Unit-4

Developing Real World Applications Using Collections:

- Storing user defined Classes in Collections,
- The Random Access Interface,
- Working withMaps,
- Comparators,
- The Collection Algorithms, and
- Arrays Class.

Storing user defined Classes in Collections

- collections are not limited to the storage of built-in objects.,
- they can store any type of object, including objects of classes that you create.

```
public class Book {
    String title;
    String author;
    double price;
    public Book(String title2, String author2, double price2) {
        title=title2;
        author=author2;
        price=price2;
    }
    public String toString() {
        return "Book [title=" + title + ", author=" + author + ", price=" +
        price + "]";
    }
}
```

Create an ArrayList that contains user-defined objects

- List that stores a list of books, which are objects of user-defined class Book

```
ArrayList<Book> books=new ArrayList<>();
```

- Create a student list, which are objects of user-defined class Student

```
ArrayList<Student> students=new ArrayList<>();
```

```
public class BookDemo {
    public static void main(String[] args) {
        ArrayList<Book> books=new ArrayList<>();
        books.add(new Book("java","sierra & bates",650.0));
        books.add(new Book("java","dietel",320.0));
        books.add(new Book("effective java","joshua Bloch",810.0));
        books.add(new Book("Complete reference","dietel and dietel",500.0));
        for (Book b: books)
            System.out.println(b);
    }
}
```

Output

```
Book [title=java, author=sierra & bates, price=650.0]
Book [title=java, author=dietel, price=320.0]
Book [title=effective java, author=joshua Bloch, price=810.0]
Book [title=Complete reference, author=dietel and dietel, price=500.0]
```

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.
- It is an alternative of Dictionary class , which is an abstract class.

Syntax

public interface **Map<K,V>**

The Map interface provides it's contents as three *collection views*

i) as a set of keys,
ii) collection of values, or
iii) set of key-value mappings.

The Map interface includes methods for

- basic operations : put, get, remove, containsKey, containsValue, size, and empty
- bulk operations : putAll and clear
- collection views : keySet, entrySet, and values

Interface	Description
Map	Maps unique keys to values
Map.Entry	Describes an element (a key/value pair) in a map
SortedMap	Extends Map so that the keys are Maintained in ascending order.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches.

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

Available in a package java.util.*

Methods used in Maps

Method	**Description**
V put(K key, V Value)	Inserts a (key,value) pair
void putAll(Map<? extends K,? extends V> m)	Copies all of the mappings from the specified map to this map (optional operation).
V get(Object key)	Returns the value associated with the key specified, or null if map contains no mapping for the key
V remove(Object key)	Removes the mapping for a key from this map if it is present (optional operation).
boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
boolean containsValue(Object key)	Returns true if this map maps one or more keys to the specified value.
Set<K> keyset()	Returns a set view of keys contained in the Map
Collection values()	Returns a Collection view of the values contained in the map.
Set<Map.Entry<K,V>> entrySet()	Returns a set view of the mappings (pairs) contained in the map

Method	Description
V **putIfAbsent**(**K** key, **V** value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V **getOrDefault**(**Object** key, **V** defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
Using lambda functions	
compute(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction)Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).	
computeIfAbsent(**K** key, **Function**<? super **K**,? extends **V**> mappingFunction)If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.	
computeIfPresent(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction)If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.	

HashMap

public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

- It is a Hash table based implementation of the Map interface.
- It provides all of the optional map operations implementations, and permits null values and the null key.
- The difference between HashMap and Hashtable is , it is synchronized where as HashMap is not.
- It does not guarantee that the order will remain constant over time.
- Constant-time performance for the basic operations (like get and put)

Constructors
HashMap() Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity) Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
HashMap(int initialCapacity, float loadFactor) Constructs an empty HashMap with the specified initial capacity and load factor.
HashMap(**Map**<? extends **K**,? extends **V**> m) Constructs a new HashMap with the same mappings as the specified Map.

Example : Finding the frequency of the words

```
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class FreqTable {
   public static void main(String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();
      String[] s= {"if", "it", "is", "to", "be", "it", "is", "up", "to",
                   "me", "to", "delegate"};
      for (String a : s) {
         Integer freq = m.get(a);
         m.put(a, (freq == null) ? 1 : freq + 1);
      }
      System.out.println(m.size() + " distinct words:");
      System.out.println(m);
   }
}
```

Output:

```
8 distinct words:
{delegate=1, be=1, me=1, is=2, it=2, to=3, up=1, if=1}
```

TreeMap

public class **TreeMap<K,V>** extends **AbstractMap**<K,V>
implements **NavigableMap**<K,V>, **Cloneable, Serializable**

- A Red-Black tree based NavigableMap implementation.
- The map is sorted according to the **natural ordering** of its keys, or by a **Comparator** provided at map creation time, depending on which constructor is used.
- This implementation provides guaranteed log(n) time cost for the containsKey,get,put, and remove operations.

Constructors
TreeMap() Constructs a new, empty tree map, using the natural ordering of its keys.
TreeMap(**Comparator**<? super **K**> comparator) Constructs a new, empty tree map, ordered according to the given comparator.
TreeMap(**Map**<? extends **K**,? extends **V**> m) Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
TreeMap(**SortedMap**<**K**,? extends **V**> m)Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

Difference between HashMap vs TreeMap vs LinkedHashMap

```
Input:   String[] s= {"if", "it", "is", "to", "be", "it", "is", "up", "to",
                      "me", "to", "delegate"};
```

Map	Implementation

HashMap
(no particular order)
	`Map<String, Integer> m = new HashMap<String, Integer>();`
	
	`Output:`
	`8 distinct words:`
	`{delegate=1, be=1, me=1, is=2, it=2, to=3, up=1, if=1}`

TreeMap
(ascending order)
	`Map<String, Integer> m = new TreeMap<String, Integer>();`
	
	`Output:`
	`8 distinct words:`
	`{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}`

LinkedHashMap
(maintains insertion order)
	`Map<String, Integer> m = new LinkedHashMap<String, Integer>();`
	
	`Output:`
	`8 distinct words:`
	`{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}`

WeakHashMap

public class **WeakHashMap<K,V>** extends
AbstractMap<K,V> implements Map<K,V>

- Hash table based implementation of the Map interface, with weak keys.
- An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently from other Map implementations.

WeakHashMap() Constructs a new, empty WeakHashMap with the default initial capacity (16) and load factor (0.75).
WeakHashMap(int initialCapacity) Constructs a new, empty WeakHashMap with the given initial capacity and the default load factor (0.75).
WeakHashMap(int initialCapacity, float loadFactor) Constructs a new, empty WeakHashMap with the given initial capacity and the given load factor.
WeakHashMap(Map<? extends K,? extends V> m) Constructs a new WeakHashMap with the same mappings as the specified map.

IdentityHashMap

- This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).

- That is, in an IdentityHashMap, two keys k1 and k2 are considered equal if and only if (k1==k2). (In normal Map implementations (like HashMap) two keys k1 and k2 are considered equal if and only if (k1==null ? k2==null : k1.equals(k2)).)

- This class is not a general-purpose Map implementation! While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals method when comparing objects. This class is designed for use only in the rare cases wherein reference-equality semantics are required.

EnumMap

- Similar to HashMap but it uses for Enum type instead of class type

Comparators

- Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps) or to provide an ordering for collections of objects that don't have a natural ordering.

- Comparators can be passed to a sort method
  - Collections.sort(collectionobj,comparatorobj);
  - Arrays.sort(comparatorobj)

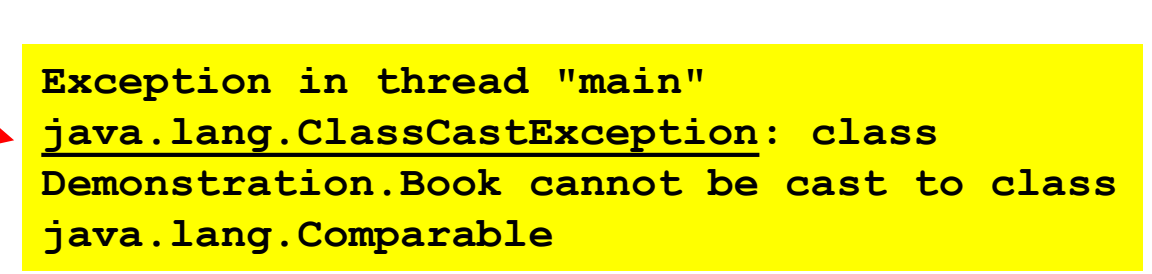
- The ordering provided by a comparator c on a set of element S is consistent with equals method
  i.e., c,compare(e1,e2)==0 is same as e1.equals(e2) for every e1 and e2 in S.

- It is a Functional Interface which can be used as a target for the lambda expression or method reference.

The method to be implemented is

```
int compare(T o1,T o2){
        /*  …*/
}
```

```
public class Book {
    String title;
    String author;
    double price;
    public Book(String title,String author,double price){
        this.title=title;
        this.author=author;
        this.price=price;
    }
    /* GETTER AND SETTER METHODS OF TITLE,AUTHOR AND PRICE*/
}
public class BookDemo {
    public static void main(String[] args) {
        ArrayList<Book> books=new ArrayList<>();
        books.add(new Book("java","sierra & bates",650.0));
        books.add(new Book("java","dietel",320.0));
        books.add(new Book("effective java","joshua Bloch",810.0));
        books.add(new Book("Complete reference","dietel and dietel",500.0));
        books.sort(null);
        for (Book b: books)
                System.out.println(b);
    }
}
```



The problem of sorting class objects can be resolved in two ways

1. Implement comparable Interface and override compareTo() method.
2. Define a comparator object and pass to sort method as an argument

1. Implement comparable Interface

```
public class Book implements Comparable<Book>{
    String title;
    String author;
    double price;
    public Book(String title,String author,double price){
        this.title=title;this.author=author;this.price=price;
    }
    /* GETTER AND SETTER METHODS OF TITLE,AUTHOR AND PRICE*/
    public int compareTo(Book o) {
        return title.compareTo(o.getTitle());
    }
}
public class BookDemo {
    public static void main(String[] args) {
        ArrayList<Book> books=new ArrayList<>();
        /*….*/
        books.sort(null);
    }
}
```

Output

```
Book [title=complete reference, author=dietel and dietel, price=500.
Book [title=effective java, author=joshua Bloch, price=810.0]
Book [title=java, author=sierra & bates, price=650.0]
Book [title=java, author=dietel, price=320.0]
```

We are not sure whether the class implements comparable and on what basis the ordering is defined.

2. Using Comparator Interface

```
public class BookDemo {
    public static void main(String[] args) {
        ArrayList<Book> books=new ArrayList<>();
        books.add(new Book("java","sierra & bates",650.0));
        books.add(new Book("java","dietel",320.0));
        books.add(new Book("effective java","joshua Bloch",810.0));
        books.add(new Book("Complete reference","dietel and dietel",500.0));
        books.sort(new Comparator<Book2>() {
                    public int compare(Book2 o1, Book2 o2) {
                        return o1.getTitle().compareTo(o2.getTitle());
                    }
        });
        for (Book b: books)
                System.out.println(b);
    }
}
```

Anonymous class	`books.sort(new Comparator<Book2>() {`
	`  public int compare(Book2 o1, Book2 o2) {`
	`  return o1.getTitle().compareTo(o2.getTitle());`
	`  }`
	`  });`

(or)	
Lambda Expression	`books.sort((b3,b4)->b3.getTitle().compareTo(b4.getTitle()));`
(or)	
Comparator's Comparing function with Lambda Exp.	`Comparator<Book> c=Comparator.comparing((b)->b.getTitle());` `books.sort(c);`
(or)	
Comparator's Comparing function with Lambda Exp.	`Comparator<Book> c=Comparator.comparing(Book::getTitle);`

If you want to compare more than one field i.e., title and if title is same then compare author

Using Comparable Interface

```
Class Book implements Comparable<Book>{
    /*……*/
    public int compareTo(Book o) {
        if(title.compareTo(o.getTitle())==0)
            return author.compareTo(o.getAuthor());
        return title.compareTo(o.getTitle());
    }
}
```

Using Comparator Interface

```
Comparator<Book> c1=Comparator.comparing(Book::getTitle)
                              .thenComparing((b)->b.getAuthor());
```

Anonymous class

```
books.sort(new Comparator<Book2>() {
                    public int compare(Book2 o1, Book2 o2) {
                        return o1.getTitle().compareTo(o2.getTitle());
                    }
                });
```

(or)

Lambda Expression

```
books.sort((b3,b4)->b3.getTitle().compareTo(b4.getTitle()));
```

(or)

Comparing static function

```
Comparator<Book> c=Comparator.comparing((b)->b.getTitle());
books.sort(c);
```

(or)

Comparing static function with method Reference

```
Comparator<Book> c=Comparator.comparing(Book::getTitle);
```

Enum

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Algorithms

static <T> int	**binarySearch**(**List**<? extends **Comparable**<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(**List**<? extends T> list, T key, **Comparator**<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static boolean	**disjoint**(**Collection**<?> c1, **Collection**<?> c2) Returns true if the two specified collections have no elements in common.
static int	**frequency**(**Collection**<?> c, **Object** o) Returns the number of elements in the specified collection equal to the specified object.

static <T extends **Object** & **Comparable**<? super T>> T	**max**(**Collection**<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**max**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends **Object** & **Comparable**<? super T>> T	**min**(**Collection**<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> **List**<T>	**nCopies**(int n, T o) Returns an immutable list consisting of n copies of the specified object.

static <T> boolean	**replaceAll**(**List**<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	**rotate**(**List**<?> list, int distance) Rotates the elements in the specified list by the specified distance.

static void	swap(List<?> list, int i, int j)Swaps the elements at the specified positions in the specified list.

Algorithms

- Sorting
- Shuffling
- Routine Data Manipulation
- Searching
- Composition
- Finding Extreme Values

Sorting

The sort operation uses a slightly **optimized *merge sort* algorithm** that is fast and stable:

- **Fast**: It is guaranteed to run in n log(n) time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n log(n) performance.

- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

static <T extends **Comparable**<? super T>> void	**sort**(**List**<T> list)Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
static <T> void	**sort**(**List**<T> list, **Comparator**<? super T> c)Sorts the specified list according to the order induced by the specified comparator.

Shuffling

- It reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness.
- It is useful in
  - ❑ implementing games of chance. For example, it could be used to shuffle a List of Card objects representing a deck.
  - ❑ for generating test cases.

static void	**shuffle**(**List**<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(**List**<?> list, **Random** rnd) Randomly permute the specified list using the specified source of randomness.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Routine Data Manipulation

- **reverse** — reverses the order of the elements in a List.
- **fill** — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
- **copy** — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- **swap** — swaps the elements at the specified positions in a List.
- **addAll** — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

static void	**reverse**(**List**<?> list) Reverses the order of the elements in the specified list.
static <T> **Comparator**<T>	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.
static <T> **Comparator**<T>	**reverseOrder**(**Comparator**<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.
static <T> void	**fill**(**List**<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static <T> void	**copy**(**List**<? super T> dest, **List**<? extends T> src)Copies all of the elements from one list into another.
static void	**swap**(**List**<?> list, int i, int j)Swaps the elements at the specified positions in the specified list.
static <T> boolean	**addAll**(**Collection**<? super T> c, T... elements)Adds all of the specified elements to the specified collection.

Searching

static <T> int	**binarySearch**(**List**<? extends **Comparable**<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(**List**<? extends T> list, T key, **Comparator**<? super T> c) Searches the specified list for the specified object using the binary search algorithm.

Rollnumber	Name	Gender	CGPA
10501A0520	Anitha K	Female	8.96
10501A0506	Ramprasad G	male	9.01
10501A0531	Arunkumar R	Female	8.56
10501A0514	Rajesh M	Male	9.26
10501A0543	Sophia R	Female	9.12
10501A0581	Ramesh H	Male	9.11

Binary search for user-defined classes

static <T> int	binarySearch(List<? extends T> list, T key, Comparator<? super T> c) Searches the specified list for the specified object using the binary search algorithm.

1. Define a comparator for sorting the user-defined classes
2. Pass the comparator argument to binarySearch()

```
List<Book2> books=new ArrayList<>();
books.add(new Book2("java","sierra & bates",650.0));
books.add(new Book2("effective java","joshua Bloch",810.0));
books.add(new Book2("complete reference","dietel and dietel",500.0));


Book2 b2=new Book2("java","sierra & bates",650.0);

Comparator<Book2> c1=Comparator.comparing((b)->b.getAuthor());
books.sort(c1);

int f=Collections.binarySearch(books, b2,c1);
System.out.println(f); // prints 2
```

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections:

- frequency — counts the number of times the specified element occurs in the specified collection
- disjoint — determines whether two Collections are disjoint; that is, whether they contain no elements in common

int	**frequency**(**Collection**<?> c, **Object** o)Returns the number of elements in the specified collection equal to the specified object.
static boolean	**disjoint**(**Collection**<?> c1, **Collection**<?> c2)Returns true if the two specified collections have no elements in common.

static int	**indexOfSubList**(List<?> source, List<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	**lastIndexOfSubList**(List<?> source, List<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T extends Object & Comparable<? super T>> T	**max**(Collection<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**max**(Collection<? extends T> coll, Comparator<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends Object & Comparable<? super T>> T	**min**(Collection<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(Collection<? extends T> coll, Comparator<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> List<T>	**nCopies**(int n, T o) Returns an immutable list consisting of n copies of the specified object.