Java - Data Structures

These data structures consist of the following interface and classes −

* Enumeration
* BitSet
* Vector
* Stack
* Dictionary
* Hashtable
* Properties

## The Enumeration

The Enumeration interface defines a means to retrieve successive elements from a data structure.

For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

## The BitSet

The BitSet class implements a group of bits or flags that can be set and cleared individually.

## The Vector

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a Vector object can be accessed via an index into the vector.

## The Stack

The Stack class implements a last-in-first-out (LIFO) stack of elements.

## The Dictionary

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.

## The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

## The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

# Java - 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, i.e., Classes** − 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.

## The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface −

|  |  |
| --- | --- |
| **Sr.No.** | **Interface & Description** |
| 1 | [**The Collection Interface**](https://www.tutorialspoint.com/java/java_collection_interface.htm)  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | [**The List Interface**](https://www.tutorialspoint.com/java/java_list_interface.htm)  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | [**The Set**](https://www.tutorialspoint.com/java/java_set_interface.htm)  This extends Collection to handle sets, which must contain unique elements. |
| 4 | [**The SortedSet**](https://www.tutorialspoint.com/java/java_sortedset_interface.htm)  This extends Set to handle sorted sets. |
| 5 | [**The Map**](https://www.tutorialspoint.com/java/java_map_interface.htm)  This maps unique keys to values. |
| 6 | [**The Map.Entry**](https://www.tutorialspoint.com/java/java_mapentry_interface.htm)  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | [**The SortedMap**](https://www.tutorialspoint.com/java/java_sortedmap_interface.htm)  This extends Map so that the keys are maintained in an ascending order. |
| 8 | [**The Enumeration**](https://www.tutorialspoint.com/java/java_enumeration_interface.htm)  This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

[**The Collection Interface**](https://www.tutorialspoint.com/java/java_collection_interface.htm)

## Example

Following is an example to explain few methods from various class implementations of the above collection methods :

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

// ArrayList

List a1 = new ArrayList();

a1.add("Zara");

a1.add("Mahnaz");

a1.add("Ayan");

System.out.println(" ArrayList Elements");

System.out.print("\t" + a1);

// LinkedList

List l1 = new LinkedList();

l1.add("Zara");

l1.add("Mahnaz");

l1.add("Ayan");

System.out.println();

System.out.println(" LinkedList Elements");

System.out.print("\t" + l1);

// HashSet

Set s1 = new HashSet();

s1.add("Zara");

s1.add("Mahnaz");

s1.add("Ayan");

System.out.println();

System.out.println(" Set Elements");

System.out.print("\t" + s1);

// HashMap

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

This will produce the following result −

## Output

ArrayList Elements

[Zara, Mahnaz, Ayan]

LinkedList Elements

[Zara, Mahnaz, Ayan]

Set Elements

[Ayan, Zara, Mahnaz]

Map Elements

{Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

[**The List Interface**](https://www.tutorialspoint.com/java/java_list_interface.htm)

* Elements can be inserted or accessed by their position in the list, using a zero-based index.
* A list may **contain duplicate elements.**

## Example

The above interface has been implemented in various classes like ArrayList or LinkedList, etc. Following is the example to explain few methods from various class implementation of the above collection methods :

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

List a1 = new ArrayList();

a1.add("Zara");

a1.add("Mahnaz");

a1.add("Ayan");

System.out.println(" ArrayList Elements");

System.out.print("\t" + a1);

List l1 = new LinkedList();

l1.add("Zara");

l1.add("Mahnaz");

l1.add("Ayan");

System.out.println();

System.out.println(" LinkedList Elements");

System.out.print("\t" + l1);

}

}

This will produce the following result −

## Output

ArrayList Elements

[Zara, Mahnaz, Ayan]

LinkedList Elements

[Zara, Mahnaz, Ayan]

[**The Set**](https://www.tutorialspoint.com/java/java_set_interface.htm)

A Set is a Collection that cannot **contain duplicate elements**.

## Example

Set has its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following is an example to explain Set functionality

import java.util.\*;

public class SetDemo {

public static void main(String args[]) {

int count[] = {34, 22,10,60,30,22};

Set<Integer> set = new HashSet<Integer>();

try {

for(int i = 0; i < 5; i++) {

set.add(count[i]);

}

System.out.println(set);

TreeSet sortedSet = new TreeSet<Integer>(set);

System.out.println("The sorted list is:");

System.out.println(sortedSet);

System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());

System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());

}

catch(Exception e) {}

}

}

This will produce the following result −

## Output

[34, 22, 10, 60, 30]

The sorted list is:

[10, 22, 30, 34, 60]

The First element of the set is: 10

The last element of the set is: 60

[**The Map**](https://www.tutorialspoint.com/java/java_map_interface.htm)

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

## The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | **AbstractCollection**  Implements most of the Collection interface. |
| 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | [**LinkedList**](https://www.tutorialspoint.com/java/java_linkedlist_class.htm)  Implements a linked list by extending AbstractSequentialList. |
| 5 | [**ArrayList**](https://www.tutorialspoint.com/java/java_arraylist_class.htm)  Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**  Extends AbstractCollection and implements most of the Set interface. |
| 7 | [**HashSet**](https://www.tutorialspoint.com/java/java_hashset_class.htm)  Extends AbstractSet for use with a hash table. |
| 8 | [**LinkedHashSet**](https://www.tutorialspoint.com/java/java_linkedhashset_class.htm)  Extends HashSet to allow insertion-order iterations. |
| 9 | [**TreeSet**](https://www.tutorialspoint.com/java/java_treeset_class.htm)  Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**  Implements most of the Map interface. |
| 11 | [**HashMap**](https://www.tutorialspoint.com/java/java_hashmap_class.htm)  Extends AbstractMap to use a hash table. |
| 12 | [**TreeMap**](https://www.tutorialspoint.com/java/java_treemap_class.htm)  Extends AbstractMap to use a tree. |
| 13 | [**WeakHashMap**](https://www.tutorialspoint.com/java/java_weakhashmap_class.htm)  Extends AbstractMap to use a hash table with weak keys. |
| 14 | [**LinkedHashMap**](https://www.tutorialspoint.com/java/java_linkedhashmap_class.htm)  Extends HashMap to allow insertion-order iterations. |
| 15 | [**IdentityHashMap**](https://www.tutorialspoint.com/java/java_identityhashmap_class.htm)  Extends AbstractMap and uses reference equality when comparing documents. |

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util :

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | [**Vector**](https://www.tutorialspoint.com/java/java_vector_class.htm)  This implements a dynamic array. It is similar to ArrayList, but with some differences. |
| 2 | [**Stack**](https://www.tutorialspoint.com/java/java_stack_class.htm)  Stack is a subclass of Vector that implements a standard last-in, first-out stack. |
| 3 | [**Dictionary**](https://www.tutorialspoint.com/java/java_dictionary_class.htm)  Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. |
| 4 | [**Hashtable**](https://www.tutorialspoint.com/java/java_hashtable_class.htm)  Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. |
| 5 | [**Properties**](https://www.tutorialspoint.com/java/java_properties_class.htm)  Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. |
| 6 | [**BitSet**](https://www.tutorialspoint.com/java/java_bitset_class.htm)  A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. |

## The Collection Algorithms

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

## Example

Following is an example, which demonstrates various algorithms.

import java.util.\*;

public class AlgorithmsDemo {

public static void main(String args[]) {

// Create and initialize linked list

LinkedList ll = new LinkedList();

ll.add(new Integer(-8));

ll.add(new Integer(20));

ll.add(new Integer(-20));

ll.add(new Integer(8));

// Create a reverse order comparator

Comparator r = Collections.reverseOrder();

// Sort list by using the comparator

Collections.sort(ll, r);

// Get iterator

Iterator li = ll.iterator();

System.out.print("List sorted in reverse: ");

while(li.hasNext()) {

System.out.print(li.next() + " ");

}

System.out.println();

Collections.shuffle(ll);

// display randomized list

li = ll.iterator();

System.out.print("List shuffled: ");

while(li.hasNext()) {

System.out.print(li.next() + " ");

}

System.out.println();

System.out.println("Minimum: " + Collections.min(ll));

System.out.println("Maximum: " + Collections.max(ll));

}

}

This will produce the following result −

## Output

List sorted in reverse: 20 8 -8 -20

List shuffled: 20 -20 8 -8

Minimum: -20

Maximum: 20

# Java - Generics

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Following example illustrates how we can print an array of different type using a single Generic method −

[Live Demo](http://tpcg.io/9xT7Ev)

public class GenericMethodTest {

// generic method printArray

public static < E > void printArray( E[] inputArray ) {

// Display array elements

for(E element : inputArray) {

System.out.printf("%s ", element);

}

System.out.println();

}

public static void main(String args[]) {

// Create arrays of Integer, Double and Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");

printArray(intArray); // pass an Integer array

System.out.println("\nArray doubleArray contains:");

printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");

printArray(charArray); // pass a Character array

}

}

## Bounded Type Parameters

public class MaximumTest {

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z) {

T max = x; // assume x is initially the largest

if(y.compareTo(max) > 0) {

max = y; // y is the largest so far

}

if(z.compareTo(max) > 0) {

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main(String args[]) {

System.out.printf("Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ));

System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

System.out.printf("Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum("pear", "apple", "orange"));

}

}

## Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

This will produce the following result −

### Output

Integer Value :10

String Value :Hello World