# Collections

GEN1 : int x =10, int y =20, int z =30, …..

GEN2 : Arrays

## Arrays

Student[] s = new Student[10000];

s[1] = new Student();

s[2] = new Student();

…

#### Advantage

Represent multiple values by using one variable so readability of the code will be improved

#### Disadvantages

* **Fixed size**

After creating an array we can’t increase or decrease the size based on requirement

* **Homogenous**

s[3] = new Employee(); won’t accept

Because arrays can hold homogeneous data-type only, so only Student object will be acceptable

Solution:

Object Type arrays

Object[] s = new Object[10000];

* **No Underlying Data Structure**

Arrays concept not implemented based on some standards data structure.

hence readymade method support not available.

so for every requirement we have to rite code explicitly which increases complicity of programming.

To overcome those problems we should go to collection concepts

## Collections

Collections are grow-able in nature that is based on our requirement we can increase or decrease the size.

Collections can hold both homogeneous and heterogeneous objects.

Every collection class is implemented based on some standard data structure. Hence for every requirement readymade method support is available.

So as a programmer we are responsible to use those methods, no need to implement those methods

#### Collection

Represent Group of objects as a single entity

#### Collection feamework

It contains several classes and interfaces which can be used to represent a group of individual objects as a single entity.

Group of classes and interfaces (libraries)

### Arrays vs. Collections

|  |  |
| --- | --- |
| Arrays | Collections |
| Fixed in size  Once create we can’t increase or decrease size based on requirement | Grow-able in nature  Based on our requirement we can increase or decrees the size |
| With respected to memory  Not recommended to use | Recommended to use |
| With respected to performance  Recommended to use | Not recommended to use |
| Can hold  Only homogeneous elements | Homogeneous and heterogeneous |
| No Underlying data structure  Readymade method support not available, as a programmer we are responsible to write the code based on requirement. | Every collection class Implemented based on some standard data structure  Readymade method support is available, as a programmer we can use those methods based on our requirement |
| Can hold  Object types and primitives | Object types not primitives |

## 9 interfaces in collection framework

**1, Collection 2, List 3, Set 4, SortedSet 5, NavigableSet 6, Queue 7, Map 8, SortedMap 9, NavigableMap**

### Collection(i)

If we want to represent a group of individual object as a single entity then we should go for collection

Collection interface define the **most common methods** which are applicable for any collection object

In general collection(i) is considered as root interface of collection framework

there is no concrete class which implements collection interface directly

#### Collection(i) vs. Collections(c)

Represent group of individual object as single entity 🡺 collection(i)

Collections is an utility class presents in **java.util** package to define several utility methods for collection objects (sorting, searching, ..)

### List(i)

it is the child interface of Collection

if we want to represent a group of individual object in to a single entity where

Duplicates are allowed

Insertion order must be preserved

Collection(i) 1.2 v

List(i) 1.2 v

legacy classes

ArrayList 1.2 v LinkedList 1.2 v Vector these are released

in 1.0 v

Stack others 1.2 v

in 1.2 version vector and stack classes are updated to implement list interface.

### Set(i)

it is the child interface of Collection

if we want to represent a group of individual object in to a single entity where

Duplicates are not allowed

Insertion order not required

Collection(i) 1.2 v

Set(i) 1.2 v

HashSet 1.2 v

LinkedHashSet 1.2 v

### SortedSet(i)

it is the child interface of Collection

if we want to represent a group of individual object in to a single entity where

Duplicates are not allowed

All objects should be inserted according to some sorting order

Collection(i) 1.2 v

Set(i) 1.2 v

SortedSet(i) 1.2 v

NavigableSet(i) 1.6 v

TreeSet 1.2 v

So TreeSet modified at 1.6 v to implement NavigableSet.

### NavigableSet(i)

it is the child interface of SortedSet

it contains several methods for navigation prepossess

(What are the previous element/ next element)

#### list vs. set

|  |  |
| --- | --- |
| List | Set |
| Duplicates are allowed | Duplicates are not allowed |
| Insertion order preserved | Insertion order not preserved |

### Queue(i)

it is the child interface of Collection

if we want to represent a group of individual object prior to processing then we should go for queue

Usually queue follows first-in-first-out order. but based on our requirement we can implement our own priority order also.

Example: before sending mail we have store in some data-structure in which order we added mail added mail-ides in the same order only mail should be delivered for this queue is best choice.

Collection(i) 1.2 v

Queue(i) 1.5 v

PriorityQueue BlockingQueue ……

PriorityBlockingQueue

LinkedBlockingQueue

### Map(i)

All above interfaces meant for representing a group of individual objects

Collections SortedSet

List NavigableSet

Set Queue

If we want to represent a group of objects as key-value pairs. then we should go for map.

Key Value

|  |  |
| --- | --- |
| Reg number | Student name |
| 101 | Durga |
| 102 | Ravi |
| 103 | Karan |
| 104 | Suja |

Map is **not** child interface of collection. Both key and values are objects. for Key duplicate not allowed, but value duplicated values are allowed.

Map(i) 1.2 v Dictionary(AC)

HashMap 1.2 v IdentityHashMap 1.4 v

WeakHashMap 1.2 v Hashtable

LinkedHashMap 1.4 v Properties

legacy classes 1.0 v

### SortedMap(i)

it is the child interface of SortedSet

If we want to represent a group of objects as key-value pairs according to some sorting order of keys. Then we should go for map.

Map(i) 1.2 v

SortedMap(i) 1.2 v

NavigableMap(i) 1.6 v

Treemap 1.2 v

here sorting should be based on key, but not based on value.

### NavigableMap(i)

it is the child interface of SortedMap

it contains several methods for navigation prepossess

## Summary

**1.2 1.2 1.0**

**Collections(i) Map(i) Dictionary(ac)**

**1.2 1.2 1.5 1.2 1.2**

**List(i) Set(i) Que(i) HM WekHM 1.2 Hashtb**

**1.4 SortMap(i)**

**1.2 1.2 1.0 1.2 1.2 PriQ BlockQ … 1.4 IdentyHM TreeMap**

**AL LL Vector HS SortSet(i) LHM 1.6 *Legacy* PriBQ LikdBQ NaviMap(i)**

**Stack 1.4 1.6**

***Legacy* LHS NaviSet(i) 1.2**

**TreeMap**

**1.2**

**TreeSet**

**Sorting**

Default Sorting: **Comparable**

Customized Sorting: **Comparator**

**Cursers** (Get Objects One by One)

**Enumeration** (at old vector)

**Iterator**

**ListIterator**

**Utility Classes**

**Collections**

**Arrays**

These are Legacy Characters

Stack (c)

Vector (c)

Dictionary (ac)

Properties (c)

Hashtable (c)

Enumeration (i)

## Collections

If we want to represent a group of individual object as a single entity then we should go for collection

Collection interface defines the most common methods which are applicable for any collections object

add(Object o)

addAll(Collection c) group of obj = collection

remove(Object o)

removeAll(Collection c)

clear() clear all objects

retainAll(Collection c) clear all objects without these

contains(Object o)

containsAll(Collection c) Return Type: boolean

boolean isEmpty()

int size()

Object[] toArray()

Iterator iterator()

There is no concrete class which Implements Collection(i).

## List

It is the child interface of Collection

If we want to represent a group of individual object in to a single entity where

Duplicates are allowed

Insertion order must be preserved

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | A |
| 0 | 1 | 2 | 3 | 4 | 5 |

We can preserve insertion order with index,

We can differentiate duplicate objects by using index,

Hence index will play very important role in list

#### Methods

l.add(Object o)

l.add(int index, Object o)

l.addAll(int index, Collection c)

l.remove(Object o)

l.remove(int index)

l.get(int index)

l.set(int index, Object o)

l.indexOf(Object o) it retains the first occurs index

l.lastIndexOf(Object o) it retains the last occurs index

l.listIterator() it retains ListIterator object

List(i) 1.2 v

ArrayList 1.2 v LinkedList 1.2 v Vector

Stack

### ArrayList

The underlying data structure is resizable-array/ growable-array

Duplicates are allowed

Insertion order is preserved

Heterogeneous (different type of) objects are allowed

(except TreeSet& TreeMap everywhere allowed)

Null insertion is possible

#### Constructors

ArrayList l = new ArrayList()

creates an empty arraylist-object with default capacity (10) once array list reaches its maximum capacity. then new arraylist-object will be created with new\_capacity (16)

new\_capacity = (current\_capacity \* 3/2) + 1

then copy all objects into new arraylist-object.

ArrayList l = new Arraylist(int initialCapacity)

Creates an empty array list object with

specified initial capacity or

sizes 10🡺 16🡺 25🡺 37 … 100000

ArrayList l = new ArrayList(Collection c)

Create an equivalent arraylist object for the given collection

ArrayList al = new ArrayList<>();

        al.add("A");

        al.add(10);

        al.add(null);

        al.add("A");

        System.out.println(al);      //[A, 10, null, A]

        al.remove(2);

        System.out.println(al);      //[A, 10, A]

        al.remove("A");

        System.out.println(al);      //[10, A]

        al.add(1,"T");

        System.out.println(al);      //[10, T, A]

        al.add("N");

        System.out.println(al);      //[10, T, A, N]

Usually we can use collections to hold and transfer objects from one location to another location [container]. To provide support for this every collection class by default implements serialize-able and clone-able interfaces.

Having cloned object

**Serializable** before doing

internet Operations

**ArrayList and Vector** classes implements RandomAccess interface so that any random element we can access with the same Speed

10th element retrieval 🡺 1s

10oooth element retrieval 🡺 1s

#### RandomAccess(i)

it presents in java.util package and it doesn’t contain any method. It’s a marker interface.

where required ability will be provided automatically by the jvm.

ArrayList al = new ArrayList<>();

LinkedList ll = new LinkedList<>();

System.out.println(al instanceof Serializable);  // true

System.out.println(ll instanceof Cloneable);  // true

System.out.println(al instanceof RandomAccess);  // true

System.out.println(ll instanceof RandomAccess);  //false

ArrayList is the best choice if our frequent operation is retrieval operation (implements RandomAccess(i))

ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle

one corer elements are here

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | … |  |  |

l.add(1,”M”); // so one corer shift operations needed, 2 months will take

l.remove(1);

#### ArrayList vs. Vector

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Non-Synchronized | Synchronized |
| Not Thread-Safe  T1 T2  T3 T4 | Thread-Safe  T1 |
| Relatively performance high | Relatively performance low |
| 1.2v non-legacy | 1.0v legacy |

#### How to get synchronized version of arraylist object.

ArrayList al = new ArrayList<>();

List sl = Collections.synchronizedList(al);

synchronized non-synchronized

By Default arrayList is non-synchronized. but we can get synchronized version of arraylist object by using synchronizedList() method of Collections class

Similarly we can get synchronized version of set and map objects by suing the following methods of collection object.

Class Collections

public static List SynchronizedList (List s)

     public static Set synchronizedSet(Set s)

     public static Map synchronizedMap(Map s)

|  |  |  |
| --- | --- | --- |
|  | B |  |

|  |  |  |
| --- | --- | --- |
|  | C |  |

### LikedList



|  |  |  |
| --- | --- | --- |
|  | A |  |

|  |  |  |
| --- | --- | --- |
|  | M |  |

|  |  |  |
| --- | --- | --- |
|  | D |  |

|  |  |  |
| --- | --- | --- |
|  | E |  |

**next**

**previous**

l.add(1,”m”);

l.remove(3);

it will take months because A🡺M🡺B🡺D🡺… 🡺Z

the underlying data-structure is **double-linkedlist**.

insertion order is preserved

duplicate objects are allowed

heterogeneous objects are allowed

null insertion is possible

LinkedList implements Serializable(i) and Cloneable(i)

but not implements RandomAccess(i)

LinkedList is the best choice if our frequent operation is insertion or deletion in the middle

LinkedList is the worst choice if our frequent operation is retrieval operation

#### constructors

LiknkedList l = new LinkedList()

creats an empty linkedlist object

LinkedList l = new LinkedList(Collection c)

Creates an equal LikedList object for the given Collection

#### Methods

Usually we can use LinkedList to develop stacks and Queues. to provide support for this requirement LinkedList class defines the following specific methods.

   void addFirst(Object o)

        void addLast(Object o)

        Object getFirst()

        Object getLast()

        Object removeFirst()

        Object removeLast()

LinkedList ll = new LinkedList<>();

ll.add("tharsi");

ll.add(10);

ll.add(null);

ll.add("tharsi");    // ["tharsi",10,null,"tharsi"]

//insetrtion order, hetrogenious, null, duplicate

ll.set(0, "element");// ["element",10,null,"tharsi"]

ll.add(0,"ele");    // ["ele","element",10,null,"tharsi"]

ll.removeLast();     // ["ele","element",10,null]

ll.addFirst("helo"); // ["helo","ele","element",10,null]

#### ArrayList vs. LinkedList

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| Best for  Retrieval operation  Implements RandomAccess(i) | Best for  **Insertion or deletion in meddle** |
| Worst for  Insertion or deletion in meddle  Several shift operation need | Worst for  Retrieval operation |
| Elements will be stored in  consecutive memory locations  undelying datastructure: growable array | Elements won’t be stored in  consecutive memory locations  undelying datastructure: **double-linkelist** |

### Vector

The underlying data structure is resizable-array/ growable-array

Duplicates are allowed

Insertion order is preserved

Heterogeneous (different type of) objects are allowed

(except TreeSet& TreeMap everywhere allowed)

Null insertion is possible

implements Serializable(i), Cloneable(i) and RandomAccess(i)

ArrayList Properties **+**

Every methods present inside vector is synchronized,

hence vector object is thread safe

#### Constructors

Vector v = new Vector()

creates an empty vector-object with default capacity (10) once vector reaches its maximum capacity. then new vector-object will be created with new\_capacity (20)

new\_capacity = current\_capacity \* 2

then copy all objects into new arraylist-object.

Vector v = new Vector(int initialCapacity)

Creates an empty vector object with

specified initial capacity

or sizes 10🡺 20🡺 40🡺 80 … 100000

Vector v = new Vector(int initialCapacity, int incrimetalCapacity)

Vector v = new Vector(1000, 10);

after reaches 1000 capacity then increment 10 then 10….

sizes 1000🡺 1010🡺 1020🡺 …

Vector v = new Vector (Collection c)

Create an **equivalent vector** object for the given collection

#### Methods

add(Object o) like Collection

add(int index, Object o) like List

addElement(Object o) like Vector

addAll(Collection c) group of obj = collection

remove(Object o) like Collection

removeElement(Object o) like Vector

remove (int index) like List

removeElementAt(int index) like Vector

clear() like Collection

removeAllElements() like Vector

Object get(int index) like List

Object elementAt(int index) like Vector

Object firstElement() like Vector

Object lastElement() like Vector

int size()

int capacity()

Enumeration elements()

Vector v = new Vector();

        System.out.println(v.capacity()); // 10

        for (int i = 1; i <= 10; i++) {

            v.addElement(i);

        }

        System.out.println(v.capacity()); // 10

        v.addElement("A");

        System.out.println(v.capacity()); // 20

### Stack

Child class of Vector

It is a spatially designed for last in first out

#### Constructors

Vector v = new Vector()

|  |  |  |
| --- | --- | --- |
| Offset |  | index |
| 1 | C | 2 |
| 2 | B | 1 |
| 3 | A | 0 |

#### Methods

push(Object o)

Object pop()

Object peek() don’t remove element

boolean empty() check isEmpty

int search(Object o) returns Offset s.search(A) 🡺 3

s.search(Z)🡺-1

         Stack s = new Stack();

        s.push("A");

        s.push("B");

        s.push("C");                        // [A,B,C]

        System.out.println(s.search("A"));  // 3

        System.out.println(s.search("Z"));  // -1

## 3 cursers of java



If we want objects one by one from the collection then we should go for curser concept.

### Enumeration

#### Methods

public boolean hasMoreElements(); monkey is there any mango?

    public Object nextObject(); monkey give me next mango

#### Example

        Enumeration e = v.elements();

        while(e.hasMoreElements()){

            Integer num =(Integer) e.nextElement();

            if(num%2 == 0){

                System.out.println(num);

            }

        }

#### limitations

Only applicable for **legacy classes**, not a universal curser

Only able to perform **read** operation, can’t perform remove

### Iterator

We can apply iterator concept for **any collection** object, hence it is universal curser

By using iterator we can perform both **read** and **remove** operations

We can create iterator by using iterator method of Collection(i)

public Iterator iterator();

#### Methods

public boolean hasNext(); monkey is there any mango?

    public Object next(); monkey give me next mango (read)

    public void remove(); monkey remove this mango from box

#### Example

Iterator itr = al.iterator();

        while(itr.hasNext()){

            Integer num =(Integer) itr.next();

            if(num%2 == 0){

                System.out.println(num);

            }else{

                itr.remove();

            }

        }

        System.out.println(al);         // [0, 2, 4, 6, 8, 10]

#### limitation

Enumeration and Iterator can move only forward direction

We can perform only read and remove operations can’t perform replacement or add operations

### ListIterator

We can move either to the forward direction or backward direction, hence it is **bidirectional** curser

By using we can perform **Replacement** and **Addison** of new objects in Addison to **read** and **remove** operations

We can create ListIterator object by using iterator method of List(i)

public ListIterator listIterator();

class LinkedList{

public ListIterator<E> listIterator(int index) {

        checkPositionIndex(index);

        return new ListItr(index);

    }

private class ListItr implements ListIterator<E> {

hasNext(){}

next(){}

hasPrevious(){}

previous(){}

…….

}

}

#### Note

the most powerful curser is ListIterator but it is only applicable for list objects.

ListIterator(i) is child interface of Iterator(i). hence read and remove is available to ListIterator

#### methods

boolean hasNext();

Object next();

int nextIndex();

boolean hasPrevious();

Object previous();

int previousIndex();

void remove(); **remove permanently** this mango from box

void add(Object  o); **add permanently** this mango into box

void set(Object  o); **replace permanently** this mango

#### example

l.add("rajini");              // [sivagi, mgr, kamal, rajini]

ListIterator ltr = l.listIterator();

while(ltr.hasNext()){

String s = (String) ltr.next();

if(s.equals("sivaji")){

ltr.remove();         // [mgr, kamal, rajini]

}else if(s.equals("mgr")){

ltr.add("vijay");  // [mgr, vijay, kamal, rajini]

      }else if(s.equals("vijay")){

ltr.set("ajith");     // [mgr, vijay, kamal, rajini]

      }

}

so ltr not consider newly added object

|  |  |  |  |
| --- | --- | --- | --- |
| property | Enumeration | Iterator | ListIterator |
| Where we can apply | Only for Legacy classes (came in 1.0v) | Any collection objects | Only for List objects |
| Movement | Single direction | Single direction | Bidirectional |
| Operations | Read | Read/ Remove | Read/ Remove Replace/ Add |
| How to get | Vector class’s .element() | Collection (i)’s  .iterator() | List (i)’s  .listIterator() |

#### example

       Vector vv = new Vector<>();

        Enumeration ee = vv.elements();

        Iterator ii = vv.iterator();

        ListIterator li = vv.listIterator();

        System.out.println(ee.getClass().getName());

        System.out.println(ii.getClass().getName());

        System.out.println(li.getClass().getName());

java.util.Vector$1 anonymous inner class

java.util.Vector$Itr private inner class

java.util.Vector$ListItr private inner class

## Set

Collection(i) 1.2 v

Set(i) 1.2 v

HashSet 1.2 v SortedSet(i) 1.2 v

LinkedHashSet 1.2 v NavigableSet(i) 1.6 v

TreeSet 1.2 v

Set(i) is Child interface of Collection(i). if we want to represent group of Objects as a single entity where duplicates are not allowed and insertion order not preserved

Set interface any new methods, and we have to use Collection(i) methods

### HashSet