Collection Framework Overview

The CF in java provides set of interfaces and classes to manage the group of objects.

It has readymade implementations of data structures.

* Framework: A predefined structure which provides reusable designs.
* Iterable Interface: Root interface which allows traversing through elements using iterators.
* Collection Interface: Extends the iterable. It defines the common methods like adding, removing and checking the size of the collection.

[Collection (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collection.html)

* List Interface

Extends the collection interface and represents an ordered collection of elements.

* Allows the duplicate elements
* Maintains the insertion order
* List Implementations
* ArrayList
* LinkedList
* Vector

1. ArrayList

Characteristics:

* Uses dynamic array internally
* It provides faster random access using indices
* Not synchronized, so it’s not thread-safe
* package arrayList;  
    
  import java.util.ArrayList;  
    
  public class ArrayListExample {  
   public static void main(String[] args) {  
   ArrayList<String> names = new ArrayList<>();  
    
   // Add elements  
   names.add("Ramesh");  
   names.add("Krishna");  
   names.add("Govinda");  
   names.add("Gopal");  
    
   // Access element  
   System.*out*.println("First Name: "+ names.get(0));  
    
    
   //Iterating the elements  
   for(int i=0; i < names.size();i++){  
   System.*out*.println(names.get(i));  
   }  
    
   //Removing an element  
   names.remove("Ramesh");  
    
   System.*out*.println("After removing an element: "+ names);  
   }  
  }

[ArrayList (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html#add(E))

Methods:

* add() : adds an element to the list
* get(int index) : retrieves the element at that index
* remove(int index)/ remove(object o): Removes an element by index or object
* size(): returns the size of list.
* Contains(Object o): To check if a list contains the specified element
* Clear(): Removes all elements.
* [**set**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html#set(int,E))(int index, [**E**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html) element) : Replaces the element at the specified position in this list with the specified element.

1. LinkedList  
   Characteristics:

- Internally uses doubly linked list structure

- Efficient for insertions and deletions

[LinkedList (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html)  
  
package linkedListExample;  
  
import java.util.LinkedList;  
  
public class LinkedListExample {  
 public static void main(String[] args) {  
 LinkedList<String> tasks = new LinkedList<>();  
  
 // add elements  
 tasks.add("Read");  
 tasks.add("Write");  
 tasks.add("Exercise");  
  
 //Add elements at specified positions  
 tasks.addFirst("Wake Up");  
 tasks.addLast("Sleep");  
  
 //To access the first task  
 System.*out*.println("First Task: "+ tasks.getFirst());  
  
 //Iterate elements  
 for(String task: tasks){  
 System.*out*.println(task);  
 }  
  
 //to remove elements  
 tasks.removeFirst();  
 tasks.removeLast();  
  
 System.*out*.println("After removal:"+ tasks);  
 }  
}

Methods of LinkedList:

1. addFirst(): add element at the beginning
2. addLast() : add elelement at last
3. getFirst()/getLast(): retrieves the first or last element
4. add(int index, E element) : Insert an element at specified position.
5. Vector:

Characteristics:

1. Similar to arraylist, but synchronized(thread-safe)
2. Slower performance compared to arraylist due synchronization.
3. It is automatically resizable.
4. package vectorExample;  
     
   import java.util.Vector;  
     
   public class VectorExample {  
    public static void main(String[] args) {  
    Vector<Integer> numbers = new Vector<>();  
    // add elements  
    numbers.add(10);  
    numbers.add(20);  
    numbers.add(30);  
     
    // to access elements  
    System.*out*.println("First Element: "+ numbers.get(0));  
     
    // to iterate the elements  
    for (Integer number: numbers){  
    System.*out*.println(number);  
    }  
     
    // Remove an element  
    numbers.remove((Integer)20); // Explicitly Type-casting   
    System.*out*.println("after removal:" + numbers);  
    }  
   }

[Typecasting in Java - GeeksforGeeks](https://www.geeksforgeeks.org/typecasting-in-java/)

Methods:

size() : returns the size of the vector

capacity() : Returns the capacity of the vector

isEmpty() :Checks if the vector is empty

Stack:

Package: java.util.stack

Features:

* A subclass of vector
* Follows LIFO (Last In First Out) principle

Methods:

Push(): adds an element to the top

Pop() : Removes and returns the top element

Peek(): Returns the top element without removing it

Empty(): Checks the stack is empty.

[Stack (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Stack.html)

When to use: When we need LIFO behaviour(eg. Undo operatios, browser history)

package stackEx;  
  
import java.util.Stack;  
  
public class StackExample {  
 public static void main(String[] args) {  
 Stack<String> books = new Stack<>();  
  
 //Adding few books  
 books.push("Black Book JAVA");  
 books.push("The Basics of SQL");  
 books.push("C# Basics");  
  
 //To get current book (Top of stack)  
 System.*out*.println("Currently Reading: "+ books.peek());  
  
 //After finishing book(pop)  
 System.*out*.println("Finished reading: "+ books.pop());  
  
 //Displaying books  
 System.*out*.println("Books in Stack: "+ books);  
  
 }  
}

ArrayList Vs Vector

package arrayListVsVector;  
  
import java.util.ArrayList;  
import java.util.Vector;  
  
public class AVExample {  
 public static void main(String[] args) {  
 Vector<Integer> vector = new Vector<>();  
 ArrayList<Integer> arrayList = new ArrayList<>();  
  
 long startTime, endTime;  
  
 //Measure Vector performance  
 startTime=System.*nanoTime*();  
 for (int i =0; i < 100000; i++){  
 vector.add(i);  
 }  
 endTime=System.*nanoTime*();  
 System.*out*.println("Time taken by vector: "+ (endTime-startTime)+ "ns");  
  
 System.*out*.println("================================================================");  
  
 //Measuring the arraylist performance  
 startTime=System.*nanoTime*();  
 for (int i = 0; i < 100000; i++){  
 arrayList.add(i);  
 }  
 endTime=System.*nanoTime*();  
 System.*out*.println("Time taken by ArrayList:"+ (endTime-startTime)+ "ns");  
 }  
}

O/P:

Time taken by vector: 15643500ns

========================================================

Time taken by ArrayList:10180800ns

To make arrayList synchronized:

List<Integer> synchronizedList = collections.synchronizedList(new ArrayList<>());

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Arraylist | LinkedList | Vector | Stack |
| Structure | Dynamic array | Doubly linkedlist | Dynamic Array | Dynamic array |
| Thread safety | No | No | Yes | Yes |
| When to use | Frequent access | Frequent insert/ remove | Thread-safety needed | LIFO required |

SET Interface:

* Stores unique elements and do not allow duplicates
* Two implementations: HashSet, LinkedHashSet

1. hashSet

it implements the set interface

it stores elements in unordered way

best for quick adding, removing and contains operations.

package setExample;  
  
import java.util.HashSet;  
  
public class HashSetExample {  
 public static void main(String[] args) {  
 // Creating the hashset  
 HashSet<String> fruits = new HashSet<>();  
  
 //Adding elements  
 fruits.add("Apple");  
 fruits.add("banana");  
 fruits.add("Grapes");  
 fruits.add("Orange");  
  
 //Displaying the hashset  
 System.*out*.println("HashSet: "+ fruits);  
  
 //Checking the element exist or not  
 System.*out*.println("Does hashset contains Grapes? :"+ fruits.contains("Grapes"));  
  
 //Removing the element  
 fruits.remove("Grapes");  
 System.*out*.println("After removal: "+ fruits);  
  
 //To check the size of set  
 System.*out*.println("Size of hashset: "+ fruits.size());  
  
 //To clear all elements  
 fruits.clear();  
 System.*out*.println("Is hashset empty? "+ fruits.isEmpty());  
 }  
}

O/P:

HashSet: [banana, Apple, Grapes, Orange]

Does hashset contains Grapes? :true

After removal: [banana, Apple, Orange]

Size of hashset: 3

Is hashset empty? True

[HashSet (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html)

LinkedHashSet:

Similar to hashset but maintains the doubly linked list

Characteristics:

* Maintains the insertion order
* Slightly slower than hashset because it maintains the order.

package linkedHashsetEx;  
  
import java.util.LinkedHashSet;  
  
public class LinkedHashSetExample {  
 public static void main(String[] args) {  
 LinkedHashSet<String> cities = new LinkedHashSet<>();  
 //Adding elements to the linkedHashSet  
 cities.add("Sambhajinagar");  
 cities.add("Pune");  
 cities.add("Nagpur");  
 cities.add("Mumbai");  
  
 //Displaying the LHS  
 System.*out*.println("LHS : "+ cities);  
  
 // To check LHS contains an element or not  
 System.*out*.println("Does it contain Mumbai? "+ cities.contains("Mumbai"));  
 }  
}

O/P ->

LHS : [Sambhajinagar, Pune, Nagpur, Mumbai]

Does it contain Mumbai? true