Contents **Topic 08 - Collections**

[Ref: Core Java Chp 13 , Intro to Java Programming [Liang] Chp 22, Absolute Java Chp 16, docs.oracle.com/javase/tutorial/collections/TOC.html]

I Introduction - Java Collection Hierarchy

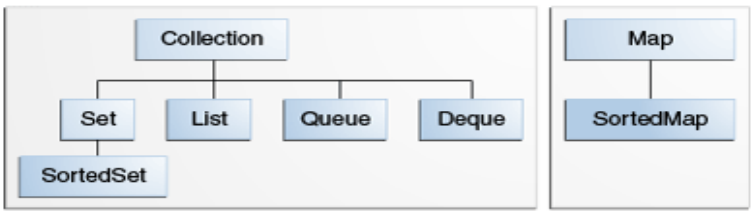
Java Collection Framework

- A collection is a container object that holds a group of objects "Elements"
- A framework is a set of classes, which form the basis for building advanced functionality
- The Java Collections Framework supports different types of collections:

  Containers for storing a collection of elements:
  1. Sets – store a group of non-duplicate elements
  2. Lists – store an ordered collection of elements
  3. Queues– store objects that are processed in first-in, first-out fashion

  Containers for storing key/value pairs:
  4. Maps – store key/value pairs

- Interfaces in the hierarchies:

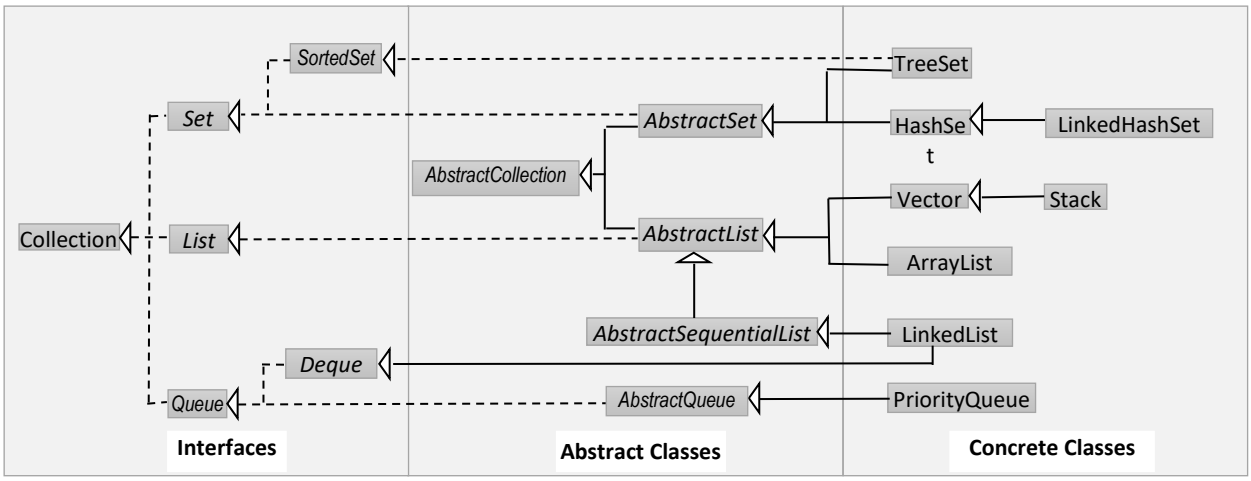


Two distinct trees: Collection and Map

[http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html]

- The Java Collection Framework is an excellent example of using interfaces, abstract classes, and concrete classes.
  - Interfaces – define the framework
  - Abstract classes – provide partial implementation
  - Concrete classes – implement the interfaces with concrete data structures

Some of the interfaces and classes in Java Collection [Liang Chp.22]:



Providing an abstract class (partial implements an interface) makes it convenient for the user to write the code.

The user can simply define a concrete class that extends the abstract class (rather than implementing all methods in the interface)

II Choosing/using collections

- How to choose a data structure from the Java Collection Framework?
  - *Need quick search?*
  - *Data should be kept sorted?*
  - *Rapid insertion/removal in the middle?*
  - *Need association between keys and values?*
  - *etc..*

- The way to use a data structure:

When we use a list, we do not need to know which implementation is actually chosen once it has been constructed.

Therefore we use the interface type for the variable (to hold the reference)

```
import java.util.*;

public class Main
{
   public static void main(String[] args) {
      List<Integer> dataList;
      dataList = new Vector<>();
      dataList.add(100);
      dataList.add(200);
      System.out.println(dataList);
   }
}
```

If we change our mind, we can easily use a different one.
e.g., change to:

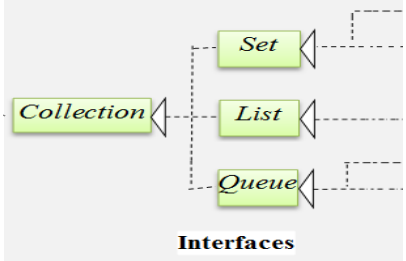
```
dataList = new ArrayList<>();
   or:
dataList = new LinkedList<>();
```

For interested students: http://beginnersbook.com/2013/12/**difference-between-arraylist-and-vector-in-java**/

III Collection and Iterator *(For interested students only)*

- The `Collection` Interface:

```
public interface Collection<E>
{
      //2 fundamental methods:
      boolean add(E element);
      Iterator<E> iterator();
      ..
}
```



- The `iterator` method returns an `Iterator` object.
- The `Iterator` object is for visiting the elements in the collection one by one (See the `Iterator` interface).

```
public static void main(String[] args)
{
      List<Integer> dataList;
      dataList = new ArrayList<>();
      for(int i=0;i<10;i++)
            dataList.add(i*i);

      //(1) Get the iterator and use it to visit elements
      Iterator<Integer> itr = dataList.iterator();
      while (itr.hasNext()){
            Integer e = itr.next();
            System.out.print(e+" ");
      }                                  Output: 0 1 4 9 16 25 36 49 64 81

      //(2) for-each loop (Actually a shortcut for (1))
      for (Integer e: dataList)
            System.out.print(e+" ");
}                                        Output: 0 1 4 9 16 25 36 49 64 81
```

The Iterator Interface:

```
public interface Iterator<E>
{
      E next();
      boolean hasNext();
      ..
}
```

IV Methods of Collection

- The Collection interface is generic
- Usage examples:
  `ArrayList<Employee> emList;`
  `ArrayList<Student> sList;`
- ie., when we create a collection, there is a type parameter for us to provide the class type of the collection elements.

```
public interface Collection<E>
{
      ..
      int size();
      boolean isEmpty();
      boolean contains(Object obj);
      boolean containsAll(Collection<?> c);
      boolean equals(Object other);
      boolean addAll(Collection<? extends E> from);
      boolean remove(Object obj)
      boolean removeAll(Collection<?> c)
      void clear()
      boolean retainAll(Collection<?> c)
      Object[] toArray()
      <T> T[] toArray(T[] arrayToFill)
}
```

```
public abstract class AbstractCollection<E> implements Collection<E>
{
      public boolean contains(Object obj)
      {
            for (E element : this) // calls iterator()
               if (element.equals(obj))
                     return true;
            return false;
      }
      ..
}
```

V Concrete classes

- Some concrete classes:

ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertion and removal at any location
ArrayDeque	A double-ended queue that is implemented as a circular array
HashSet	An unordered set collection (set: rejects duplicates)
TreeSet	A sorted set collection (set: rejects duplicates)
PriorityQueue	A collection that allows efficient removal of the smallest element
HashMap	A map data structure that stores key/value associations
TreeMap	A map data structure that stores key/value associations (sorted by keys)

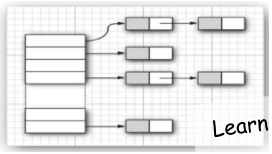
VI Implementation (Hash table, resizable array, tree, linked-list)

- Commonly used implementations (concrete classes) for collection interfaces

General-purpose Implementations				
Interfaces	Resizable array Implementations	Linked list Implementations	Hash table Implementations	Tree Implementations
Set			HashSet	TreeSet
List	ArrayList	LinkedList		
Queue, Deque	ArrayDeque	LinkedList		
Map			HashMap	TreeMap

[http://docs.oracle.com/javase/tutorial/collections/implementations/index.html]

- Hash table implementation: fast lookup, data unsorted, require *hash code*
  In Java, hash tables are implemented as an array of buckets (linked-lists)



- Tree implementation: fast lookup, data sorted, implemented as Red-black tree

Learn in CS3334

VII HashSet, TreeSet, Comparator

HashSet

- Example: using `HashSet<String>` to store the words in "Alice in Wonderland"
- Hash code - an integer for each object to be hashed
  - computed quickly based on the state (field values) of the object
  - determine where to insert the object in the hash table.
- Hash codes for Strings in Java:

String	Hash Code
"Lee"	76268
"lee"	107020
"eel"	100300

```
public static void main(String[] args) throws FileNotFoundException
{
     Set<String> words = new HashSet<>(); // HashSet implements Set
     Scanner in = new Scanner(new File("alice.txt"));
     while (in.hasNext())
     {
          String word = in.next();
          words.add(word);
     }

     System.out.println(words); //Output: [..,..,..] <== all distinct words in the file
     in.close();
}
```

Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

TreeSet

- Example: using `TreeSet<String>` to store the words in "Alice in Wonderland"
- `TreeSet` is: - similar to `Hashset`
  - plus improvement: as sorted collection
    ie. when iterated, values are presented in sorted order
  - insertion is slower than HashSet **but** much faster than array/linked-list

```
public static void main(String[] args) throws FileNotFoundException
{
     Set<String> words = new TreeSet<>(); // TreeSet implements Set

     Scanner in = new Scanner(new File("alice.txt"));
     while (in.hasNext())
     {
          String word = in.next();
          words.add(word);
     }

     System.out.println(words); //Output: [..,..,..] <== all distinct words in the file
     in.close();
}
```

In sorted (alphabetical, case sensitive) order

Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

Comparator

- We've learnt the `Comparable` interface for comparison of objects (also used in sorting)
- But, how to sort items by field `f1` in one collection, then field `f2` in another collection?

```
class Product implements Comparable<Product> {
      private int part_number;          //sometimes we want to compare part_number
      private String product_name;      //sometimes we want compare product name
      public int compareTo(Product other) {return Integer.compare(part_number, other.part_number);
      ..
}
```

- Solution in Java: pass a `Comparator` object into the `TreeSet` constructor
  - `Comparator` is an interface:

```
public interface Comparator<T>
{
      int compare(T a, T b);
}
```

- We often implement it as an inner class (with no class name).

This is an anonymous inner class, which implements the Comparator<Product> interface

```
public static void main(String[] args)
{
   SortedSet<Product> parts = new TreeSet<>();
   parts.add(new Product("Widget", 4562));
   parts.add(new Product("Toaster", 1234));
   parts.add(new Product("Modem", 9912));
   System.out.println(parts);
          [Toaster(1234), Widget(4562), Modem(9912)]

   SortedSet<Product> sortByName = new TreeSet<>(
      new Comparator<Product>()
      {
         public int compare(Product a, Product b)
         {
            String descrA = a.getName();
            String descrB = b.getName();
            return descrA.compareTo(descrB);
         }
      }
   );
   sortByName.addAll(parts);
   System.out.println(sortByName);
}       [Modem(9912), Toaster(1234), Widget(4562)]
```

Ordered by part_number (See compareTo)

Ordered by product_name (See compare)

- Further learning of `Comparator`:
  See lecture exercise:
  Using `Comparator` in `.sort(..)`.

VIII Priority Queue *(For interested students only)*

- **PriorityQueue** (Underlying implementation: priority heap)

  Example: job scheduling

```
class Assignment implements Comparable<Assignment>
{
      private int priority; //1 means highest priority
      private String name; //e.g. "CS2312 Assignment", "CS3342 Project"  "CS3334 Survey"
      public Assignment(String n,int p) { priority=p; name=n; }
      public int compareTo(Assignment other) {return Integer.compare(priority, other.priority);}
      public String toString() {return name+"(Priority:"+priority+")";}
}
```

```
public static void main(String[] args)
{
      PriorityQueue<Assignment> qToDo = new PriorityQueue<>();
      qToDo.add(new Assignment("CS3342 Project", 2));
      qToDo.add(new Assignment("CS3334 Survey", 1));
      qToDo.add(new Assignment("CS2312 Assignment", 1));
      System.out.println(qToDo); //order not guaranteed
            [CS3334 Survey(Priority:1), CS3342 Project(Priority:2), CS2312 Assignment(Priority:1)]

      System.out.println(qToDo.remove()); //removed based on priority
      System.out.println(qToDo.remove());
      System.out.println(qToDo.remove());
}
                                          CS3334 Survey(Priority:1)
                                          CS2312 Assignment(Priority:1)
                                          CS3342 Project(Priority:2)
```

IX HashMap, TreeMap

- **The `Map` Interface (implementing classes: `HashMap`, `TreeMap`)**

  - A map stores key/value pairs. Both key and value must be objects.

    Example: we have some key info, we want to look up the associated element.

- **Implementing classes:**

  `HashMap` unordered implementation of `Map`; hashing the key (Learn in CS3334)

  `TreeMap` ordered implementation of `Map`; ordering on the key[which implements Comparable] (Red-black tree; Learn in CS3334)

  - Both `HashMap` and `TreeMap` hash/compare on keys.
  - For `Hashmap`, the class of the keys needs to provide `equals()` and `hashCode()`. `hashCode()` should return the hash code such that 2 objects which are considered as equal should have the same *hash code*.

    How JAVA locates an object: Find the location with `hashcode()`; then use `equals()` to identify it.

  - Useful "view" methods to get the set of keys, collection of values, or set of key-value pairs.

- **Example:** `.get, .put`

```
class Product {
      private String product_name;
      public Product(String name) {product_name=name;}
      public String toString() {return product_name;}
}
```

```
public static void main(String[] args)
{
      Map<Integer,Product> parts = new HashMap<>();
      parts.put(4562,new Product("Widget"));
      parts.put(1234,new Product("Toaster"));
      parts.put(9912,new Product("Modem"));

      System.out.println(parts.get(9912));      //output: Modem
      parts.put(9912,new Product("Router"));
      System.out.println(parts.get(9912));      //output: Router

      //Get the view: a set of the keys, for iteration;
      //output: Toaster(1234) Router(9912) Widget(4562)
      Set<Integer> kSet=parts.keySet();
      for(Integer k:kSet)
            System.out.printf("%s(%d) ",parts.get(k),k);
}
```

X Conversion to/from array *(For interested students only)*

From array to collection wrapper

- **Lightweight Collection Wrappers:** created using `asList(..)` of the `Arrays` class

```
public static void main(String[] args)
{
    String [] nArr = {"Helena", "Kit", "Jason"};    //an array

    List<String> nList = Arrays.asList(nArr);       //returns a List wrapper

    nList.set(0, "Marian");  //cannot apply .add or .remove which changes array size

    System.out.println(Arrays.toString(nArr));      //output: [Marian, Kit, Jason]
    System.out.println(nList);                      //output: [Marian, Kit, Jason]
}
```

From collection to an array copy

- **`Collection.toArray(..)`**

```
public static void main(String[] args)
{
    Collection<String> c = new ArrayList<>();
    c.add("Helena");c.add("Kit");c.add("Jason");
    String[] arr = c.toArray(new String[1]); //create a new array copy; initial size=1; expand as needed

    arr[0]="Tom";

    System.out.println(c);                          //output: [Helena, Kit, Jason]
    System.out.println(Arrays.toString(arr));       //output: [Tom, Kit, Jason]
}
```

XI Simple algorithms: shuffling, sorting, binarySearch, reverse, disjoint..

- **`Collections.xxx(..)` – Useful methods for collections**

```
public static void main(String[] args)
{
    List<String> c = new ArrayList<>(); c.add("Helena");c.add("Kit");c.add("Jason");
    System.out.println(c); //Output: [Helena, Kit, Jason]

    //Sorting
    Collections.sort(c); System.out.println(c);             //Output: [Helena, Jason, Kit]

    //Binary search
    System.out.println(Collections.binarySearch(c,"Helena")); //returns the index: 0

    //Others:
    System.out.println(Collections.disjoint(c,c.subList(0, 1)));  //Output: false
}

//Other methods: max, min, frequency, reverse, rotate, shuffle
```

true if no elements in common