

# Session 04

# Collections

(<http://docs.oracle.com/javase/tutorial/collections/index.html>)

# Objectives

- Collections Framework (package `java.util`):
  - List: `ArrayList`, `Vector` → Duplicates are agreed
  - Set: `HashSet`, `TreeSet` → Duplicates are not agreed
  - Map: `HashMap`, `TreeMap`
  - Queue: `LinkedList`, `PriorityQueue`
  - Deque: `LinkedList`, `ArrayDeque`

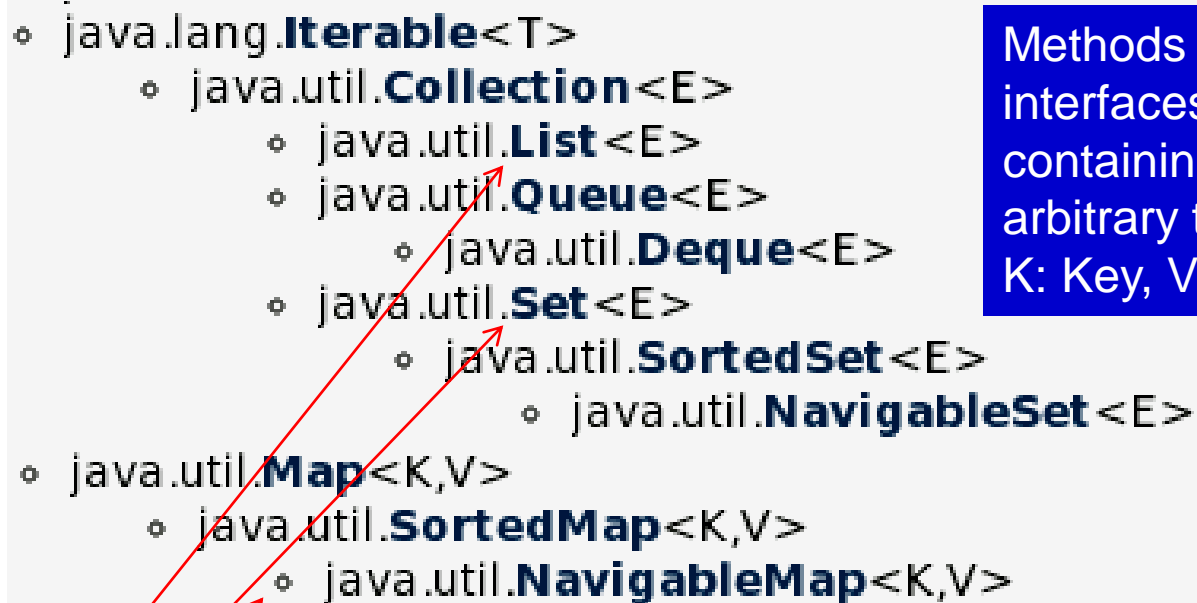
# The Collections Framework

- The Java 2 platform includes a new *collections framework*.
- A *collection* is an object that represents a group of objects.
- The Collections Framework is a unified architecture for representing and manipulating collections.
- The collections framework as a whole is not threadsafe.

# The Collections Framework...

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.

# Collection Interfaces

- java.lang.**Iterable**<T>
    - java.util.**Collection**<E>
      - java.util.**List**<E>
      - java.util.**Queue**<E>
        - java.util.**Deque**<E>
      - java.util.**Set**<E>
        - java.util.**SortedSet**<E>
          - java.util.**NavigableSet**<E>
    - java.util.**Map**<K,V>
      - java.util.**SortedMap**<K,V>
      - java.util.**NavigableMap**<K,V>
- 

Methods declared in these interfaces can work on a list containing elements which belong to arbitrary type. T: type, E: Element, K: Key, V: Value

Details of this will be introduced in the topic Generic

## 3 types of group:

List can contain duplicate elements

Set can contain distinct elements only

Map can contain pairs <key, value>. Key of element is data for fast searching

Queue, Deque contains methods of restricted list.

Common methods on group are: Add, Remove, Search, Clear,...

# Common Methods of the interface Collection

Method	Description	
<code>add(Object x)</code>	Adds x to this collection	<p>Elements can be stored using some ways such as an array, a tree, a hash table.</p> <p>Sometimes, we want to traverse elements as a list → We need a list of references → Iterator</p>
<code>addAll(Collection c)</code>	Adds every element of c to this collection	
<code>clear()</code>	Removes every element from this collection	
<code>contains(Object x)</code>	Returns true if this collection contains x	
<code>containsAll(Collection c)</code>	Returns true if this collection contains every element of c	
<code>isEmpty()</code>	Returns true if this collection contains no elements	
<code>iterator()</code>	Returns an Iterator over this collection (see below)	
<code>remove(Object x)</code>	Removes x from this collection	
<code>removeAll(Collection c)</code>	Removes every element in c from this collection	
<code>retainAll(Collection c)</code>	Removes from this collection every element that is not in c	
<code>size()</code>	Returns the number of elements in this collection	
<code>toArray()</code>	Returns an array containing the elements in this collection	

# The Collection Framework...

## Central Interfaces

- `java.util.Collection<E>`
  - `java.util.List<E>`
    - `java.util.Queue<E>`
      - `java.util.Deque<E>`
    - `java.util.Set<E>`
      - `java.util.SortedSet<E>`
        - `java.util.NavigableSet<E>`
  - `java.util.Map<K,V>`
    - `java.util.SortedMap<K,V>`
      - `java.util.NavigableMap<K,V>`

## Common Used Classes

- `java.util.ArrayList<E>`
- `java.util.Vector<E>`
- `java.util.HashSet<E>`
- `java.util.TreeSet<E>`
- `java.util.HashMap<K,V>`
- `java.util.TreeMap<K,V>`

Store: Dynamic array  
Use index to access an element.

Store: Specific structure/tree  
Use iterator to access elements

**java.lang.Comparable interface**

`keySet()`  
`values()`

Use  
iterator

A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings.

If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the Comparable interface.

# Arrays vs Collections

## Arrays

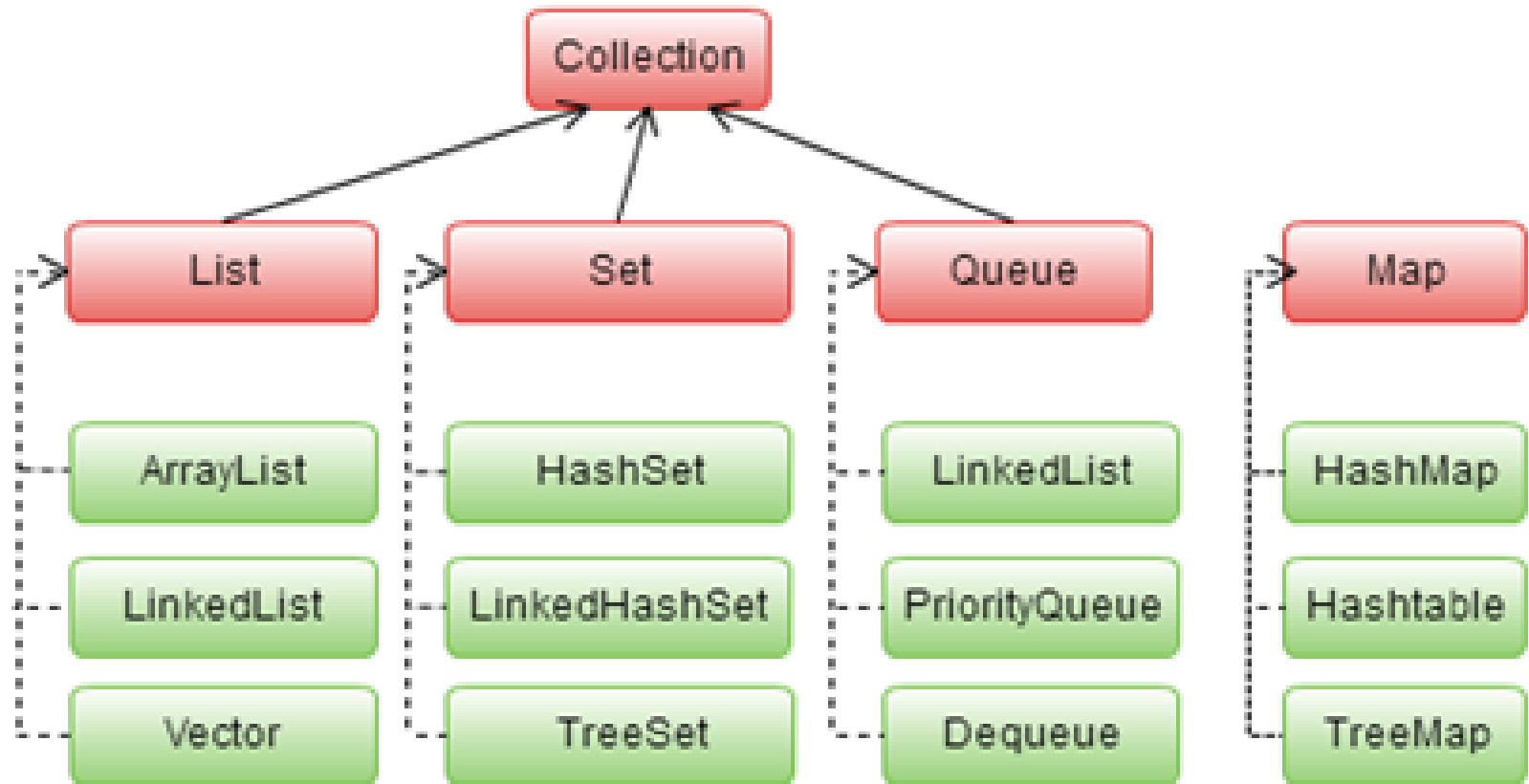
1. Size is fixed
2. to memory arrays are not good to use, but to performance its better to use arrays
3. Can hold both primitive types(byte, sort, int, long ...etc) and object types.
4. There is no underlying data structures in arrays. The array itself used as data structure in java.
5. There is no utility methods in arrays

## Collections

1. Size is not fixed (dynamic in size), size is growable.
2. to memory Collections are better to use, but to performance collection are not good to use.
3. Can hold only object types
4. Every Collection class there is underlying data structure.
5. Every Collection provides utility methods (sorting, searching, retrieving etc...). It will reduce the coding time.



# Collection interface



# Lists

- A List keeps its elements in the order in which they were added.
- Each element of a List has an index, starting from 0.
- Common methods:
  - **void add(int index, Object x)**
  - **Object get(int index)**
  - **int indexOf(Object x)**
  - **Object remove(int index)**

# Classes Implementing the interface List

- AbstractList
- ArrayList
- Vector (like ArrayList but it is **synchronized**)
- LinkedList: *linked lists can be used as a stack, queue, or double-ended queue (deque)*

# ArrayList

- ArrayList is a class that implements the List interface.
- The advantage of ArrayList over the general Array is that ArrayList is dynamic and the size of ArrayList can grow or shrink.
- ArrayList stores the elements in the insertion order.
- ArrayList can have duplicate elements. ArrayList is non synchronized.

# ArrayList

Sr.No.	Constructor & Description
1	<b>ArrayList()</b> This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	<b>ArrayList(Collection c)</b> This constructor is used to create a list containing the elements of the specified collection.
3	<b>ArrayList(int initialCapacity)</b> This constructor is used to create an empty list with an initial capacity.

# Methods

Methods	Description
<b>void add(int index, Object element)</b>	Inserts the specified element at the specified position in this list.
<b>boolean add(Object o)</b>	Appends the specified element to the end of this list.
<b>boolean addAll(Collection c)</b>	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
<b>void clear()</b>	Removes all of the elements from this list.
<b>boolean contains(Object o)</b>	returns true if this list contains the specified element.
<b>Object get(int index)</b>	returns the element at the specified position in this list.

Methods	Description
<b>protected void removeRange(int fromIndex, int toIndex)</b>	removes from this list all of the elements whose index is between fromIndex(inclusive) and toIndex(exclusive).
<b>Object set(int index, Object element)</b>	replaces the element at the specified position in this list with the specified element.
<b>int size()</b>	returns the number of elements in this list.
<b>int indexOf(Object o) int lastIndexOf(Object o)</b>	returns the index of the first (last) occurrence of the specified element in this list, or -1 if this list does not contain the element.
<b>Object remove(int index)</b>	removes the element at the specified position in this list.
<b>boolean remove(Object o)</b>	removes the first occurrence of the specified element from this list, if it is present.

# Example

```
public class ArrayListDemo {  
    public static void main(String  
        args[]) {  
        ArrayList al = new ArrayList();  
        System.out.println("The size at  
beginning: " + al.size());  
        //add elements  
        al.add("C");           al.add("A");  
        al.add("E");           al.add("B");  
        al.add("D");           al.add("F");  
        al.add(1, "A2");  
    }  
}
```



```
System.out.println("The size after: "
    + al.size());
    System.out.println("Content: " +
al);
    al.remove("F"); al.remove(2);
    System.out.println("The Size after
to remove: " + al.size());
    System.out.println("Content: " +
al);
}
}
```

# Output

The size at beginning: 0

The size after: 7

Content: [C, A2, A, E, B, D, F]

The Size after to remove: 5

Content: [C, A2, E, B, D]

# Generic in java

- The **Java Generics** programming is introduced in Java SE 5 to deal with type-safe objects.
- Generics, forces the java programmer to store specific type of objects.
- There are mainly 3 advantages of generics:
  - **Type-safety** : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
  - **Type casting is not required**: There is no need to typecast the object.

we need to type cast.

```
List list = new ArrayList();
```

```
list.add("hello");
```

```
String s = (String) list.get(0); //typecasting
```

we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();  
list.add("hello");
```

```
String s = list.get(0);
```

- **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32); //Compile Time Error
```

# Example

```
ArrayList<Integer> mylist = new  
    ArrayList<Integer>();
```

```
mylist.add(10);  
mylist.add("Hi");//error  
mylist.add(true);//error  
mylist.add(15);
```

*Generic*

# ArrayList Implementing Classes

```
ArrayList al = new ArrayList();  
for (int i = 101; i <= 110; i++) {  
    al.add(i);  
}  
for (int i = 0; i < al.size(); i++) {  
    System.out.println(al.get(i));  
}  
//or using Iterator  
    Iterator iter = al.iterator();  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }
```

# Example: ArrayList of Car, Motor, Truck

```
public class ListVehicle {  
    ArrayList<Vehicle> list;  
    public ListVehicle() {  
        list=new ArrayList<Vehicle>();  
    }  
    private Vehicle input() {  
        Vehicle v;  
        String manufacturer;  
        int year;           double cost;  
        String color;  
        Scanner in=new Scanner(System.in);
```

```
while (true) {  
    System.out.println("manufacturer (Mazda201  
        7) : ");  
        manufacturer = in.nextLine();  
    if (manufacturer.matches("^[A-Za-  
        z]\\d{4}$"))  
        break; }  
}
```



```
while (true) {  
    System.out.println("manufacturer (Mazda2017) :  
        ");  
        manufacturer = in.nextLine();  
    if (manufacturer.matches ("^[A-Za-z] \\d{4}$"))  
        break; }  
    System.out.print ("Manufacture year:");  
    year=Integer.parseInt(in.nextLine());  
    System.out.print ("Cost:");  
    cost=Double.parseDouble(in.nextLine());  
    System.out.print ("Color:");  
    color=in.nextLine();  
    v=new Vehicle (manufacturer, year, cost,  
        color);  
        return v;    }
```

```
public boolean inputCar() {  
    String typeofEngine;  
    int seats;  
    Vehicle v=input();  
    Scanner in=new Scanner(System.in);  
    System.out.print("Type of engine:");  
    typeofEngine=in.nextLine();  
    System.out.print("Number of Seats:");  
    seats=in.nextInt();  
    return list.add(new  
    Car(v.getManufacturer(),  
    v.getYear(),v.getCost(),v.getColor(),  
        typeofEngine, seats));  
}
```

```
public boolean inputMotor() {  
    double power;  
    Vehicle v=input();  
    Scanner in=new Scanner(System.in);  
    System.out.print("Power:");  
    power=in.nextDouble();  
    return  
list.add(new  
    Motor(v.getManufacturer(),v.getYear(  
    ),v.getCost(),v.getColor(),power));  
}
```

```
public boolean inputTruck() {  
    double load;  
    Vehicle v=input();  
    Scanner in=new  
Scanner(System.in);  
    System.out.print("Load:");  
    load=in.nextDouble();  
    return list.add(new  
Truck(v.getManufacturer(),v.getYea  
r(), v.getCost(),v.getColor(),  
load));  
}
```

```

public void output() {
    System.out.println("List of
Cars");
    System.out.println("Manufacturer
Year      Cost      Color TypeofEngine
NumberofSeats");
    for(Vehicle v:list) {
        if(v instanceof Car)
            System.out.println(v.toString());
    }
    System.out.println("-----
-----");
}

```

```
System.out.println("List of Motors");  
    System.out.println("Manufacturer  
Year      Cost      Color Power");  
        for(Vehicle v:list) {  
            if(v instanceof Motor)  
System.out.println(v.toString());  
        }  
  
        System.out.println("-----  
-----");
```

```
System.out.println("List of Trucks");  
    System.out.println("Manufacturer  
Year      Cost      Color Load");  
    for(Vehicle v:list) {  
        if(v instanceof Truck)  
System.out.println(v.toString());  
    }  
}
```

```

public void getByManufacturer (String
    manufacturer) {
    System.out.println("====A list of
    Vehicles=====");
    System.out.println("Manufacture    Year
    Cost Color");
    for (Vehicle v: list)
    if (v.getManufacturer().equals(manufact
    urer))
    System.out.println(v.toString());
    System.out.println("=====
    =====");
}

```



```

public void getByManufacturerA(String
    manufacturer) {
        System.out.println("====A
list of Vehicles=====");
        System.out.println("Manufacture
Year    Cost    Color");
        for (Vehicle v: list)
            if (v.getManufacturer().indexOf(manuf
acturer) >= 0)
                System.out.println(v.toString());
        System.out.println("=====
=====");
    }

```

```
public void getByCostGreater (double
    cost) {
    System.out.println("A list of
    Vehicles");
    System.out.println("Manufacture    Year
    Cost    Color");
    for (Vehicle v: list)
        if (v.getCost() >= cost)
            System.out.println(v.toString());
    System.out.println("=====");
}
```

```
public void getByCostFrom(double from,
    double to) {
    System.out.println("A list of
    Vehicles");
    System.out.println("Manufacture    Year
    Cost Color");
    for (Vehicle v: list)
        if ((v.getCost() >= from) &&
            (v.getCost() <= to))
            System.out.println(v.toString());
    System.out.println("=====");
}
```

# The Collections class

- A support class containing static methods which accept **collections as their parameters**.

# Collections Demo

```
] import java.util.ArrayList;
import java.util.Vector;
import java.util.Collections;
- import java.util.Random;
public class CollectionsDemo {
]     public static void main(String[] args){
        ArrayList ar= new ArrayList();
        Vector v = new Vector();
        Random rd= new Random();
        for (int i=1; i<=10; i++){
            ar.add(rd.nextInt(30));
            v.add(rd.nextInt(30));
        }
        System.out.println("ar=" + ar);
        System.out.println("v=" + v);
        boolean dis= Collections.disjoint(ar, v);
        System.out.println("ar and v is disjunct: " + dis);
        Collections.addAll(v, ar.toArray());
        System.out.println("After adding, v=" + v);
        int minVal= (int)Collections.min(v);
        int maxVal= (int) Collections.max(v);
    }
}
```

```

System.out.println("min= " + minVal + ", max= " + maxVal);
int fre= Collections.frequency(v, 8);
System.out.println("Occurences of 8: " + fre);
Collections.sort(v);
System.out.println("After sorting, v=" + v);
int pos = Collections.binarySearch(v, 8);
System.out.println("Position of 8: " + pos);
Collections.shuffle(v);
System.out.println("After shuffling, v=" + v);
}
}

```

run:

ar=[16, 22, 13, 29, 12, 8, 23, 8, 17, 10]

v=[3, 2, 24, 13, 24, 18, 22, 8, 3, 1]

ar and v is disjunct: false

After adding, v=[3, 2, 24, 13, 24, 18, 22, 8, 3, 1, 16, 22, 13, 29, 12, 8, 23, 8, 17, 10]

min= 1, max= 29

Occurences of 8: 3

After sorting, v=[1, 2, 3, 3, 8, 8, 8, 10, 12, 13, 13, 16, 17, 18, 22, 22, 23, 24, 24, 29]

Position of 8: 4

After shuffling, v=[3, 3, 17, 8, 23, 8, 12, 24, 13, 18, 2, 24, 1, 29, 22, 16, 22, 13, 10, 8]

# Sorting

- The **sort(List<T>,Comparator<T> c)** method is used to sort the specified list according to the order induced by the specified comparator.
- `Collections.sort(List<T> list, Comparator<T> c)`
  - **list** – This is the list to be sorted.
  - **c** – This is the comparator to determine the order of the list.

# Comparator Interface

- A comparison function, which imposes a total ordering on some collection of objects
- The following demonstration will show you the way to sort a list based on your own criteria: A list of employees will be sorted based on descending salaries then ascending IDs.



# Comparator Interface – Demo

```
package sort;

import java.lang.Comparable;
import java.util.Comparator;

public class Employee implements Comparable {
    String ID="", name="";
    int salary=0;

    public Employee(String id, String n, int s){
        ID= id; name= n; salary=s;
    }

    @Override
    public String toString(){
        return ID + "," + name + "," + salary;
    }

    @Override // standard comparing
    public int compareTo(Object emp){
        return ID.compareTo(((Employee)emp).ID);
    }
}
```

# Comparator Interface- Demo

Comparing 2 employees based on descending salaries then ascending IDs

```
// comparing on salary descending then ID
public static Comparator compareObj= new Comparator() {
    @Override
    public int compare(Object e1, Object e2){
        Employee emp1 = (Employee) e1;
        Employee emp2 = (Employee) e2;
        int d= emp1.salary - emp2.salary;
        if (d>0) return -1; // lower salary -> move upper
        if (d==0) return emp1.ID.compareTo(emp2.ID);
        return 1;
    }
};
```

Create an  
anonymous  
object for  
comparing 2  
employees

# Comparator Interface- Demo

```
package sort;
import java.util.ArrayList;
import java.util.Collections;
public class SortDemo {
    public static void main(String[] args){
        ArrayList<Employee> list= new ArrayList<Employee>();
        list.add(new Employee("ID004", "Michel", 400));
        list.add(new Employee("ID001", "Helen", 200));
        list.add(new Employee("ID003", "Hemming", 400));
        System.out.println("Sorting on IDs ascending");
        Collections.sort(list);
        System.out.println(list);
        System.out.println("Sorting on descending salary then ascending IDs");
        Collections.sort(list, Employee.compareObj);
        System.out.println(list);
    }
}
```

run:

Sorting on IDs ascending

[ID001,Helen,200, ID003,Hemming,400, ID004,Michel,400]

Sorting on descending salary then ascending IDs

[ID003,Hemming,400, ID004,Michel,400, ID001,Helen,200]

# Routine Data Manipulation

- The Collections class provides five algorithms for doing routine data manipulation on List objects, including:
  - reverse()
  - fill()
  - copy()
  - swap()
  - addAll()

# Searching

- Condition: The list in ascending order
- The binarySearch algorithm searches for a specified element in a sorted List.
  - Return  $\text{pos} \geq 0 \rightarrow \text{Present}$
  - Return  $\text{pos} < 0 \rightarrow \text{Absent}$

# Composition

- `frequency` — counts the number of times the specified element occurs in the specified collection.
- `disjoint` — determines whether two `Collections` are disjoint; that is, whether they contain no elements in common.

```
public void sortByManufacturer() {  
    Collections.sort(list, new  
    Comparator<Vehicle>() {  
        @Override  
        public int compare(Vehicle v1,  
Vehicle v2) {  
            return  
            ((v1.getManufacturer().compareToIgnore  
reCase(v2.getManufacturer())));  
        }  
    }  
);  
}
```

```
public void sortByCost() {  
    Collections.sort(list, new  
    Comparator<Vehicle>() {  
        public int compare(Vehicle v1,  
        Vehicle v2) {  
            return (int)v1.getCost() -  
            (int)v2.getCost();  
        }  
    }  
);  
}
```



```
public static void main(String[] args) {  
    ListVehicle a=new ListVehicle();  
    Scanner in=new Scanner(System.in);  
    while(true) {  
        System.out.print("\n 1. input a Car");  
        System.out.print("\n 2. input a Motor");  
        System.out.print("\n 3. input a Truck");  
        System.out.print("\n 4. output a list of Vehicles");  
        System.out.print("\n 5. Search by manufacturer");  
        System.out.print("\n 6. Search by manufacturer ()");  
        System.out.print("\n 7. Search by cost (greater than)");  
        System.out.print("\n 8. Search by cost (from to)");  
        System.out.print("\n 9. Sort by manufacturer");  
        System.out.print("\n 10. Sort by type of engine");  
        System.out.print("\n 0. Exit");  
        System.out.print("\n Your choice(0->10): ");  
        int choice;
```

# Example: Document, Book, Magazine, Newspaper

```
public static void main(String[] args) {  
    ListDocument d=new ListDocument();  
    Scanner in=new Scanner(System.in);  
    while(true) {  
        System.out.print("\n 1. input a Book");  
        System.out.print("\n 2. input a Magazine");  
        System.out.print("\n 3. input a Newspaper");  
        System.out.print("\n 4. output a list of Documents");  
        System.out.print("\n 5. Delete a Document");  
        System.out.print("\n 6. Edit a Document");  
  
        System.out.print("\n 0. Exit");  
        System.out.print("\n Your choice(0->6): ");  
        int choice;  
        choice=in.nextInt();  
    }  
}
```

# ArrayList to Array Conversion

- **public Object[] toArray()**

```
List<String> list = new ArrayList();  
    // thêm lần lượt 4 phần tử.  
    list.add(" Journal of ");  
    list.add("Science");  
    list.add("and Technology");  
    list.add("on Information and  
Communications");  
    //convert listString tới array.  
String[] array = list.toArray(new  
    String[list.size()]);  
    System.out.println("\n  
"+Arrays.toString(array));
```

# Convert Array to ArrayList

- `Arrays.asList(T... a)`

```
List<String> listNames =  
    Arrays.asList("John", "Peter",  
        "Tom", "Mary");  
List<Integer> listNumbers =  
    Arrays.asList(1, 3, 5, 7, 9, 2,  
        4, 6, 8);  
System.out.println(listNames);  
System.out.println(listNumbers);
```

# The Vector Class

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collections framework.
- Constructors:
  - **Vector( )**: This constructor creates a default vector, which has an initial capacity of 10.
  - **Vector(int size)**: This constructor accepts an argument that equals to the required size

- **Vector(int size, int incr):** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr.
- **Vector(Collection c):** This constructor creates a vector that contains the elements of collection c.

# Methods

**void add(int index, Object element)**

Inserts the specified element at the specified position in this Vector.

**boolean add(Object o)**

Appends the specified element to the end of this Vector.

**boolean addAll(Collection c)**

Appends all of the elements in the specified Collection to the end of this Vector

**void clear()**

Removes all of the elements from this vector.

**boolean contains(Object elem)**

Tests if the specified object is a component in this vector.

**boolean containsAll(Collection c)**

Returns true if this vector contains all of the elements in the specified Collection.

## **Object elementAt(int index)**

Returns the component at the specified index.

## **Enumeration elements()**

Returns an enumeration of the components of this vector.

## **boolean equals(Object o)**

Compares the specified Object with this vector for equality.

## **Object firstElement()**

Returns the first component (the item at index 0) of this vector.

## **Object get(int index)**

Returns the element at the specified position in this vector.

## **int indexOf(Object elem), int indexOf(Object elem, int index) :**

Searches for the first occurrence of the given argument (beginning the search at index, ), testing for equality using the equals method.



**void insertElementAt(Object obj, int index)**

Inserts the specified object as a component in this vector at the specified index.

**boolean isEmpty()**

Tests if this vector has no components.

**Object lastElement()**

Returns the last component of the vector.

**Object remove(int index), boolean remove(Object o):** Removes the element at the specified position in this vector. Removes the first occurrence of the specified element in this vector,

**void removeAllElements()**

Removes all components from this vector and sets its size to zero.

**Object set(int index, Object element)**

Replaces the element at the specified position in this vector with the specified element.

**int size()**

Returns the number of components in this vector.

## **List subList(int fromIndex, int toIndex)**

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

## **Object[] toArray()**

Returns an array containing all of the elements in this vector in the correct order.

## **String toString()**

Returns a string representation of this vector, containing the String representation of each element.

## **void trimToSize()**

Trims the capacity of this vector to be the vector's current size.

## Example

```
public class VectorDemo {  
    public static void main(String args[]) {  
        Vector v = new Vector(3, 2);  
        System.out.println("Begin size: " +  
v.size());  
        System.out.println("start Capacity: "  
+v.capacity());  
        v.addElement(new Integer(1));  
        v.addElement(new Integer(2));  
        v.addElement(new Integer(3));  
        v.addElement(new Integer(4));  
        System.out.println("Capacityla:  
"+v.capacity());  
    }  
}
```

```
v.addElement(new Double(5.45));  
System.out.println("Capacity : " +  
    v.capacity());  
    v.addElement(new Double(6.08));  
    v.addElement(new Integer(7));  
System.out.println("Capacity : " +  
v.capacity());  
    System.out.println("The first element:  
" +  
        (Integer)v.firstElement());  
    System.out.println("The last element:  
" +  
        (Integer)v.lastElement());  
    if(v.contains(new Integer(3)))  
System.out.println("Vector contains 3");
```

```
Enumeration vEnum =  
    v.elements();  
        System.out.println("\n  
Vector:");  
while (vEnum.hasMoreElements())  
    System.out.print(vEnum.nextElement()  
        + " ");  
        System.out.println();  
    } }
```

# Using the Vector class

`java.util.Vector<E>` (implements `java.lang.Cloneable`,  
`java.util.List<E>`, `java.util.RandomAccess`, `java.io.Serializable`)

The Vector class is obsolete from Java 1.6 but it is still introduced because it is a parameter in the constructor of the `javax.swing.JTable` class, a class will be introduced in GUI programming.

```
import java.util.Vector;
class Point {
    int x,y;
    Point() { x=0; y=0; }
    Point(int xx, int yy) {
        x=xx; y=yy;
    }
    public String toString() { return "[" + x + "," + y + "];"}
}
public class UseVector {
    public static void main(String[] args) {
        Vector v = new Vector();
        v.add(15);
        v.add("Hello");
        v.add(new Point());
        v.add(new Point(5,-7));
        System.out.println(v);
        v.remove(2);
        System.out.println(v);
        for (int i=0;i<v.size();i++) System.out.print(v.get(i) + ", ");
        System.out.println();
    }
}
```

Output - Chapter08 (run)

run:

[15, Hello, [0,0], [5,-7]]

[15, Hello, [5,-7]]

15, Hello, [5,-7],

# Fraction class

```
public class Fraction{  
    private int numerator;  
    private int denominator;  
    public Fraction(int n, int d) {  
        if(d != 0) {  
            numerator = n;  
            denominator = d;  
        }  
        else  
            System.exit(0);  
    }  
    // getter/setter
```

```

private static int GCD(int x, int y) {
    int mod;
    if(x < y) {
        mod = x;
        x = y;
        y = mod;    }
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;  }
    return y;
}

private Fraction reduce(int n, int d) {
    int gcd = GCD(n,d);
    d = d / gcd;
    n = n / gcd;
    return new Fraction(n,d);
}

```



```
public Fraction add(Fraction b) {  
    int num1 = (this.numerator *  
    b.denomirator) + (b.numerator *  
    this.denomirator);  
    int num2 = this.denomirator *  
    b.denomirator;  
    return reduce(num1, num2);    }  
public Fraction subtract(Fraction b) {  
    int num1 = (this. numerator *  
    b.denomirator) - (b.numerator *  
    this.denomirator);  
    int num2 = this.denomirator *  
    b.denomirator;  
    return reduce(num1, num2);    }
```

```
public Fraction multiply(Fraction b) {  
    int num1 = this.numerator *  
    b.numerator;  
    int num2 = this.denominator *  
    b.denominator;  
    return reduce(num1, num2);  
}  
public Fraction divide(Fraction b) {  
    int num1 = this.numerator *  
    b.denominator;  
    int num2 = this.denominator * b.  
    numerator;  
    return reduce(num1, num2);  
}
```

```
public String toString() {  
    if (numerator>denomirator  
        &&denomirator>1)  
        return (numerator+"/"+  
denomirator+" or "+(numerator  
/denomirator)+" "+(numerator  
%denomirator)+" / "+  
denomirator);  
    else  
        return (numerator+"/"+  
denomirator);  
} }
```

# Sets

- Lists are based on an ordering of their members. Sets have no concept of order.
- A Set is just a cluster of references to objects.
- Sets may **not** contain **duplicate** elements.
- Sets use the `equals()` method, not the `==` operator, to check for duplication of elements.

```
void addTwice(Set set) {  
    set.clear();  
    Point p1 = new Point(10, 20);  
    Point p2 = new Point(10, 20);  
    set.add(p1);  
    set.add(p2);  
    System.out.println(set.size());  
}
```



will print out 1, not 2.

# Sets...

- Set extends Collection but does not add any additional methods.
- The two most commonly used implementing classes are:
  - **TreeSet**
    - Guarantees that the sorted set will be in ascending element order.
    - $\log(n)$  time cost for the basic operations (add, remove and contains).
  - **HashSet**
    - Constant time performance for the basic operations (add, remove, contains and size).

# TreeSet and Iterator

- Ordered Tree – Introduced in the subject Discrete Mathematics
- Set: Group of different elements
- TreeSet: Set + ordered tree, each element is called as node
- Iterator: An operation in which references of all nodes are grouped to make a linked list. Iterator is a way to access every node of a tree.
- Linked list: a group of elements, each element contains a reference to the next

# TreeSet = Set + Tree

The result may be:

```
Random r = new Random();
TreeSet myset = new TreeSet();
for (int i = 0; i < 10; i++) {
    int number = r.nextInt(100);
    myset.add(number);
}
//using Iterator
Iterator iter = myset.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```



7  
27  
36  
41  
43  
46  
49  
57  
75  
83

# Using the TreeSet class & Iterator

```
import java.util.TreeSet;
import java.util.Iterator;
public class UseTreeSet {
    public static void main (String[] args){
        TreeSet t= new TreeSet();
        t.add(5); t.add(2); t.add(9);t.add(30); t.add(9);
        System.out.println(t);
        t.remove(9);
        System.out.println(t);
        Iterator it= t.iterator();
        while (it.hasNext())
            System.out.print(it.next() + ", ");
        System.out.println();
    }
}
```

**Output - Chapter08 (run)**

run:  
[2, 5, 9, 30]  
[2, 5, 30]  
2, 5, 30,

A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings.

If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the `Comparable` interface.



# Hash Table

- In array, elements are stored in a contiguous memory blocks → Linear search is applied → slow, binary search is an improvement.
- Hash table: elements can be stored in a different memory blocks. The index of an element is determined by a function (hash function) → Add/Search operation is very fast ( $O(1)$ ).



The hash function  $f$  may be:

`'S'*10000+'m'*1000+'i'*100+'t'*10+'h' % 50`

49	
14	Brown
9	Hoa
5	Smith
0	Line1

# HashSet = Set + Hash Table

```
Random r = new Random();
HashSet myset = new HashSet();
for (int i = 0; i < 10; i++) {
    int number = r.nextInt(100);
    myset.add(number);
}
//using Iterator
Iterator iter = myset.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

The result may be:



84
55
7
76
77
95
94
12
91
44

# HashSet or TreeSet?

- If you care about iteration order, use a Tree Set and pay the time penalty.
- If iteration order doesn't matter, use the higher-performance Hash Set.

# How to TreeSet ordering elements?

- Tree Sets rely on all their elements implementing the interface `java.lang.Comparable`.

`public int compareTo(Object x)`

- Returns a positive number if the current object is “greater than” x, by whatever definition of “greater than” the class itself wants to use.

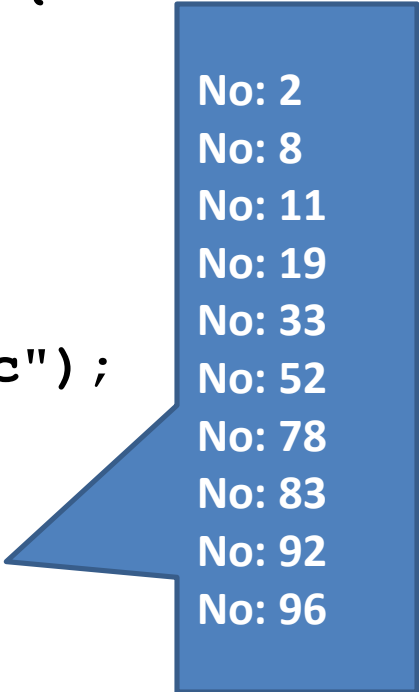
# How to TreeSet ordering elements?

```
class Student implements Comparable{
    int no;
    ...
    public int compareTo(Object o) {
        Student st = (Student) o;
        if(no > st.getNo())
            return 1;
        else if(no == st.getNo())
            return 0;
        else
            return -1;
    }
    . . .
}
```

Comparing 2 students  
based on their IDs (  
field no)

# How to TreeSet ordering elements?

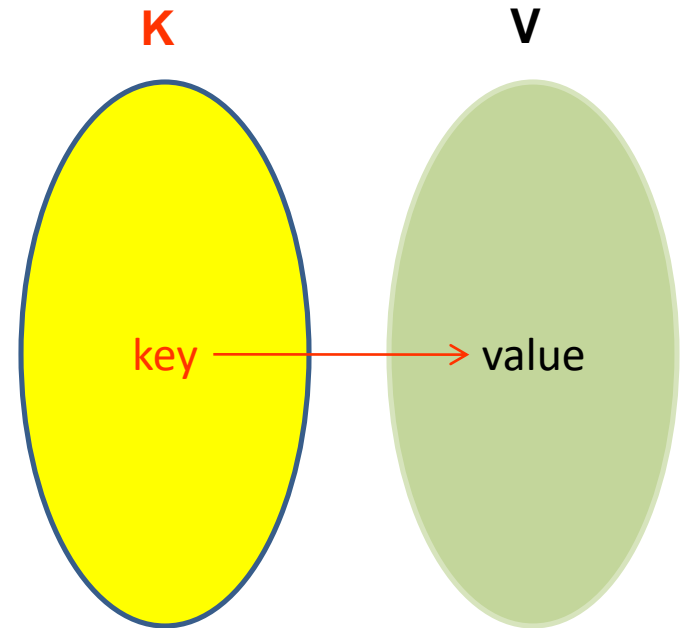
```
public static void main(String[] args) {  
    Random r = new Random();  
    TreeSet myset = new TreeSet();  
    for (int i = 0; i < 10; i++) {  
        int no = r.nextInt(100);  
        Student st = new Student(no, "abc");  
        myset.add(st);  
    }  
    //using Iterator  
    Iterator iter = myset.iterator();  
    while (iter.hasNext()) {  
        Student st = (Student)iter.next();  
        System.out.println("No: " + st.getNo());  
    }  
}
```



No: 2  
No: 8  
No: 11  
No: 19  
No: 33  
No: 52  
No: 78  
No: 83  
No: 92  
No: 96

# Maps

- Map doesn't implement the `java.util.Collection` interface.
- A Map combines *two* collections, called keys and values.
- The Map's job is to associate exactly one value with each key.
- A Map like a dictionary.
- Maps check for key uniqueness based on the `equals()` method, not the `==` operator.
- IDs, Item code, roll numbers are keys.
- The normal data type for keys is `String`.



Each element: `<key,value>`

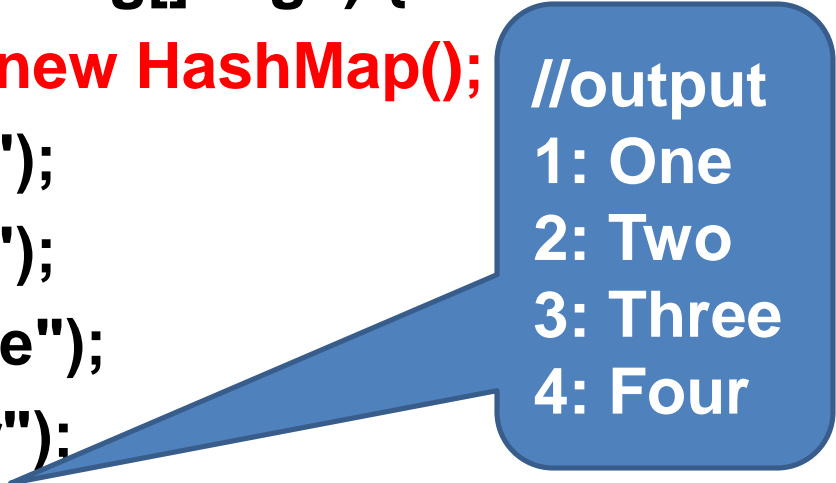
# Maps..

- Java's two most important Map classes:
  - HashMap (mapping keys are unpredictable order – hash table is used, hash function is pre-defined in the Java Library).
  - TreeMap (mapping keys are natural order)-> all keys must implement Comparable (a tree is used to store elements).



# HashMap

```
public static void main(String[] args) {  
    HashMap mymap = new HashMap();  
    mymap.put(1, "One");  
    mymap.put(2, "Two");  
    mymap.put(3, "Three");  
    mymap.put(4, "Four");  
    //using Iterator  
    Iterator iter = mymap.keySet().iterator();  
    while (iter.hasNext()) {  
        Object key = iter.next();  
        System.out.println(key + ": " + mymap.get(key));  
    }  
}
```



//output  
1: One  
2: Two  
3: Three  
4: Four

Key: integer, value: String

# Using HashMap class & Iterator

```
1 import java.util.HashMap;
2 import java.util.Iterator;
3 public class UseHashMap {
4     public static void main(String[] args){
5         HashMap h = new HashMap();
6         h.put("Sáu Tấn", "Huỳnh Anh Tuấn");
7         h.put("Bình Gà", "Nguyễn Tấn Sầu");
8         h.put("Ba Địa", "Trần Mai Hoà");
9         System.out.println(h);
10        h.put("Sáu Tấn", "Nguyễn Văn Tuấn");
11        System.out.println(h);
12        h.remove("Bình Gà");
13        System.out.println(h);
14        Iterator it = h.keySet().iterator();
15        while (it.hasNext())
16        { String key= (String)(it.next());
17          String value = (String)(h.get(key));
18          System.out.println(key + ", " + value);
19        }
20    }
21 }
```

Key: String, value: String

## Output - Chapter08 (run)

```
run:
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Huỳnh Anh Tuấn, Bình Gà=Nguyễn Tấn Sầu}
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Nguyễn Văn Tuấn, Bình Gà=Nguyễn Tấn Sầu}
{Ba Địa= Trần Mai Hoà, Sáu Tấn=Nguyễn Văn Tuấn}
Ba Địa, Trần Mai Hoà
Sáu Tấn, Nguyễn Văn Tuấn
BUILD SUCCESSFUL (total time: 1 second)
```

# Stream Collectors groupingBy()

- Java 8 now directly allows you to do GROUP BY in Java by using `Collectors.groupingBy()` method.
- ```
class Item {  
    private String name;  
    private int qty;  
    private Double price;  
    //constructor and getter and setter  
    // toString
```

```
//Group by Name and counting
Map<String, Long> counting =
items.stream().collect(
Collectors.groupingBy(Item::getName,
Collectors.counting()));

        System.out.println(counting);
```

```
//Group by Name and sum of qty
Map<String, Integer> sum =
items.stream().collect(
Collectors.groupingBy(Item::getName,
Collectors.summingInt(Item::getQty)));

        System.out.println(sum);
```

```
//Group by Name and average of qty
Map<String,Double> avg =
items.stream().collect(
Collectors.groupingBy(Item::getName,
Collectors.averagingInt(Item::getQty)));
        System.out.println(avg);
//max of a list of Items
Optional<Item> max =
        items.stream()
        .collect(Collectors.maxBy(Comparator.
comparing(Item::getQty)));
        System.out.println("Item with
max qty:"+(max.isPresent()?
max.get():"Not Applicable"));
```

```
//min of a list of Items
Optional<Item> min =
    items.stream()
        .collect(Collectors.minBy(Comparator.
            comparing(Item::getQty)));
    System.out.println("Item with
min qty:"+(min.isPresent()?
min.get():"Not Applicable"));
```

```
// max in each group
Map<String, Item> o =
    items.stream().collect(
        Collectors.groupingBy(Item::getName,
            Collectors.collectingAndThen(
                Collectors.reducing((Item d1,
                    Item d2) -> d1.getPrice() >
                        d2.getPrice() ? d1 : d2),
                    Optional::get)));
    System.out.println(o);
```

# Interface Queue and Deque



- Interfaces for restricted list (limited manipulation), programmers can not access an arbitrary element but elements at the beginning or the end of the list only.
- **Deque**: A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.



# Interface Queue



public interface **Queue**<E> extends [Collection](#)<E>

|                   |                                                                                                                                                                                                                                                                                                    |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean           | <a href="#"><b>add</b></a> ( <a href="#">E</a> e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an <code>IllegalStateException</code> if no space is currently available. |
| <a href="#">E</a> | <a href="#"><b>element</b></a> () Retrieves, but does not remove, the head of this queue.                                                                                                                                                                                                          |
| boolean           | <a href="#"><b>offer</b></a> ( <a href="#">E</a> e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.                                                                                                                  |
| <a href="#">E</a> | <a href="#"><b>peek</b></a> () Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.                                                                                                                                                                     |
| <a href="#">E</a> | <a href="#"><b>poll</b></a> () Retrieves and removes the head of this queue, or returns null if this queue is empty.                                                                                                                                                                               |
| <a href="#">E</a> | <a href="#"><b>remove</b></a> () Retrieves and removes the head of this queue.                                                                                                                                                                                                                     |

## Classes:

- java.util.**AbstractQueue**<E> (implements java.util.[Queue](#)<E>)
  - java.util.**PriorityQueue**<E> (implements java.io.[Serializable](#))

# Interface Deque...



public interface **Deque**<E> extends [Queue](#)<E>

IN addition to methods inherited from the interface Queue, some methods are declared:

Summary of Deque methods

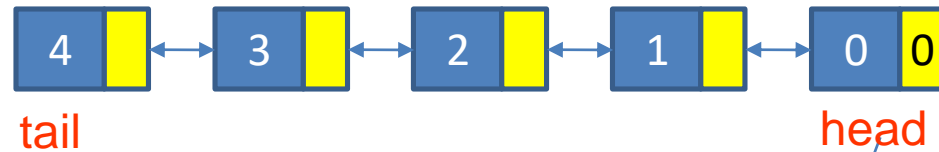
|                | First Element (Head)       |                            | Last Element (Tail)       |                           |
|----------------|----------------------------|----------------------------|---------------------------|---------------------------|
|                | <i>Throws exception</i>    | <i>Special value</i>       | <i>Throws exception</i>   | <i>Special value</i>      |
| <b>Insert</b>  | <code>addFirst(e)</code>   | <code>offerFirst(e)</code> | <code>addLast(e)</code>   | <code>offerLast(e)</code> |
| <b>Remove</b>  | <code>removeFirst()</code> | <code>pollFirst()</code>   | <code>removeLast()</code> | <code>pollLast()</code>   |
| <b>Examine</b> | <code>getFirst()</code>    | <code>peekFirst()</code>   | <code>getLast()</code>    | <code>peekLast()</code>   |

## Classes

java.util.[LinkedList](#)<E> (implements java.lang.[Cloneable](#), java.util.[Deque](#)<E>, java.util.[List](#)<E>, java.io.[Serializable](#))

java.util.[ArrayDeque](#)<E> (implements java.lang.[Cloneable](#), java.util.[Deque](#)<E>, java.io.[Serializable](#))

# Queue/Deque Demo.



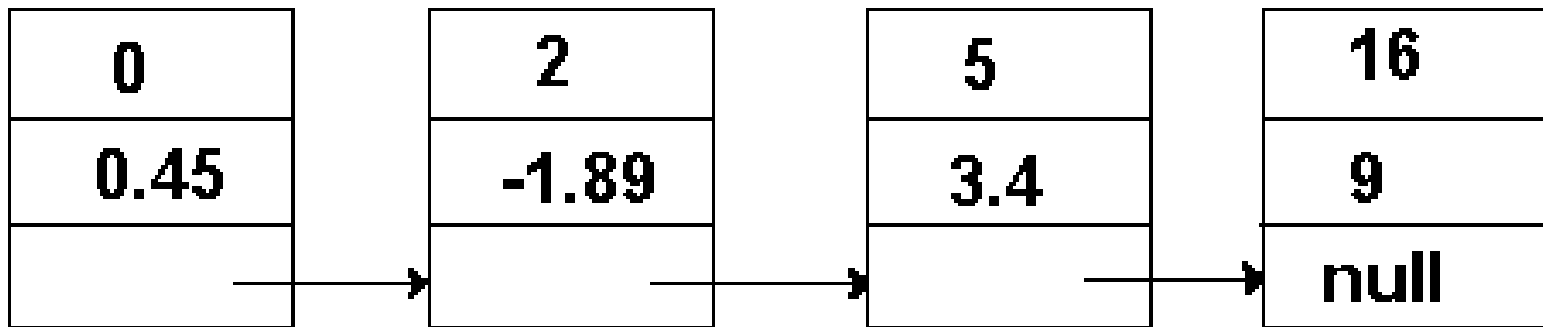
```
import java.util.LinkedList;
public class DequeDemo {
    public static void main(String args[]){
        int N=5;
        // 3 list are the same
        LinkedList list1= new LinkedList();
        LinkedList list2= new LinkedList();
        LinkedList list3= new LinkedList();
        for (int i=0; i<N; i++) {
            list1.add(i); list2.add(i); list3.add(i);
        }
        // Access list1 as a queue
        while(!list1.isEmpty()) System.out.print(list1.remove() + ",");
        System.out.println();
        // Access list2 from it's head
        while(!list2.isEmpty()) System.out.print(list2.removeFirst()+ ",");
        System.out.println();
        // Access list2 from it's tail
        while(!list3.isEmpty()) System.out.print(list3.removeLast()+ ",");
        System.out.println();
    }
}
```

run:

0,1,2,3,4,  
0,1,2,3,4,  
4,3,2,1,0,

# LinkedList

- Constructor:
  - `LinkedList( )`
  - `LinkedList(Collection c)`
- $0.45 - 1.89 x^2 + 3.4 x^5 + 9 x^{16}$



# Example: Polynomial

```
public class Polynomial {  
    private Node first = new Node(0, 0);  
    private Node last  = first;  
    private static class Node {  
        int coef; // coefficient – hệ số  
        int exp;  // Exponent – số mũ  
        Node next;  
        Node(int coef, int exp) {  
            this.coef = coef;  
            this.exp  = exp;  
        }  
    }  
}
```

```
private Polynomial() { }  
    // a * x^b - - 1.89 x2  
    public Polynomial(int coef,  
int exp) {  
        last.next = new Node(coef,  
exp);  
        last = last.next;  
    }
```

```

// return c = a + b
public Polynomial plus(Polynomial b) {
    Polynomial a = this;
    Polynomial c = new Polynomial();
    Node x = a.first.next;
    Node y = b.first.next;
    while (x != null || y != null) {
        Node t = null;
        if(x == null){ t = new Node(y.coef, y.exp); y = y.next; }
        else if(y == null){t = new Node(x.coef, x.exp); x = x.next; }
        else if(x.exp > y.exp){t=new Node(x.coef, x.exp);x = x.next; }
        else if(x.exp < y.exp){t=new Node(y.coef, y.exp); y=y.next; }
        else {
            int coef = x.coef + y.coef;
            int exp  = x.exp;
            x = x.next;
            y = y.next;
            if (coef == 0) continue;
            t = new Node(coef, exp);}
        c.last.next = t;
        c.last = c.last.next;
    }
    return c;
}

```

```

// return c = a * b
public Polynomial multiply(Polynomial b) {
    Polynomial a = this;
    Polynomial c = new Polynomial();
    for(Node x=a.first.next; x!= null; x= x.next)
    {
        Polynomial temp = new Polynomial();
        for (Node y = b.first.next; y!= null; y =
y.next) {
            temp.last.next = new Node(x.coef *
y.coef, x.exp + y.exp);
            temp.last = temp.last.next;
        }
        c = c.plus(temp);
    }
    return c;
}

```



```

public String toString() {
    String s = "";
    for(Node x = first.next; x != null; x = x.next)
    {
        if(x.coef > 0) s = s + " + " + x.coef +
"x^" + x.exp;
        else if (x.coef < 0) s = s + " - " + (-
x.coef) + "x^" + x.exp;}
    return s;
}

public static void main(String[] args) {
    Polynomial zero = new Polynomial(0, 0);
    // 4x^3 + 3x^2 + 1
    Polynomial p1    = new Polynomial(4, 3);
    Polynomial p2    = new Polynomial(3, 2);
    Polynomial p3    = new Polynomial(1, 0);
    Polynomial p     = p1.plus(p2).plus(p3);
}

```

```

// 3x^2 + 5
    Polynomial q1    = new Polynomial(3, 2);
    Polynomial q2    = new Polynomial(5, 0);
    Polynomial q      = q1.plus(q2);

    Polynomial r      = p.plus(q);
    Polynomial s      = p.multiply(q);
    System.out.println("zero(x) = " +
zero);
    System.out.println("p(x) = " + p);
    System.out.println("q(x) = " + q);
    System.out.println("p(x) + q(x) = " + r);
    System.out.println("p(x) * q(x) = " + s);
    }
}

```

# Summary

- The Collections Framework
  - **The *Collection* Super interface and Iteration**
  - **Lists**
  - **Sets**
  - **Maps**
  - **Support Classes**
  - **Collections and Code Maintenance**