Abstract Data Type(ADT)

Abstract Data Type or ADT

**Abstract Data Types** are a way of classifying data structures based on how they are used and the behaviors they provide. They do not specify how the data structure must be implemented or laid out in memory, but simply provide a standard interface and set of behaviors.

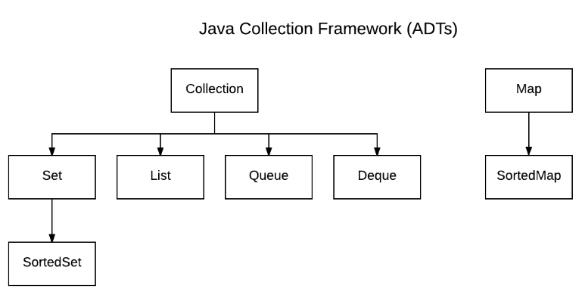
An **ADT** is a data type, where only behavior is defined but not implementation. Java provides ADTs through its Collection Framework.

A **Data Structure** is a way of storing and organizing data in a computer so that it can be accessed efficiently. Data Structures are used to implement ADTs.

Collection

A **Collection** is a group of individual objects represented as a single unit. Java provides the Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are two main root interfaces of Java collection classes.



Advantages of Collection Framework

1. Consistent API
   * The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector, etc.) which implements, these interfaces have some common set of methods.
2. Reduces programming effort
   * The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.
3. Increases program speed and quality
   * Increases performance by providing high-performance implementations of useful data structures and algorithms

Java Interfaces

[Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html): Root interface with basic methods like add(), remove(), contains(), isEmpty(), addAll(), etc.

[Set](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html): Does not allow duplicates. Example implementations of Set interface are [HashSet](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html) (data structure – hash table) and [TreeSet](https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html) (data structure – binary search tree).

[List](https://docs.oracle.com/javase/7/docs/api/java/util/List.html): Can contain duplicates and elements are ordered. Example implementations are [ArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) (data structure – dynamic array) , [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html) (data structure – linked list), and [Stack](https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html).

[Queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html): Typically order elements in FIFO (first-in-first-out) order. Example: waiting in line at the grocery store. One exception to this ordering is the [PriorityQueue](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html). A priority queue assigns a priority to its elements. Elements with the highest priority move to the front of the line. Elements with same priority following FIFO ordering.

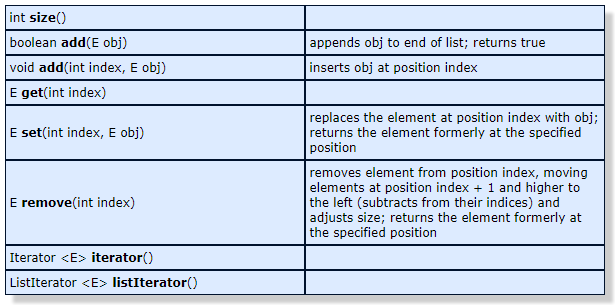
[Deque](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html): Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html): Contains key value pairs. Does not allow duplicates. Example implementations are [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) and [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html).

List Interface

To demonstrate how interfaces are used in Java let’s look at how the List interface is used to implement the classes [ArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) and [LinkedList](https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html).

Below is a parital list of methods declared (not implemented) by the List interface.



Both the ArrayList and LinkedList classes implement the List interface:

class ArrayList<E> implements List<E>  
class LinkedList<E> implements List<E>

A class that implements an interface is **required** to implement all methods declared in that interface, therefore both the ArrayList class and the LinkedList class must implement all of the List interface methods show in the table above and these methods must exhibit the prescribed behavior.

For example, both classes are required to implement the add(E obj) method and this method must append the element obj to the end of their list and return true if it was successful.

Below is a sample program that demonstrates the use of these List methods in both the ArrayList class and the LinkedList class.

import java.util.\*;

public class ListDemo {  
 public static void main(String[] args) {  
 List<String> arrayList = new ArrayList<>();  
 List<String> linkedList = new LinkedList<>();  
 //ArrayList<String> arrayList = new ArrayList<>();  
 //LinkedList<String> linkedList = new LinkedList<>();

// append items to end of list  
 arrayList.add("red");  
 linkedList.add("red");  
 arrayList.add("white");  
 linkedList.add("white");  
 arrayList.add("blue");  
 linkedList.add("blue");  
 printList(arrayList);  
 printList(linkedList);  
 System.out.println();  
  
 // insert items within list  
 arrayList.add(2, "black");  
 linkedList.add(2, "black");  
 printList(arrayList);  
 printList(linkedList);  
 System.out.println();

// edit items  
 arrayList.set(2, "flag");  
 linkedList.set(2, "flag");  
 printList(arrayList);  
 printList(linkedList);  
 System.out.println();

// remove items  
 arrayList.remove(2);  
 linkedList.remove(2);  
 printList(arrayList);  
 printList(linkedList);  
 System.out.println();  
 }  
 public static void printList(List list) {  
 for(int i=0; i < list.size(); i++)  
 {  
 System.out.print(list.get(i) + " ");  
 }  
 System.out.println();  
 }  
}

Sample Program Output

red white blue

red white blue

red white black blue

red white black blue

red white flag blue

red white flag blue

red white blue

red white blue