Java Technologies
Collections Framework and more

Prepared for: DDU CE Semester 4

Prepared by: Prof. Niyati J. Buch

Collections

- A collection is an object that represents a group of objects.
- A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

Advantages

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

The collections framework consists of:

- **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- **General-purpose implementations**. Primary implementations of the collection interfaces.
- **Legacy implementations**. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**. Implementations designed for highly concurrent use.

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

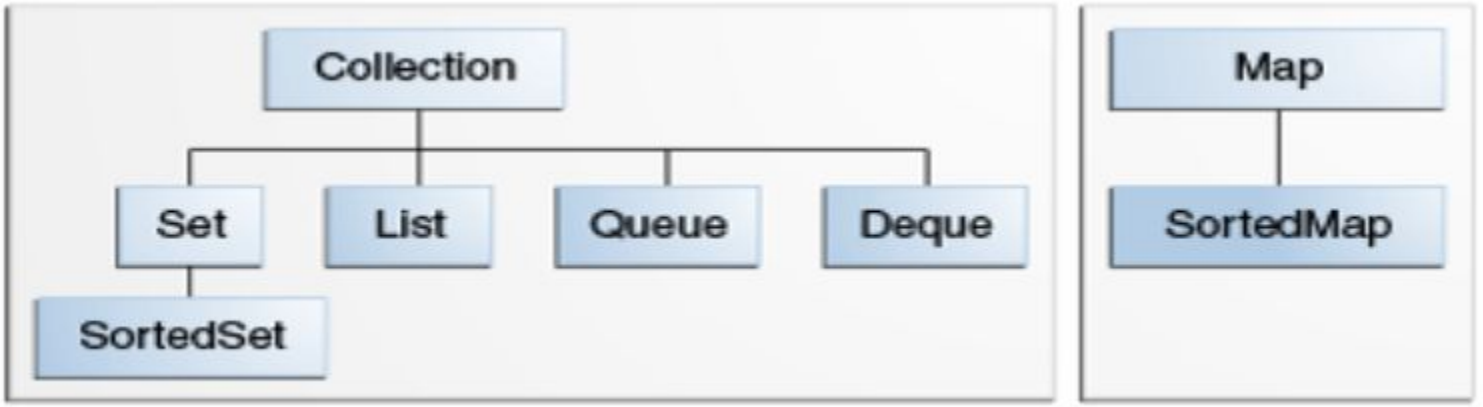
The collections framework consists of: (cont.)

- **Wrapper implementations**. Add functionality, such as synchronization, to other implementations.
- **Convenience implementations**. High-performance "mini-implementations" of the collection interfaces.
- **Abstract implementations**. Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms**. Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure**. Interfaces that provide essential support for the collection interfaces.
- **Array Utilities**. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

The core collection interfaces

All the core collection interfaces are generic.



https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

- **Collection** — the root of the collection hierarchy. A collection represents a group of objects known as its elements. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.
- **Set** — a collection that cannot contain duplicate elements.
- **List** — an ordered collection (sometimes called a **sequence**). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- **Queue** — a collection used to hold multiple elements prior to processing. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are **priority queues**, which order elements according to a supplied comparator or the elements' natural ordering.
- **Deque** — a collection used to hold multiple elements prior to processing. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.

- **Map** — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

- **SortedSet** — a Set that maintains its elements in ascending order. Sorted sets are used for naturally ordered sets.

- **SortedMap** — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs.

Operations on Collections

- Basic operations
  - int size()
  - boolean isEmpty()
  - boolean contains(Object element)
  - boolean add(E element)
  - boolean remove(Object element)
  - Iterator<E> iterator()
- Methods for array operations
  - Object[] toArray()
  - <T> T[] toArray(T[] a)
- Operations on entire collection
  - boolean containsAll(Collection<?> c)
  - boolean addAll(Collection<? extends E> c)
  - boolean removeAll(Collection<?> c)
  - boolean retainAll(Collection<?> c)
  - void clear()

To traverse a Collection

```
for (Object o : collection)
    System.out.println(o);
```

```
public interface Iterator<E> { boolean
     hasNext();
    E next();
    void remove(); //optional
}
```

Use Iterator instead of the for-each construct when you need to:

- Remove the current element.
  - The for-each construct hides the iterator, so you cannot call remove.
  - Therefore, the for-each construct is not usable for filtering.
- Iterate over multiple collections in parallel.

https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html

The Set Interface

- A Set is a Collection that cannot contain duplicate elements.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.
- Two Set instances are equal if they contain the same elements.

Three general purpose Set implementations

1. **HashSet**, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.
2. **TreeSet**, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.
3. **LinkedHashSet**, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

Suppose you want to know which words occur only once and which occur more than once, it can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates.

```
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo {
   public static void main(String args[]) {
      String s[] = {"the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"};
      Set<String> unique = new HashSet<>();
      Set<String> repeat = new HashSet<>();
      for (String a : s) {
         if (!unique.add(a)) {
            repeat.add(a);
         }
      }
      unique.removeAll(repeat); // Destructive set-difference
      System.out.println("Unique words: " + unique);
      System.out.println("Duplicate words: " + repeat);
   }
}
```

OUTPUT

Unique words: [over, quick, lazy, jumps, brown, dog, fox]

Duplicate words: [the]

Suppose you want to know which words occur only once and which occur more than once, it can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates.

```
import java.util. LinkedHashSet;
import java.util.Set;
public class LinkedHashSetDemo {
   public static void main(String args[]) {
      String s[] = {"the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"};
      Set<String> unique = new LinkedHashSet<>();
      Set<String> repeat = new LinkedHashSet<>();
      for (String a : s) {
         if (!unique.add(a)) {
            repeat.add(a);
         }
      }
      unique.removeAll(repeat); // Destructive set-difference
      System.out.println("Unique words: " + unique);
      System.out.println("Duplicate words: " + repeat);
   }
}
```

OUTPUT

Unique words: [quick, brown, fox, jumps, over, lazy, dog]

Duplicate words: [the]

Suppose you want to know which words occur only once and which occur more than once, it can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates.

```
import java.util. TreeSet;
import java.util.Set;
public class TreeSetDemo {
   public static void main(String args[]) {
      String s[] = {"the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"};
      Set<String> unique = new TreeSet<>();
      Set<String> repeat = new TreeSet<>();
      for (String a : s) {
         if (!unique.add(a)) {
            repeat.add(a);
         }
      }
      unique.removeAll(repeat); // Destructive set-difference
      System.out.println("Unique words: " + unique);
      System.out.println("Duplicate words: " + repeat);
   }
}
```

OUTPUT

Unique words: [brown, dog, fox, jumps, lazy, over, quick]

Duplicate words: [the]

```
import java.util.*;
public class ComparingSet {
   public static void main(String args[]) {
      String s[] = {"the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"};
      Set<String> words = new LinkedHashSet<>();
      Set<String> sorted = new TreeSet<>();
      words.addAll(Arrays.asList(s));
      sorted.addAll(words);
      System.out.println("words: " + words);
      System.out.println("sorted: " + sorted);
      System.out.println("words and sorted are same: " + words.equals(sorted));
      sorted.remove("the");
      System.out.println("removed \"the\": " + sorted);
      System.out.println("words and sorted are disjoint: " + Collections.disjoint(sorted, words));
   }
}
```

```
OUTPUT
words: [the, quick, brown, fox, jumps, over, lazy, dog]
sorted: [brown, dog, fox, jumps, lazy, over, quick, the]
words and sorted are same : true
removed "the": [brown, dog, fox, jumps, lazy, over, quick]
words and sorted are disjoint : false
```

The List Interface

- A **List** is an ordered Collection (sometimes called a *sequence*).
- Lists may contain duplicate elements.
- In addition to the operations inherited from Collection:-
  - Positional access: get(), set(), add(), addAll(), and remove().
  - Search: indexOf() and lastIndexOf().
  - Iteration: listIterator()
  - Range-view: sublist()
- Two general- purpose List implementations:
  - **ArrayList**
  - **LinkedList**

List Algorithms

- **sort**: sorts a List using a merge sort algorithm, which provides a fast, stable sort.
- **shuffle:** randomly permutes the elements in a List.
- **reverse**: reverses the order of the elements in a List.
- **rotate**: rotates all the elements in a List by a specified distance.
- **swap:** swaps the elements at specified positions in a List.
- **replaceAll:** replaces all occurrences of one specified value with another.
- **fill**: overwrites every element in a List with the specified value.
- **copy:** copies the source List into the destination List.
- **binarySearch:** searches for an element in an ordered List using the binary search algorithm.
- **indexOfSubList:** returns the index of the first sublist of one List that is equal to another.
- **lastIndexOfSubList:** returns the index of the last sublist of one List that is equal to another.

https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class ArrayListDemo {
   public static void main(String args[]) {
      String s[] = {"cocoanut", "hazelnut", "walnut", "peanut", "betalnut", "hazelnut"};
      ArrayList<String> al = new ArrayList<>();
      al.addAll(Arrays.asList(s));
      //for (String a : s)
      //    al.add(a);
      System.out.println("Original: "  +al);
      Collections.sort(al);
      System.out.println("Sorted: "  +al);
      List<String> subList = al.subList(al.indexOf("hazelnut"), al.size());
      System.out.println("Sublist: "  +subList);
   }
}
```

OUTPUT

```
Original: [cocoanut, hazelnut, walnut, peanut, betalnut, hazelnut]
Sorted: [betalnut, cocoanut, hazelnut, hazelnut, peanut, walnut]
Sublist: [hazelnut, hazelnut, peanut, walnut]
```

```
import java.util.*;
public class LinkedListDemo {
   public static void main(String args[]) {
      String s[] = {"cocoanut", "hazelnut", "walnut", "peanut", "betalnut", "hazelnut"};
      List<String> list = new LinkedList<>();
      list.addAll(Arrays.asList(s));
      System.out.println("Original: " + list);
      Collections.sort(list);
      System.out.println("Sorted: " + list);
      List<String> subList = list.subList(list.indexOf("hazelnut"), list.size());
      System.out.println("Sublist: " + subList);
      Collections.shuffle(subList);
      System.out.println("Shuffled Sublist: " + subList);
      Collections.reverse(list);
      System.out.println("Reversed: " + list);
   }
}
```

OUTPUT

```
Original: [cocoanut, hazelnut, walnut, peanut, betalnut, hazelnut]
Sorted: [betalnut, cocoanut, hazelnut, hazelnut, peanut, walnut]
Sublist: [hazelnut, hazelnut, peanut, walnut]
Shuffled Sublist: [walnut, peanut, hazelnut, hazelnut]
Reversed: [hazelnut, hazelnut, peanut, walnut, cocoanut, betalnut]
```

The 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.
- The Map interface includes methods for:
  - basic operations (such as put, get, remove, containsKey, containsValue, size and empty)
  - bulk operations (such as putAll and clear)
  - collection views (such as keySet, entrySet, and values)
- Three general purpose Map implementations:
  - **HashMap**, **TreeMap**, and **LinkedHashMap**
- Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet

Generate a frequency table of the words in a given string. The frequency table maps each word to the number of times it occurs in the string.

```
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo {
   public static void main(String args[]) {
      Map<String, Integer> m = new HashMap<>();
      String str = "Strawberry Raspberry Blueberry Raspberry Cranberry Cranberry Raspberry";
      String s[] = str.split(" ");
      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

```
4 distinct words
{Strawberry=1, Blueberry=1, Raspberry=3, Cranberry=2}
```

Generate a frequency table of the words in a given string. The frequency table maps each word to the number of times it occurs in the string.

```
import java.util.LinkedHashMap;
import java.util.Map;
public class LinkedHashMapDemo {
   public static void main(String args[]) {
      Map<String, Integer> m = new LinkedHashMap<>();
      String str = "Strawberry Raspberry Blueberry Raspberry Cranberry Cranberry Raspberry";
      String s[] = str.split(" ");
      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
4 distinct words
{Strawberry=1, Raspberry=3, Blueberry=1, Cranberry=2}
```

Generate a frequency table of the words in a given string. The frequency table maps each word to the number of times it occurs in the string.

```
import java.util.TreeMap;
import java.util.Map;
public class TreeMapDemo {
   public static void main(String args[]) {
      Map<String, Integer> m = new TreeMap<>();
      String str = "Strawberry Raspberry Blueberry Raspberry Cranberry Cranberry Raspberry";
      String s[] = str.split(" ");
      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
4 distinct words
{Blueberry=1, Cranberry=2, Raspberry=3, Strawberry=1}
```

Comparator Interface

- A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order.
- 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.

```
package mypack;
public class Student implements Comparable<Student> {
   private String name;
   private double cpi;
   public Student(String name, double cpi) {
      this.name = name;
      this.cpi = cpi;
   }
   @Override
   public String toString() {
      return "Student{" + "name=" + name + ", cpi=" + cpi + '}';
   }
   @Override
   public int compareTo(Student arg0) {
      if (this.cpi > arg0.cpi) {
         return 1;
      } else if (this.cpi < arg0.cpi) {
         return -1;
      } else {
         return this.name.compareTo(arg0.name);
      }
   }
}
```

```
package mycomparator;
import java.util.Comparator;
import mypack.Student;
public class StudentComparator implements Comparator<Student> {
   @Override
   public int compare(Student arg0, Student arg1) {
      return arg0.compareTo(arg1);
   }
}
```

```
package mysets;
import java.util.*;
import mypack.Student;
public class StudentTreeSet {
   public static void main(String args[]) {
      Student s[] = {new Student("abc", 9.0), new Student("xyz", 7.0),
            new Student("pqr", 9.8), new Student("def", 9.8)};
      Set<Student> set = new TreeSet<>(new StudentComparator());
      set.addAll(Arrays.asList(s));
      Iterator<Student> i = set.iterator();
      while (i.hasNext()) {
         System.out.println(i.next());
      }
   }
}
```

OUTPUT

```
Student{name=xyz, cpi=7.0}
Student{name=abc, cpi=9.0}
Student{name=def, cpi=9.8}
Student{name=pqr, cpi=9.8}
```

Topics covered so far

- Collection Framework
  - Set
  - List/Sequence
  - Map
  - Collections class
  - Comparable and Comparator interface