# Lecture - The Java Collection Framework

## Introduction

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

## What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

**Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

**Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

set: unorder and unrepeated collection of things.

## Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

Reduces programming effort: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the

Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

Increases program speed and quality: This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

Allows interoperability among unrelated APIs: The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

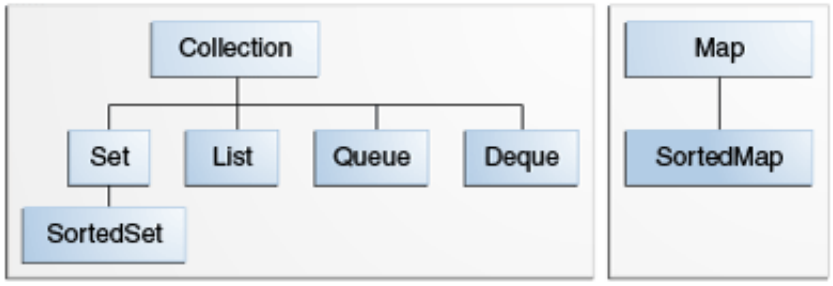
Reduces effort to learn and to use new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

## Interfaces

The Collections Framework defines several core interfaces. The main interfaces are Collection, Set, List, Queue, Dequeue, Sorted Set, Map and Sorted Map. The table below shows a brief description of each interface.




Note: Map is a part of Collection Framework but don't use the Collection interface. A Map is not a true Collection.

In this unit, we only cover Set, List and Map interface and their related classes.

`Collection`: the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

Collection is a generic interface that has this declaration:

```
interface Collection<E>
```

`Set`: a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

Set is a generic interface that has this declaration:

```
interface Set<E>
```

`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). If you've used Vector, you're familiar with the general flavor of List.

List is a generic interface that has this declaration:

```
interface List <E>
```

`Map`: an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

Map is a generic interface that has this declaration:

```
interface Map<E>
```

In array, you need to resizing

but in list. it just added at the end of the list.

## Iterating over collections

As collections are group of data, we need to iterator over them. The Collection framework provides many ways to do it.

### Iterator interface

The traditional way is to use and Iterator interface.

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

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class IteratorDemo {
    public static void main(String[] args) {
        Collection collection = new ArrayList();
        collection.add("Apple");
        collection.add("Banana");
        collection.add("Orange");
        /*
        Need to call .iterator() method
        chnage collection with any type of collection such as ArrayList.
        **/
        Iterator iterator = collection.iterator();
        while (iterator.hasNext()) {// looking for the next element if existed
            Object element = iterator.next();//finding the next element and putting it in a the ite
            System.out.println(element);
        }
    }
}
```

Another way we can iterator through a  collection is using a traditional for loop. Replace the line 15 to 19 with the following code.

```
for (int i = 0; i < elements.length; i++) {
    String element = elements[i];
    System.out.println(element);
}
```

The modern for-each loop that we covered in previous chapter is another way of iterating through the loop. Performance wise, it is faster but a bit hard to use due to its structure. Replace the line 15 to 19 with the following code.

```
for (String element : elements) {
    System.out.println(element);
}
```

```
import java.util.ArrayList;
import java.util.List;

public class ForEachDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");

        for (String element : list) {
            System.out.println(element);
        }
    }
}
```

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("Apple");
    list.add("Banana");
    list.add("Orange");

    for (String element : list) {
        System.out.println(element);
    }

    for(int i = 0; i < list.size(); i++){
        System.out.println(list.get(i));
    }
}
```

list.add("Kiwi");
list.add(1, "Kiwi"); // add at the index1

## The Comparator Interface

Comparator is a generic interface that has this declaration:

```
interface Comparator <T>
```

Here, T specifies the type of objects being compared.

It is used to compare and order the objects in the collection.

The details of Comparater interface are accesible at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html

A simple sorting example using comparator is given below

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        // Sort the list using Collections.sort and the custom comparator
        Collections.sort(people, new AgeComparator());

        System.out.println("People sorted by age (ascending):");
        for (Person person : people) {
          System.out.println(person);
        }
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
```

```
    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

// comparator class that compare age
class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}
```

generic

# Implementations

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces slide. There are many types of implementations but we focus on general purpose implementations only in this unit. These are the most commonly used implementations, designed for everyday use.

**General-purpose Implementations**

| Interfaces | Hash table Implementations | Resizable array Implementations | Tree Implementations | Linked list Implementations | Hash table + Linked list Implementations |
|---|---|---|---|---|---|
| Set | HashSet | | TreeSet | | LinkedHashSet |
| List | | ArrayList | | LinkedList | |
| Queue | | | | | |
| Deque | | ArrayDeque | | LinkedList | |
| Map | HashMap | | TreeMap | | LinkedHashMap |

In each one of the general purpose implementation — HashSet, ArrayList, and HashMap — is clearly the one to use for most applications, all other things being equal. Each of those interfaces has one implementation (TreeSet and TreeMap) and is listed in the Set and the Map rows. There are two general-purpose Queue implementations — LinkedList, which is also a List implementation, and PriorityQueue, which is omitted from the table. These two implementations provide very different semantics: LinkedList provides FIFO semantics, while PriorityQueue orders its elements according to their values.

Each of the general-purpose implementations provides all optional operations contained in its interface. All permit null elements, keys, and values. None are synchronised (thread-safe). All have fail-fast iterators, which detect illegal concurrent modification during iteration and fail quickly and cleanly rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. All are Serializable and all support a public clone method.

## Set Implementations

There are three general-purpose Set implementations — HashSet, TreeSet, and LinkedHashSet. Which of these three to use is generally straightforward. HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the SortedSet interface, or if value-ordered iteration is required, use TreeSet; otherwise, use HashSet. It's a fair bet that you'll end up using HashSet most of the time.

## List Implementations

There are two general-purpose List implementations — ArrayList and LinkedList. Most of the time, you'll probably use ArrayList, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the List, and it can take advantage of

System.arraycopy when it has to move multiple elements at the same time. Think of ArrayList as Vector without the synchronization overhead.

## Map Implementations

The three general-purpose Map implementations are HashMap, TreeMap and LinkedHashMap. If you need SortedMap operations or key-ordered Collection-view iteration, use TreeMap; if you want maximum speed and don't care about iteration order, use HashMap; if you want near-HashMap performance and insertion-order iteration, use LinkedHashMap. In this respect, the situation for Map is analogous to Set. Likewise, everything else in the Set Implementations section also applies to Map implementations.

## Summary of all three interfaces

| Feature | List | Map | Set |
|---|---|---|---|
| Order | Elements maintain the order of insertion. | Elements do not have a defined order. | Elements do not have a defined order. |
| Duplicates | Allows duplicate elements. | Does not allow duplicate keys. | Does not allow duplicate elements. |
| Key-Value Pairs | No key-value pairs. | Uses key-value pairs to store data. | No key-value pairs. |
| Null Values | Allows null elements (both value and index). | Allows a single null key and any number of null values. | Allows at most one null element. |
| Common Implementations | ArrayList, LinkedList, Vector | HashMap, TreeMap, LinkedHashMap | HashSet, LinkedHashSet, TreeSet |
| Use Cases | Ordered collections (sequences). | Storing key-value relationships. | Unique elements (removing duplicates). |
| Access by Index | Yes, elements can be accessed by index. | No, elements cannot be accessed by index. | No, elements cannot be accessed by index. |
| Iteration | Can iterate through elements in order. | Can iterate through key-value pairs. | Can iterate through elements, but order is not guaranteed. |
| Searching | Can search for elements by index or value. | Can search for values using keys. | Can check if a specific element exists. |