Introduction

Arrays are used to store group of objects. Let us take an example where we want to store 100 objects of `Employee` class into an array. For this purpose, we need to create an array of `Employee` type as:

```
Employee arr[] = new Employee[100];
```

So, here we are familiar with how to store a group of objects into an array and retrieve them again easily. But there are also certain limitations in this mechanism. They are as follows:

- We cannot store different class objects into the same array. The reason is that an array can store only one data type of elements.
- Adding the objects at the end of an array is easy. But, inserting and deleting the elements in the middle of the array is difficult. In this case, we have to re-arrange all the elements of the array.
- Retrieving the elements from an array is easy but after retrieving the elements, if we want to process them, then there are no methods available.

Due to these problems, programmers want a better mechanism to store a group of objects. The alternative is using an object to store a group of other objects. It means that we can use a class object as an array. Such an object is called "Collection object" or "Container object".

Collection Objects

A collection in java is an object that can hold multiple objects (like an array). A collection object has a class called as "collection class". All these collection classes are available in the package `java.util`(util stands for Utility). A group of collection classes is called as `‘‘Collection Framework’’`.

A collection framework is a common architecture for representing and manipulating all the collections. This architecture has a set of interfaces on the top and implementing classes down the hierarchy.

All the collection classes in `java.util` package are the implementation classes of different interfaces as shown below.

Interface	Implementation classes
Set<T>	HastSet<T>
	TreeSet<T>
	Stack<T>
List<T>	LinkedList<T>
	ArrayList<T>
	Vector<T>
Queue<T>	LinkedList<T>
Map<K, V>	HashMap<K, V>
	HasHable<K, V>

Collection is the root interface in the collection hierarchy. The items in the collection is refereed to as elements. `Collection` interface extends another interface called `Iterable`.

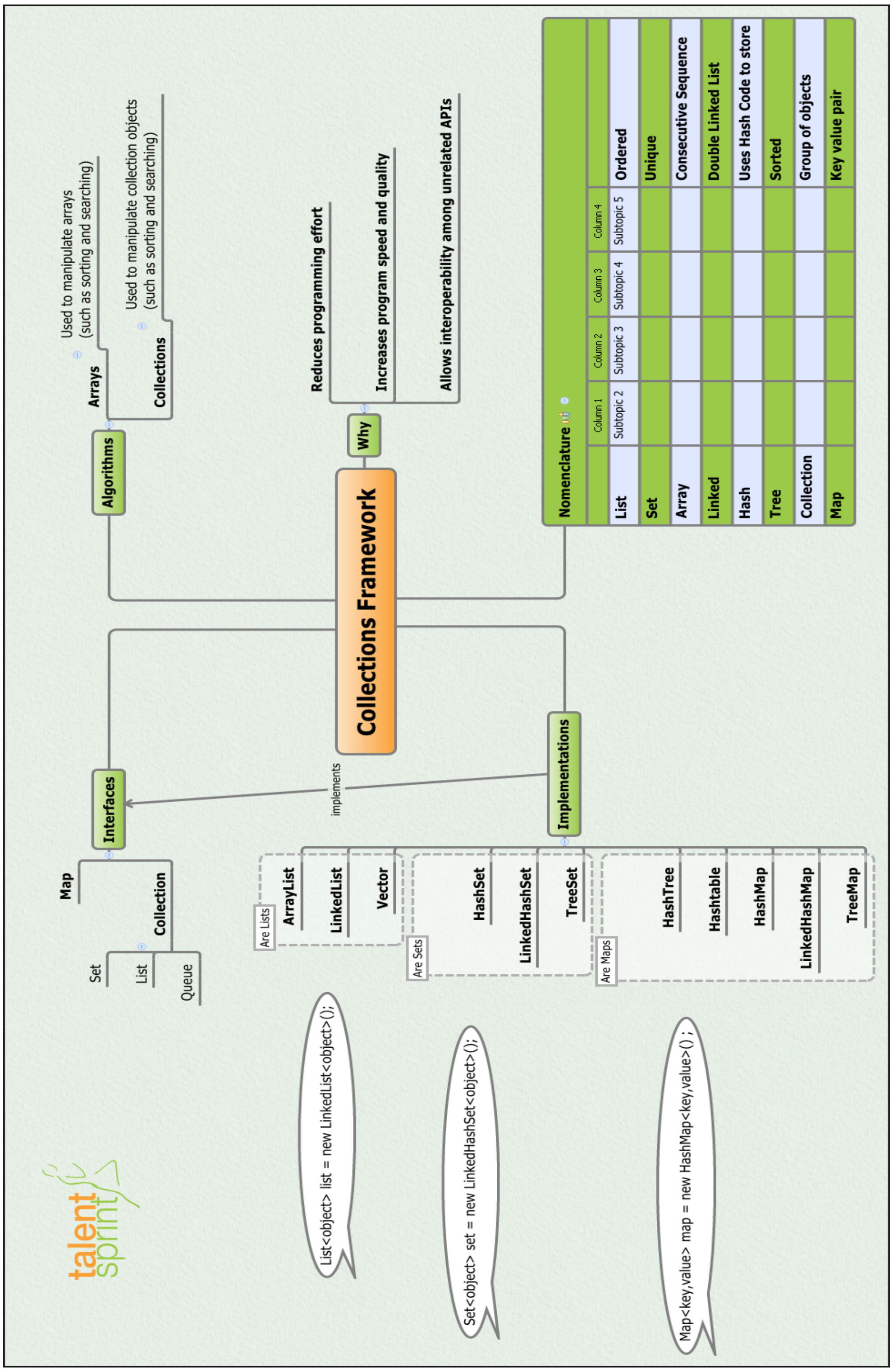
Set represents a group of elements arranged just like an array. The Set will grow dynamically when the elements are stored into it. A set will not allow duplicate elements.

Lists are like sets. They store a group of elements. But lists allow duplicate values to be stored. The subclasses of List are ordered collection of objects. List is also called sequence.

Queue represents arrangement of elements in FIFO (First In First Out) order.

Map store elements in the form of key and value pairs. If the key is provided then its corresponding value can be obtained. The keys should have unique values.

Note
We cannot store primitive data types in the collection objects. We can store only objects.


Collections Framework
Algorithms
Arrays
Used to manipulate arrays (such as sorting and searching)
Collections
Used to manipulate collection objects (such as sorting and searching)
Why
Reduces programming effort
Increases program speed and quality
Allows interoperability among unrelated APIs
Interfaces
Map
Collection
Set
List
Queue
implements
Implementations
Are Lists
ArrayList
LinkedList
Vector
Are Sets
HashSet
LinkedHashSet
TreeSet
Are Maps
HashTree
Hashtable
HashMap
LinkedHashMap
TreeMap
List<object> list = new LinkedList<object>();
Set<object> set = new LinkedHashSet<object>();
Map<key,value> map = new HashMap<key,value>() ;
talent sprint

Nomenclature

	Column 1	Column 2	Column 3	Column 4	
List	Subtopic 2	Subtopic 3	Subtopic 4	Subtopic 5	Ordered
Set					Unique
Array					Consecutive Sequence
Linked					Double Linked List
Hash					Uses Hash Code to store
Tree					Sorted
Collection					Group of objects
Map					Key value pair

Retrieving Elements from Collections

Following are the four ways to retrieve any element from a collection object.

for-each loop

`for-each` loop is like `for` loop which repeatedly executes a group of statements for each element of the collection.

```
for ( variable-name : collection-object) {
    statement(s);
}
```

Iterator

`Iterator` is an interface that contain methods to retrieve the elements one by one from a collection object. It has three methods.

boolean hasNext() returns true if the iterator has more elements.

element next() returns the next element in the list.

void remove() removes the last element from the collection returned by the iterator.

ListIterator

`ListIterator` is an interface that contain methods to retrieve the elements from a collection object, both in forward and reverse directions. It has the following methods.

boolean hasNext() returns true if the `ListIterator` has more elements when traversing the list in the forward direction.

boolean hasPrevious() returns true if the `ListIterator` has more elements when traversing the list in the reverse direction.

element next() returns the next element in the list.

element previous() returns the previous element in the list.

void remove() removes the last element from the collection returned by the iterator.

Enumeration

This interface is useful to retrieve one by one element like the iterator. It has two methods,

boolean hasMoreElements() checks if the `Enumeration` has any more elements or not.

element nextElement() returns the next element that is available in `Enumeration`.

HashSet Class

A `HashSet` represents a set of elements (objects). It does not guarantee the order of elements. Also it does not allow the duplicate elements to be stored.

We can write the `HashSet` class as,

class HashSet<T>

Here, <T> represents the generic type parameter. It represents which type of elements are being to be stored into the `HashSet`.

For example,

HashSet<String> hs = **new** HashSet<String>();

HashSet Class Methods

`HashSet` class provides the following methods

boolean add(obj) adds an element `obj` to the `HashSet`. It returns true if the element is added to `HashSet` else it returns false. If the same element is already available in the `HashSet`, then the present element is not added.

boolean remove(obj) removes the element (obj) from the `HashSet`, if it is present. It returns true if the element is removed successfully otherwise false.

void clear() removes all the elements from the `HashSet`.

boolean contains(obj) retruns true if the `HashSet` contains the specified element `obj`.

boolean isEmpty() returns true if the `HashSet` contains no elements.

int size() returns the number of elements present in the `HashSet`.

COPYRIGHT ©TALENTSPRINT, 2015. ALL RIGHTS RESERVED.

Example

Below example shows how to compare two sets, and retain the values which are common on both set objects. By calling `retainAll()` method we can retrieve common objects.

```
import java.util.HashSet;
public class CompareHashSets {
    public static void main(String args[]) {
        HashSet<String> fruitSet = new HashSet<>();
        //add elements to HashSet called 'fruitSet'
         fruitSet.add(''apple'');
         fruitSet.add(''banana'');
         fruitSet.add(''lemons'');
         fruitSet.add(''oranges'');
         fruitSet.add(''mango'');
        System.out.println(''Spring Season Fruits: '');
        System.out.println(fruitSet);
        // add elements to another HashSet called 'subSet'
        HashSet<String> subSet = new HashSet<String>();
        subSet.add(''banana'');
        subSet.add('' cherries '');
        subSet.add(''grapes'');
        subSet.add(''oranges'');
        System.out.println(''Summer Season Fruits: '');
        System.out.println(subSet);
         fruitSet.retainAll(subSet);
        System.out.println('' Fruits in All Seasons:'');
        System.out.println(fruitSet);
    }
}
```

TreeSet Class

`TreeSet` contains unique elements like HashSet. The `TreeSet` class implements `NavigableSet` interface that extends the `SortedSet` interface. `TreeSet` maintains ascending order.

Example

The easiest way to find duplicate entries from the given array is, create `TreeSet` object and add array entries to the `TreeSet`. Since the set does not support duplicate entries, you can easily findout duplicate entries. Below example add each element to the set, and checks the returns status.

```
import java.util.TreeSet;
public class TreesetDuplicateEntry {
    public static void main(String args[]) {
        String[] strArr = {"one", "two",
        "three", "four", "four", "five"};
        TreeSet<String> unique = new TreeSet<>();
        for (String str : strArr) {
            if (!unique.add(str)) {
                System.out.println ("Duplicate Entry is: " + str);
            }
        }
    }
}
```

ArrayList Class

An `ArrayList` is like an array, which can grow in memory dynamically. It means that when we store elements into the `ArrayList`, depending on the number of elements, the memory is dynamically allocated and re-allocated to accommodate all the elements. `ArrayList` increases its size every time by 50 percent (half).

`ArrayList` is not synchronized. But, we can use `synchronizedList()` method to synchronize the `ArrayList` as,

```
List list = Collections.synchronizedList(new ArrayList());
```

The `ArrayList` class can be written as,

```
class ArrayList<E>
```

For example,

```
ArrayList<String> = new ArrayList<String>();
```

ArrayList Class Methods

boolean add(element obj) appends the specified element to the end of the `ArrayList`. If the element is added successfully then the preceding method returns true.

void add(int position, element obj) inserts the specified element at the specified position in the `ArrayList`.

void clear() removes all the elements from the `ArrayList`.

boolean contains(Object obj) returns true if the `ArrayList` contains the specified element.

element get(int position) returns the element available at the specified position in this `ArrayList`.

int indexOf(Object obj) returns the index of the first occurrence of the specified element in the `ArrayList`, or -1 if the `ArrayList` does not contain the element.

boolean isEmpty() returns true if this list contains no elements.

int lastIndexOf(Object obj) returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

element remove(int position) removes the element at the specified position in the `ArrayList`.

boolean remove(Object obj) removes the first occurrence of the specified element from the `ArrayList`, if it is present.

element set(int index, element obj) replaces the element at the specified position in the `ArrayList` with the specified element `obj`.

int size() returns the number of elements present in the `ArrayList`.

Object[] toArray() returns an array containing all of the elements in this list in proper sequence (from first to last element).

Example

This example gives how to shuffle elements in the ArrayList. By calling `Collections.shuffle()` method you can shuffle the content of the ArrayList. Everytime you call `shuffle()` method, it generates different order of output.

```
import java.util.ArrayList;
import java.util.Collections;
public class ShuffleArrayList {
    public static void main(String a[]) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Java");
        list.add("Cric");
        list.add("Play");
        list.add("Watch");
        list.add("Glass");
        list.add("Movie");
        list.add("Girl");

        System.out.println("List before shuffle operation: ");
        for (String element : list) {
            System.out.print(element + " ");
        }
        Collections.shuffle(list);
        System.out.println("Results after shuffle operation:");
        for (String element : list) {
            System.out.print(element + " ");
        }

        Collections.shuffle(list);
        System.out.println("Results after shuffle operation:");
        for (String element: list) {
            System.out.print(element + " ");
        }
    }
}
```

Vector Class

A `Vector` also stores elements(objects) similar to `ArrayList`, but `Vector` is synchronized. `Vector` increases its size every time by doubling it.

We can write a `Vector` class as,

```
class Vector<E>
```

Here, `E` represents the type of elements stored into the `Vector`.

For example,

```
Vector<Float> v = new Vector<Float>();
```

The preceding statement creates a `Vector` object `v` which can be used to store `Float` type objects.

Vector Class Methods

boolean add(element obj) appends the specified element to the end of the `Vector`. If the element is added successfully then the preceding method returns true.

void add(int position, element obj) method inserts the specified element at the specified position in the `Vector`.

void clear() removes all of the elements from the `Vector`.

boolean contains(Object obj) returns true if the `Vector` contains the specified element.

element get(int position) returns the element available at the specified position in the `Vector`.

int indexOf(Object obj) returns the index of the first occurrence of the specified element in the `Vector`, or -1 if the `Vector` does not contain the element.

boolean isEmpty() returns true if this list contains no elements.

int lastIndexOf(Object obj) returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

element remove(int position) removes the element at the specified position in the `Vector`.

boolean remove(Object obj) removes the first occurrence of the specified element from the `Vector`, if it is present.

element set(int index, element obj) replaces the element at the specified position in the `Vector` with the specified element `obj`.

int size() returns the number of elements present in the `Vector`.

Object[]toArray() returns an array containing all of the elements in this list in proper sequence (from first to last element).

int capacity() returns the current capacity of the `Vector`.

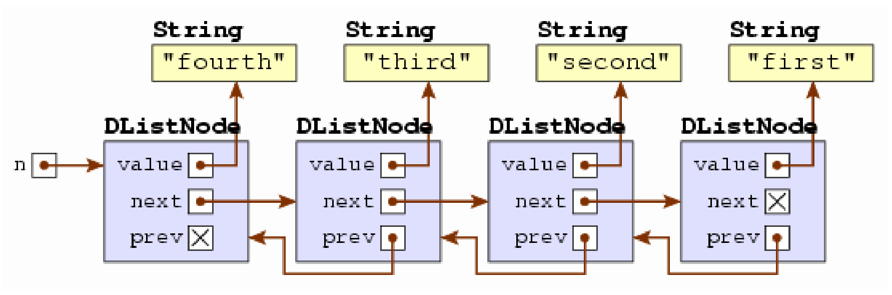
Example

Here, we can copy all elements of a vector object to an array. By passing an array object to copyInto() method, you can copy content of a vector object to an array and sort array in an alphabetical order by calling `Arrays.sort(array_name)`.

```java
import java.util.Vector;
import java.util.Arrays;
public class CopyVectorToArray {
    public static void main(String args[]) {
        Vector<String> vct = new Vector<>();
            vct.add("Python");
            vct.add("Java");
            vct.add("C");
            vct.add("Ajax");
            vct.add("SQL");
            System.out.println ("Actual vector: ");
            System.out.println (vct);
            String [] copyArr = new String[vct.size()];
            vct.copyInto(copyArr);
            System.out.println ("Ordered Copied Array: ");
            Arrays.sort(copyArr);
            for (String str : copyArr) {
                System.out.print(str + " ");
            }
            System.out.println ();
    }
}
```

LinkedList Class

A LinkedList contains a group of elements in the form of nodes. Each node will have three fields- the data field contains data and link field contain references to previous and next nodes.



LinkedList is a convenient to store data. Inserting the elements into the LinkedList and removing the elements from the LinkedList is done quickly and takes the same amount of time.

LinkedList is not synchronized. But, we can use synchronizedList() method to synchronize the LinkedList as,

```
List list = Collections.synchronizedList(new LinkedList());
```

LinkedList Class Methods

boolean add(element obj) appends the specified element to the end of this list.

void add(int position, element obj) inserts the specified element obj at the specified position in this list.

void addFirst(element obj) inserts the specified element obj at the beginning of this list.

void addLast(element obj) appends the specified element obj to the end of this list.

element removeFirst() removes and returns the first element from this list.

element removeLast() removes and returns the last element from this list.

element get(int position) returns the element at the specified position in this list.

element getFirst() returns the first element in this list.

element getLast() returns the last element in this list.

int size() returns the number of elements in this list.

Example

Below example shows how to iterate through LinkedList in reverse order. The method `descendingIterator()` returns an `Iterator` object with reverse order. By iterating through it, you can get the elements in the reverse order.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Scanner;
public class LinkedListReverseIteration {

    public static void main(String a[]) {
        Scanner sc = new Scanner(System.in);
        LinkedList<String> linkedList = new LinkedList<>();
        String choice = "";
        do {
            System.out.print("enter name to store in a list: ");
            String name = sc.next();
            linkedList.add(name);
            System.out.println();
            System.out.print("want to add one more element(Y/N)?");
            choice = sc.next();
        } while (choice.equals("y"));
        System.out.println("Original list: ");
        System.out.println(linkedList);
        System.out.println("Reversed list: ");
        Iterator<String> iterator = linkedList.descendingIterator();
        while (itr.hasNext()) {
            System.out.print(itr.next());
        }
        System.out.println();
    }
}
```

HashMap Class

HashMap is a collection that store elements in the form of key-value pairs. If key is provided later, its corresponding value can be easily retrieved from the HashMap. Keys should be unique.

HashMap allows null keys and null values to be stored.

HashMap is not synchronized. But, use synchronizedMap() to synchronize HashMap as,

Map m = Collections.synchronizedMap(**new** HashMap(...));

HashMap Class Methods

void clear() removes all of the mappings from this map.

value get(Object key) returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

boolean isEmpty() returns true if this map contains no key-value mappings.

Set<K> keySet() returns a Set view of the keys contained in this map.

value put(key, value) associates the specified value with the specified key in this map.

value remove(Object key) removes the mapping for the specified key from this map if present.

int size() returns the number of key-value mappings in this map.

Collection<V> values() returns a Collection view of the values contained in this map.

Example

```
import java.util.HashMap;
import java.util.Set;
public class GetHashMapkeys {
    public static void main(String a[]) {
        HashMap<Integer, String> hashMap = new HashMap<>();
        hashMap.put(12, ``one two'');
        hashMap.put(999, ``nine nine nine'');
        hashMap.put(2014, ``two zero one four'');
        System.out.println(hashMap);
        Set<Integer> keys = hashMap.keySet();
        System.out.println ("keys in HashMap: ");
        for (Integer key: keys) {
            System.out.print(key + " ");
        }
        System.out.println();
    }
}
```

HashTable Class

`HashTable` is similar to `HashMap` which can store elements in the form of key-value pairs. But `HashTable` is synchronized.

`HashTable` does not allow null keys or values.

HashTable Class Methods

void clear() clears this hashTable so that it contains no keys.

Enumeration<V> elements() returns an enumeration of the values in this hashTable.

value get(Object key) returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

boolean isEmpty() checks if this hashTable maps no keys to values.

Enumeration<K> keys() returns an enumeration of the keys in this hashTable.

Set<K> keySet() returns a Set view of the keys contained in this map.

value put(key, value) maps the specified key to the specified value in this hashTable.

int size() returns the number of keys in this hashTable.

Collection<V> values() returns a Collection view of the values contained in this map.

Example

Below example shows how to get all keys as `Enumeration` object. By calling `keys()` method, we can get all keys as Enumeration object. By using Enumeration methods like `hasMoreElements()` and `nextElement()` we can read all values from HashTable.

```
import java.util.Enumeration;
import java.util.Hashtable;
public class ReadHashtableValues {
    public static void main(String args[]) {
        Hashtable<String, String> hm = new Hashtable<>();
        hm.put("one", "January");
        hm.put("two", "Febraury");
        hm.put("three", "March");
        Enumeration<String> keys = hm.keys();
        while (keys.hasMoreElements()) {
            String key = keys.nextElement();
            System.out.println ("Value of " + key + " is: "
                                + hm.get(key));
        }
    }
}
```

StringTokenizer

`StringTokenizer` class is useful to break a string into pieces, called 'tokens'. These tokens are then stored in the `StringTokenizer` object from where they can be retrieved.

An object to `StringTokenizer` class is,

```
StringTokenizer st = new StringTokenizer(str, "delimiter");
```

In the preceding statement, the actual string `str` is broken into pieces at the positions marked by a group of characters, called 'delimiter'. For example, to break the string wherever a comma is found, can write as,

```
StringTokenizer st = new StringTokenizer(str, ",");
```

Similarly, to break the string wherever a comma or colon or both are found, we can use

```
StringTokenizer st = new StringTokenizer(str, ",:");
```

StringTokenizer Methods

int countTokens() counts and returns the number of tokens available in a `StringTokenizer` object.

boolean hasMoreTokens() checks if there are more tokens available in the `StringTokenizer` object or not.

String nextToken() returns the next token from the `StringTokenizer`.

Example

Below example shows number of token count after breaking the string by delimiter. You can get the count by using `countTokens()` method.

```
import java.util.StringTokenizer;
public class CountTokens{
    public static void main(String args[]) {
        String msg = "TalentSprint Module-4 EJD";
        StringTokenizer st = new StringTokenizer(msg, " ");
        System.out.println ("No of Tokens: " + st.countTokens());
        System.out.println ("Tokens are: ");
        while (st.hasMoreTokens()) {
            System.out.println (st.nextToken());
        }
    }
}
```