – Skeleton – Display methods – repainting – Status window – HTML applet tag – Passing parameter - Creating a swing applet - Painting in swing - A paint example, Exploring swing

---

Working with maps

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot. There is one key point about maps that is important to mention at the outset: they don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can't obtain an iterator to a map. However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the for loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches. (Added by Java SE 6.)
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

The Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key. Map is generic and is declared as shown here:

interface Map<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by Map are summarized in Table.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
V remove(Object *k*)	Removes the entry whose key equals *k*.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Several methods throw a ClassCastException when an object is incompatible with the elements in a map. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map. An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map. An IllegalArgumentException is thrown if an invalid argument is used.

Maps revolve around two basic operations: get( ) and put( ). To put a value into a map, use put( ), specifying the key and the value. To obtain a value, call get( ), passing the key as an argument. The value is returned. As mentioned earlier, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the Collection interface. However, you can obtain a collection-view of a map. To do this, you can use the entrySet( ) method. It returns a Set that contains the elements in the map. To obtain a collection-view of the keys, use keySet( ). To get a collection-view of the values, use values( ). Collection-views are the means by which maps are integrated into the larger Collections Framework.

The SortedMap Interface

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending order based on the keys. SortedMap is generic and is declared as shown here:

interface SortedMap<K, V>

Here, K specifies the type of keys, and V specifies the type of values.
The methods declared by SortedMap are summarized in Table.

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

The NavigableMap Interface

The NavigableMap interface was added by Java SE 6. It extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. NavigableMap is a generic interface that has this declaration:

interface NavigableMap<K,V>

Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys. In addition to the methods that it inherits from SortedMap, NavigableMap adds those summarized in Table.

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.

Several methods throw a ClassCastException when an object is incompatible with the keys in the map. A NullPointerException is thrown if an attempt is made to use a null object and null keys are not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.

The Map.Entry Interface

The Map.Entry interface enables you to work with a map entry. Recall that the entrySet( ) method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object. Map.Entry is generic and is declared like this:

interface Map.Entry<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

Notice that AbstractMap is a superclass for all concrete map implementations. WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused.

The HashMap Class

The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map. This allows the execution time of get( ) and put( ) to remain constant even for large sets. HashMap is a generic class that has this declaration:

class HashMap<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The following constructors are defined:

HashMap( )

HashMap(Map<? extends K, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of m. The third form initializes the capacity of the hash map to capacity. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.

The meaning of capacity and fill ratio is the same as for HashSet, described earlier. The default capacity is 16. The default fill ratio is 0.75.

HashMap implements Map and extends AbstractMap. It does not add any methods of its own. You should note that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates HashMap. It maps names to account balances. Notice how a set-view is obtained and used.

```
import java.util.*;
class HashMapDemo
{
	public static void main(String args[])
	{
		HashMap<String, Double> hm = new HashMap<String, Double>();
		hm.put("John Doe", new Double(3434.34));
		hm.put("Tom Smith", new Double(123.22));
		hm.put("Jane Baker", new Double(1378.00));
		hm.put("Tod Hall", new Double(99.22));
		hm.put("Ralph Smith", new Double(-19.08));
		Set<Map.Entry<String, Double>> set = hm.entrySet();
		for(Map.Entry<String, Double> me : set)
		{
			System.out.print(me.getKey() + ": ");
```

```
                System.out.println(me.getValue());
            }
            System.out.println();
            double balance = hm.get("John Doe");
            hm.put("John Doe", balance + 1000);
            System.out.println("John Doe's new balance: " + hm.get("John Doe"));
        }
}
```

Output from this program is shown here (the precise order may vary):

Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe's new balance: 4434.34

The TreeMap Class

The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

TreeMap is a generic class that has this declaration:

class TreeMap<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The following TreeMap constructors are defined:

TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the Comparator comp. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

TreeMap has no methods beyond those specified by the NavigableMap interface and the AbstractMap class. The following program reworks the preceding example so that it uses TreeMap:

```
import java.util.*;
class TreeMapDemo
{
	public static void main(String args[])
	{
		TreeMap<String, Double> tm = new TreeMap<String, Double>();
		tm.put("John Doe", new Double(3434.34));
		tm.put("Tom Smith", new Double(123.22));
		tm.put("Jane Baker", new Double(1378.00));
		tm.put("Tod Hall", new Double(99.22));
		tm.put("Ralph Smith", new Double(-19.08));
		Set<Map.Entry<String, Double>> set = tm.entrySet();
		for(Map.Entry<String, Double> me : set)
		{
			System.out.print(me.getKey() + ": ");
			System.out.println(me.getValue());
		}
		System.out.println();
		double balance = tm.get("John Doe");
		tm.put("John Doe", balance + 1000);
		System.out.println("John Doe's new balance: " + tm.get("John Doe"));
	}
}
```

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22
John Doe’s current balance: 4434.34
```

The LinkedHashMap Class

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a LinkedHashMap, the

elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed. LinkedHashMap is a generic class that has this declaration:

class LinkedHashMap<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

LinkedHashMap defines the following constructors:

LinkedHashMap( )

LinkedHashMap(Map<? extends K, ? extends V> m)

LinkedHashMap(int capacity)

LinkedHashMap(int capacity, float fillRatio)

LinkedHashMap(int capacity, float fillRatio, boolean Order)

The first form constructs a default LinkedHashMap. The second form initializes the LinkedHashMap with the elements from m. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for HashMap. The default capacity is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If Order is true, then access order is used. If Order is false, then insertion order is used. LinkedHashMap adds only one method to those defined by HashMap. This method is removeEldestEntry( ) and it is shown here:

protected boolean removeEldestEntry(Map.Entry<K, V> e)

This method is called by put( ) and putAll( ). The oldest entry is passed in e. By default, this method returns false and does nothing. However, if you override this method, then you can have the LinkedHashMap remove the oldest entry in the map. To do this, have your override return true. To keep the oldest entry, return false.

The IdentityHashMap Class

IdentityHashMap extends AbstractMap and implements the Map interface. It is similar to HashMap except that it uses reference equality when comparing elements. IdentityHashMap is a generic class that has this declaration:

class IdentityHashMap<K, V>

Here, K specifies the type of key, and V specifies the type of value. The API documentation explicitly states that IdentityHashMap is not for general use.

The EnumMap Class

EnumMap extends AbstractMap and implements Map. It is specifically for use with keys of an enum type. It is a generic class that has this declaration:

class EnumMap<K extends Enum<K>, V>

Here, K specifies the type of key, and V specifies the type of value. Notice that K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.

EnumMap defines the following constructors:

EnumMap(Class<K> kType)

EnumMap(Map<K, ? extends V> m)

EnumMap(EnumMap<K, ? extends V> em)

The first constructor creates an empty EnumMap of type kType. The second creates an EnumMap map that contains the same entries as m. The third creates an EnumMap initialized with the values in em. EnumMap defines no methods of its own.
