Collections

Arathi Bhandari

Overview

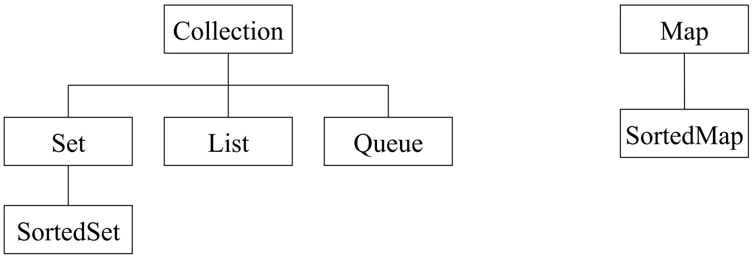
- A Collection is a container that groups similar elements into an entity.
- Examples would include a list of bank accounts, set of students, group of telephone numbers.
- The Collections framework in Java offers a unified approach to store, retrieve and manipulate a group of data.

Benefits

- Helpful to developers by providing all standard data structures algorithms to let the developer concentrate more on functionality rather than the lower level details
- Faster execution: with a range of options available for choice of implementation. The framework aids in improving the quality and performance of Java applications.
- Code to interface: Since the collections framework is coded on the principles of “Code to Interface”, it aids in inter-operability between APIs that are not directly related.
- Learning Curve: Easy to learn and start using Collections due to the “Code to Interface” principles.

Core Collections Framework

- The Collection framework forms a suitable hierarchy:



Collection

Set

List

Queue

SortedSet

Map

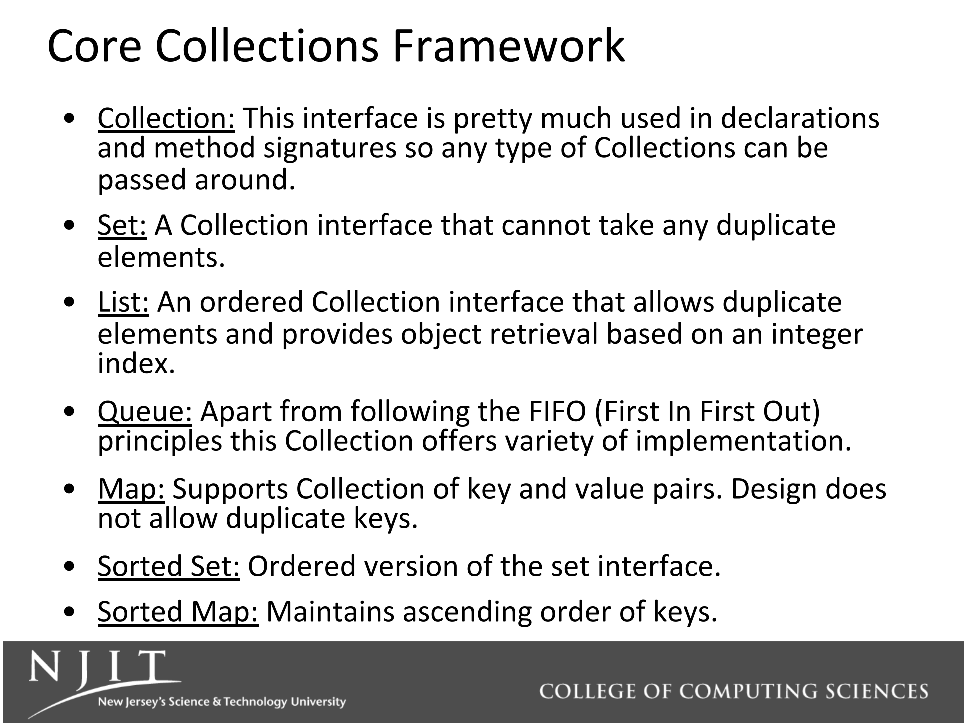
SortedMap

Concrete Collections

concrete collection	implements	description
`HashSet`	`Set`	hash table
`TreeSet`	`SortedSet`	balanced binary tree
`ArrayList`	`List`	resizable-array
`LinkedList`	`List`	linked list
`Vector`	`List`	resizable-array
`HashMap`	`Map`	hash table
`TreeMap`	`SortedMap`	balanced binary tree
`Hashtable`	`Map`	hash table

Core Collections Framework

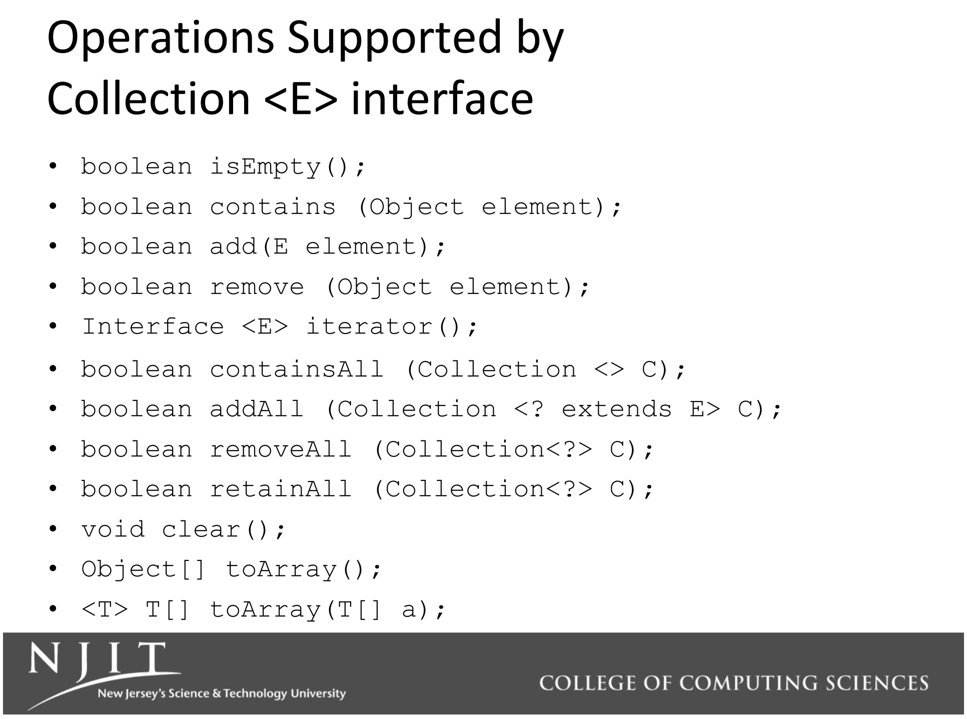
- The core Collections framework is very generic in nature and provides a standard interface to all operations that can be performed.
- For example, the root interface Collection is declared as public interface Collection <E>
  - E represents a generic object.



Core Collections Framework

- Collection: This interface is pretty much used in declarations and method signatures so any type of Collections can be passed around.
- Set: A Collection interface that cannot take any duplicate elements.
- List: An ordered Collection interface that allows duplicate elements and provides object retrieval based on an integer index.
- Queue: Apart from following the FIFO (First In First Out) principles this Collection offers variety of implementation.
- Map: Supports Collection of key and value pairs. Design does not allow duplicate keys.
- Sorted Set: Ordered version of the set interface.
- Sorted Map: Maintains ascending order of keys.

7



Operations Supported by Collection <E> interface

```
• boolean isEmpty();
• boolean contains (Object element);
• boolean add(E element);
• boolean remove (Object element);
• Interface <E> iterator();
• boolean containsAll (Collection <> C);
• boolean addAll (Collection <? extends E> C);
• boolean removeAll (Collection<?> C);
• boolean retainAll (Collection<?> C);
• void clear();
• Object[] toArray();
• <T> T[] toArray(T[] a);
```

8

Two ways to iterate over Collections

Iterator

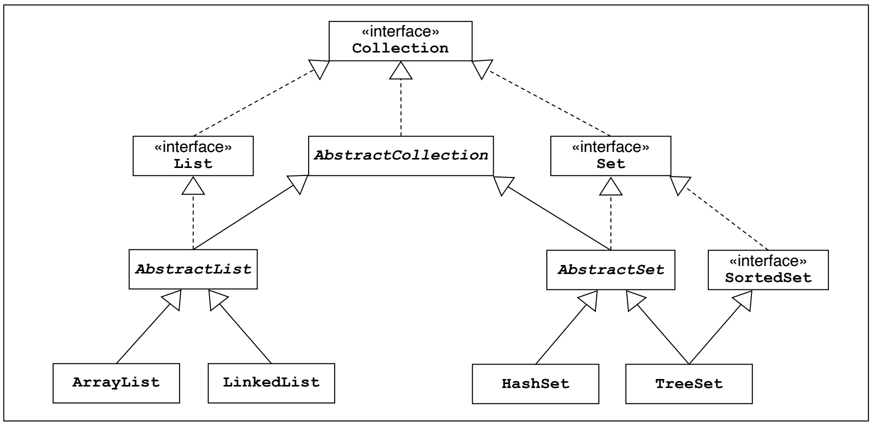
```
static void loopThrough(Collection col){
Iterator <E> iter = col.iterator();
while (iter.hasnext()) {
      Object obj=iter.next();
  }
}
```

For-each

```
static void loopThrough(Collection col){
    for (Object obj: col) {
           //access object
    }
}
```

The `Collection` Hierarchy

The following diagram shows the portion of the Java Collections Framework that implements the **`Collection`** interface. The dotted lines specify that a class implements a particular interface.



AbstractCollection Class

java.util.*AbstractCollection*

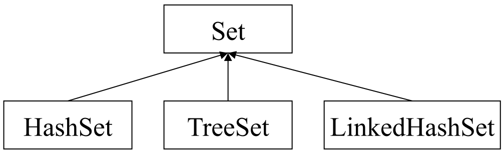
- Abstract class which is partial implementation of of Collection interface

- Implements all methods except iterator() and size()

- Makes it much less work to implement Collections Interface

Set

- Set Interface: It offers inherited services from the root Collection interface with the added functionality of restricting duplicate elements.
- Implementations:



Example Code

- Adding unique objects to a collection

```
Collection <String> uniqueString (Collection<String> c) {
  Set<String> uniqueSetStr = new HashSet<String>();
  for (String str: c) {
      if(!uniqueStrSet.add(str)) {
               System.out.println( “Duplicate deleted: ”+ str);
      }
}
return uniqueStrSet;
```

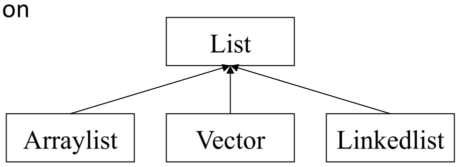
Set Implementation Comparisons

	HashSet	**TreeSet**	**Linked HashSet**
Storage Type	Hash Table	Red-Black Tree	Hash Table with a Linked List
Performance	Best performance	Slower than HashSet	Little costly than HashSet
Order of Iteration	No guarantee of order of iteration	Order based	Orders elements based on insertion

- Idioms for using bulk operations on sets:
- Copy a set when doing bulk operations to retain the original set.
- Example:
  Set <String> unionSet = new TreeSet<String>(set1); unionSet.addAll(set2);

List

- In addition to the core services inherited from the root collection interface, the list interface offers
  - Positional access
  - Search
  - Customized Iteration
  - Range-view
- List Implementation



List

Arraylist

Vector

Linkedlist

  - ArrayList offers better performance compared to Linkedlist.

`ArrayList` *vs* `LinkedList`

- Two classes in the Java Collections Framework that implement the **`List`** interface: **`ArrayList`** and **`LinkedList`**.
- Because these classes implement the same interface, it is generally possible to substitute one for the other.
- The fact that these classes have the same effect, however, does not imply that they have the same performance characteristics.
  - The **`ArrayList`** class is more efficient if you are selecting a particular element or searching for an element in a sorted array.
  - The **`LinkedList`** class can be more efficient if you are adding or removing elements from a large list.
- Choosing which list implementation to use is therefore a matter of evaluating the performance tradeoffs.

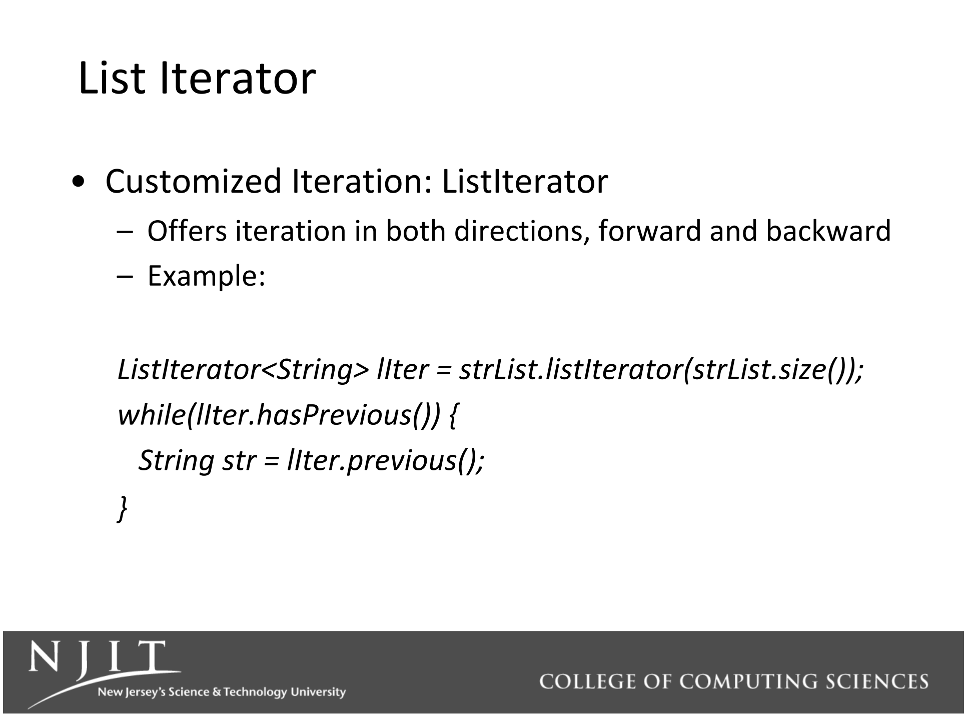
Vector class

- Like an ArrayList, but synchronized for multithreaded programming.
- Mainly for backwards-compatibility with old java.
- Used also as base class for *Stack* implementation.

Typical usage of List

- Bulk Operations
  - *listA.addAll(listB); //* append elements.
- Positional access and search
  - Eg: To swap elements in a list.

```
public static<String> void swap(List<String>strList, int k, int l)
  {
  String strTmp = strList.get(l);
  strList.set(k,strList.get(l));
  strList.set(l,strTmp);
  }
```

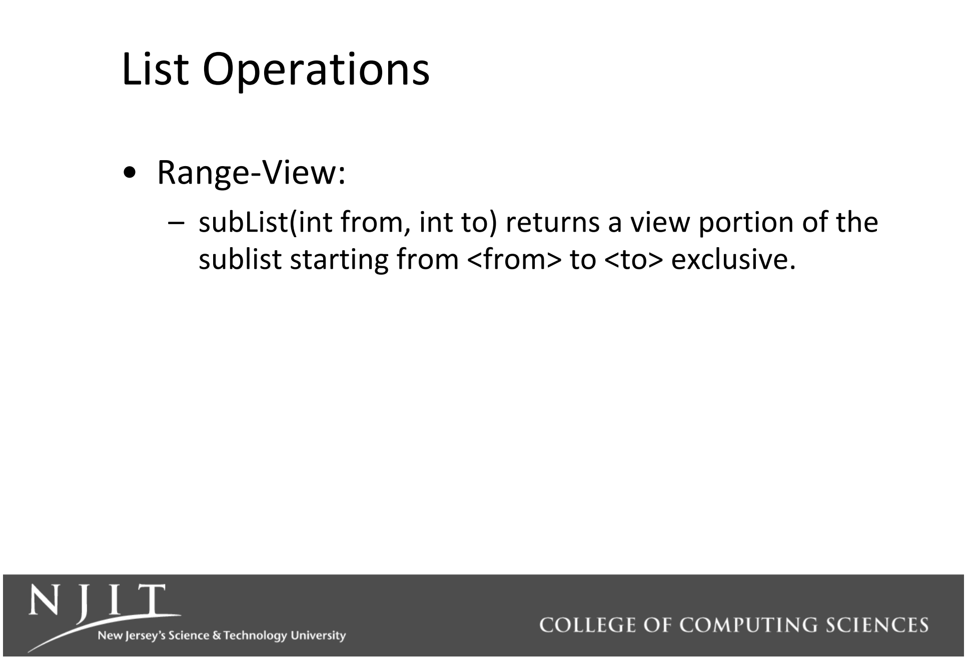


List Iterator

- Customized Iteration: ListIterator
  - Offers iteration in both directions, forward and backward
  - Example:

```
ListIterator<String> lIter = strList.listIterator(strList.size());
while(lIter.hasPrevious()) {
  String str = lIter.previous();
}
```

19



List Operations

- Range-View:
  - subList(int from, int to) returns a view portion of the sublist starting from <from> to <to> exclusive.

20

Algorithms available for List

- sort: Uses mergesort
- shuffle: Randomly shuffles elements
- reverse: Reverses order of elements
- rotate: Rotates list by specified number
- fill: Overwrites every element with specified element.
- copy: Copies source list into destination.
- binarySearch: Performs search using binary search.
- indexOfSubList: Returns index of first subList found.
- lastIndexOfSubList: Returns index of the last subList found.

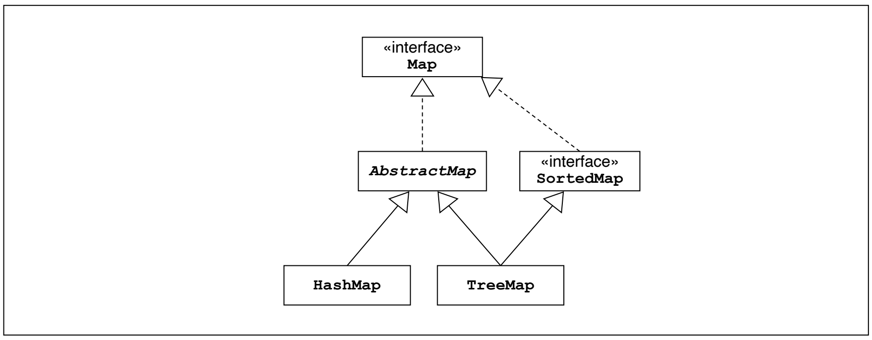
Queue

- In addition to the inherited core services offered by Collection, queue offers following methods in two flavors:

Operatiom	**Purpose**	**Throw Exception**	**Return Special Value**
Insert	Inserts an elements to the queue	add(obj)	offer(obj)
Remove	Remove head of the queue and return it.	remove()	poll()
Examine	Return the head of the queue	element()	peek()

The **Map** Hierarchy

- The following diagram shows the portion of the Java Collections Framework that implements the **Map** interface.
- The structure matches that of the **Set** interface in the **Collection** hierarchy.
- The distinction between **HashMap** and **TreeMap** is the same as that between **HashSet** and **TreeSet**.



Map

- Basic operations:
  - Val put (Object key);
  - Val get (Object key);
  - Val remove (Object key);
  - boolean containsKey (Object key);
  - boolean containsValue (Object value);
  - int size();
  - boolean isEmpty();
- Bulk services
  - void putAll(Map<? extends Key, ? extends Val> map);
  - void clear();

Map

- <u>Views</u>

  public Set<Key> keySet();

  public Collection<Val> values();

  public Set<Maps.Entry<Key,Val>> entrySet();

Map

Iteration over a map

for(Map.Entry<?,vtype> elem: map.entrySet())
System.out.print(element.getKey()+ ”: ”+element.getValue());

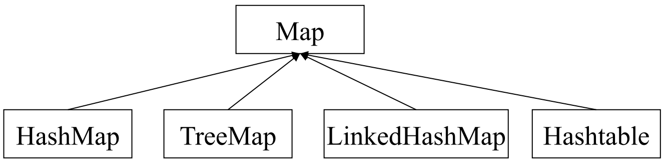
To find if a map contains another map

If(map1.entrySet().containsAll(Map2.entrySet()))

If two map objects contain same keys

If(map1.keyset().equals(map2.keyset()))

Map Implementations



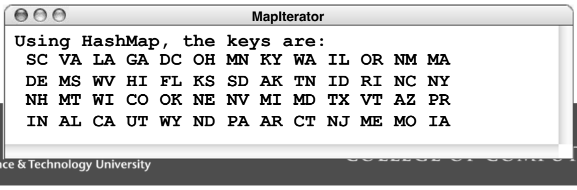
Performance Increases

Iteration Order in a **HashMap**

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map, int nPerLine) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   Iterator<String> iterator = map.keySet().iterator();
   for (int i = 1; iterator.hasNext(); i++) {
      print(" " + iterator.next());
      if (i % nPerLine == 0) println();
   }
}
```

If you call this method on a **HashMap** containing the two-letter state codes, you get:



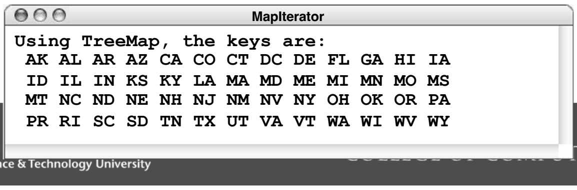
```
Using HashMap, the keys are:
 SC VA LA GA DC OH MN KY WA IL OR NM MA
 DE MS WV HI FL KS SD AK TN ID RI NC NY
 NH MT WI CO OK NE NV MI MD TX VT AZ PR
 IN AL CA UT WY ND PA AR CT NJ ME MO IA
```

Iteration Order in a **`TreeMap`**

The following method iterates through the keys in a map:

```
private void listKeys(Map<String,String> map, int nPerLine) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   Iterator<String> iterator = map.keySet().iterator();
   for (int i = 1; iterator.hasNext(); i++) {
      print(" " + iterator.next());
      if (i % nPerLine == 0) println();
   }
}
```

If you call instead this method on a **`TreeMap`** containing the same values, you get:



```
Using TreeMap, the keys are:
 AK AL AR AZ CA CO CT DC DE FL GA HI IA
 ID IL IN KS KY LA MA MD ME MI MN MO MS
 MT NC ND NE NH NJ NM NV NY OH OK OR PA
 PR RI SC SD TN TX UT VA VT WA WI WV WY
```

Hashtable

- Initially **<u>Hashtable</u>** was not the part of collection framework it has been made a collection framework member later after being retrofitted to implement the Map interface.

- HashMap implements Map interface and is a part of collection framework since the beginning.

Hashtable vs HashMap

- HashMap allows one null key and any number of null values.
- Hashtable doesn’t allow null keys and null values.

Hashtable vs HashMap

- HashMap implementation LinkedHashMap maintains the insertion order and TreeMap sorts the mappings based on the ascending order of keys.
- Hashtable doesn’t guarantee any kind of order. It doesn’t maintain the mappings in any particular order.

Hashtable vs HashMap

- HashMap is non-synchronized. This means if it’s used in multithread environment then more than one thread can access and process the HashMap simultaneously.

- Hashtable is synchronized. It ensures that no more than one thread can access the Hashtable at a given moment of time. The thread which works on Hashtable acquires a lock on it to make the other threads wait till its work gets completed.

SortedSet

- SortedSet stores elements in ascending order as per the natural ordering.
- The iterator and toArray methods iterate and return array in sorted order.
- Additional Services provided:
  - SortedSet <Element> subSet (Element fromElement, Element toElement);
  - SortedSet <Element> headSet (Element toElement);
  - SortedSet <Element> tailSet (Element fromElement);
  - Element first();
  - Element last();

SortedMap

- Organizes data in ascending order based on natural ordering of the.
- Services provided in addition to Map interface:
  - SortedMap<Key, Value> subMap(Key from, Key to);
  - SortedMap<Key, Value> headMap (Key to);
  - SortedMap<Key, Value> tailMap(Key from);
  - Key firstKey();
  - Key lastKey();

Map: Pitfalls

- ThreadSafety
  - The bulk operations like *keySet(), values()* and *entrySet()* return collections that can be modified automatically when the underlying collection changes.

Thread-Safety in Collections

- Collections by default are not thread-safe.
- In order to achieve thread-safety the following utility methods from <Collection> can be employed.

Bibliography

- http://java.sun.com/docs/books/tutorial/collections/index.html
- http://www.purpletech.com/talks/Collections.ppt