List:

* List store duplicates.
* List has index.
* List supports multiple null values.
* List preserve insertion order.

ArrayList:

* ArrayList store duplicates.
* ArrayList has index.
* ArrayList supports multiple null values.
* ArrayList preserve insertion order.
* Underlying datastructure of arraylist is Resizable array

Constructors of ArrayList:

* ArrayList(): it will create an empty arraylist object of initial capacity 10.
* ArrayList(Collection c):Constructs a list containing elements of the specified collection.
* ArrayList(int initialCapacity):Constructs an empty arraylist with the specified initial capacity.

Example for ArrayList:

package com.ty.arraylist;

import java.util.ArrayList;

import java.util.List;

public class ArrayListExample {

public static void main(String[] args) {

List a=new ArrayList();

a.add(4);

a.add(2.3);

a.add('a');

a.add("hello");

a.add(4);

a.add(null);

a.add(null);

System.out.println(a);

System.out.println(a.get(5));

for(int i=0;i<a.size();i++) {

System.out.println(a.get(i));

}

}

}

package com.ty.arraylist;

import java.util.ArrayList;

//objects---predefined objects

public class Example {

public static void main(String[] args) {

ArrayList<Integer> a=new ArrayList<Integer>();

a.add(2);

a.add(8);

a.add(9);

ArrayList<Double> a1=new ArrayList<Double>();

a1.add(2.3);

a1.add(1.4);

}

}

How to sort predefined Objects?

By using Collections.sort(ref variable);

* Collections.sort() internally calls compareTo() .
* compareTo() is property of Comparable interface which is present in java.lang package.
* In every wrapper class compareTo() is overridden.
* To call Collections.sort() we can add only homogenous elements else it throws ClassCastException.
* To call Collections.sort() ,if we add null value in list then it will throw NullPointerException.Hence null values are not allowed during sorting in arraylist.

package com.ty.arraylist;

import java.util.ArrayList;

import java.util.Collections;

public class SortPredefinedObjects {

public static void main(String[] args) {

ArrayList<Integer> a=new ArrayList<>();

a.add(7);

a.add(1);

a.add(6);

a.add(9);

a.add(7);

a.add(6);

a.add(3);

System.out.println(a);

Collections.sort(a);

System.out.println(a);

}

}

package com.ty.arraylist;

//How to store user defined objects

public class Student {

private String name;

private double marks;

private int id;

public Student(String name, double marks, int id) {

this.name = name;

this.marks = marks;

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getMarks() {

return marks;

}

public void setMarks(double marks) {

this.marks = marks;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

@Override

public String toString() {

return "Student [name=" + name + ", marks=" + marks + ", id=" + id + "]";

}

}

package com.ty.arraylist;

import java.util.ArrayList;

public class StudentMainClass {

public static void main(String[] args) {

Student s1=new Student("allen",67.8,1);

Student s2=new Student("smith",32.5,2);

Student s3=new Student("scott",54.3,3);

Student s4=new Student("martin",98.6,4);

ArrayList<Student> stuList=new ArrayList<Student>();

stuList.add(s1);

stuList.add(s2);

stuList.add(s3);

stuList.add(s4);

System.*out*.println(stuList);

for(int i=0;i<stuList.size();i++) {

System.*out*.println(stuList.get(i));

}

}

}

How to sort user defined Objects?

* Create a class which implements Comparable interface.
* Override compareTo() method and specify sorting logic.
* Create an objects of above class and add it to arraylist.
* Call Collections.sort(arraylist ref).

package com.ty.arraylist;

//How to store user defined objects

public class Student implements Comparable<Student>{

private String name;

private double marks;

private int id;

public Student(String name, double marks, int id) {

this.name = name;

this.marks = marks;

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getMarks() {

return marks;

}

public void setMarks(double marks) {

this.marks = marks;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

@Override

public String toString() {

return "Student [name=" + name + ", marks=" + marks + ", id=" + id + "]";

}

@Override

public int compareTo(Student o) {

return this.id-o.id;//sort ascending order

}

}

package com.ty.arraylist;

import java.util.ArrayList;

import java.util.Collections;

public class SortUserDefinedObjects {

public static void main(String[] args) {

Student s1=new Student("allen",67.8,1);

Student s2=new Student("smith",32.5,2);

Student s3=new Student("scott",54.3,3);

Student s4=new Student("martin",98.6,4);

ArrayList<Student> stuList=new ArrayList<Student>();

stuList.add(s1);

stuList.add(s3);

stuList.add(s4);

stuList.add(s2);

System.out.println(stuList);

Collections.sort(stuList);

System.out.println(stuList);

}

}

Vector:

* Vector store duplicates.
* Vector has index.
* Vector supports multiple null values.
* Vector preserve insertion order.
* Underlying datastructure of Vector is Resizable array
* In vector programmer can control the incremental capacity by specifying the incremental capacity.

Constructors of Vector:

* Vector (): it will create an empty Vector object of initial capacity 10.
* Vector (Collection c):Constructs a list containing elements of the specified collection.
* Vector (int initialCapacity):Constructs an empty Vector with the specified initial capacity.
* Vector (int initialCapacity, int incremental capacity):Constructs an empty Vector with the specified initial capacity and incremental capacity.

455

Note: Vector is a thread safe.

* Methods of vector class has been synchronized.

Set:

1. Set do not store duplicate Values.
2. Set stores only one null value.
3. The set do not have an index.
4. The set do not preserve insertion order.

HashSet:

1. HashSet doesn’t store duplicate Values.
2. HashSet stores only one null value.
3. The HashSet doesn’t have an index.
4. The HashSet doesn’t follow insertion order.
5. The underlying data structure of HashSet is Hashtable.

Constructors of HashSet:

* HashSet (): it will create an empty set with initial capcity 16 and load factor 0.75.
* HashSet (Collection c):it will create an empty set containing the elements in the specified collection.
* HashSet (int initialCapacity):it will create an empty set with specified initial capacity and default load factor 0.75.
* HashSet (int initialCapacity, floatLoadFactor):it will create an empty set with specified initial capacity and specified load factor.

package com.ty.set;

import java.util.HashSet;

public class SetExample {

public static void main(String[] args) {

HashSet<Integer> h=new HashSet<>();

h.add(4);

h.add(8);

h.add(4);

h.add(null);

h.add(null);

h.add(2);

System.*out*.println(h);

//Foreach loop

/\*

\* for(Datatype ref:collection ref variable{

\*

\* }

\* \*/

for(Integer i:h) {

System.*out*.println(i);

}

}

}

LinkedHashSet:

1. LinkedHashSet doesn’t store duplicate Values.
2. LinkedHashSet stores only one null value.
3. LinkedHashSet doesn’t have an index.
4. LinkedHashSet preserves insertion order.
5. The underlying data structure of LinkedHashSet is Hashtable and linked list.

Constructors of LinkedHashSet :

* LinkedHashSet (): it will create an empty set with initial capcity 16 and load factor 0.75.
* LinkedHashSet (Collection c):it will create an empty set containing the elements in the specified collection.
* LinkedHashSet (int initialCapacity):it will create an empty set with specified initial capacity and default load factor 0.75.
* LinkedHashSet (int initialCapacity, floatLoadFactor):it will create an empty set with specified initial capacity and specified load factor.

package com.ty.set;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String[] args) {

LinkedHashSet<Object> l=new LinkedHashSet<>();

l.add('h');

l.add(2);

l.add(null);

l.add(null);

l.add(4.5);

l.add(2);

// System.out.println(l);

//

// for(Object o:l) {

// System.out.println(o);

// }

ArrayList<Integer> a=new ArrayList<>();

a.add(2);

a.add(3);

a.add(5);

a.add(2);

a.add(6);

a.add(6);

a.add(6);

System.out.println(a);

HashSet<Integer> h=new HashSet<>(a);

System.out.println(h);

}

}

TreeSet:

1. TreeSet doesn’t store duplicate Values.
2. TreeSet doesn’t stores null value (not even single null value). If we try to store null value it will throw NullPointerException.
3. TreeSet doesn’t have an index.
4. TreeSet doesn’t preserve insertion order.
5. In TreeSet heterogeneous type of data is not allowed.
6. The underlying data structure of TreeSet is BinarySearchTree.

Constructors of TreeSet:

* TreeSet (): it will create an empty treeset object where elements will be inserted according to default natural sorting order.
* TreeSet (Collection c):
* TreeSet (Comparator c):

TreeSetct=new TreeSet(Comparator c);

it will create an empty treeset object where elements will be inserted according to customized sorting order.

If we add Integer or any other wrapper class objects to TreeSet by calling zero argument constructor.

TreeSet<Integer>();

TreeSet<Integer> t=

If we add user defined objects with TreeSet zero argument constructor then we should mandatorily implements Comparable and override compareTo() method explicitly.

It calls compareto() method present in Integer class.

package com.ty.set;

import java.util.Iterator;

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

TreeSet<Integer> t=new TreeSet<>();

t.add(3);

t.add(6);

t.add(1);

System.out.println(t);

for(Integer i:t) {

System.out.println(i);

}

//using iterator

Iterator<Integer> i=t.iterator();

while(i.hasNext()) {

System.out.println(i.next());

}

}

}

package com.ty.set;

import java.util.TreeSet;

public class Employee implements Comparable<Employee> {

String name;

int id;

double sal;

public Employee(String name, int id, double sal) {

super();

this.name = name;

this.id = id;

this.sal = sal;

}

@Override

public String toString() {

return "Employee [name=" + name + ", id=" + id + ", sal=" + sal + "]";

}

public static void main(String[] args) {

Employee e1=new Employee("smith", 101, 3.5);

Employee e2=new Employee("allen", 102,5.0);

Employee e3=new Employee("mark", 103, 4.3);

Employee e4=new Employee("scott", 104, 3.8);

TreeSet<Employee> t=new TreeSet<>();

t.add(e1);

t.add(e2);

t.add(e3);

t.add(e4);

System.*out*.println(t);

}

@Override

public int compareTo(Employee o) {

return this.id-o.id;

}

}

LinkedList: it is a implementation class of list and queue.

* LinkedList store duplicates.
* LinkedList has index.
* LinkedList supports multiple null values.
* LinkedList preserve insertion order.
* Underlying datastructure of LinkedList is Doubly LinkedList.

Constructor:

LinkedList();

LinkedList(Collection c );

Methods of LinkedList:

* pollFirst():- it will get the first element and remove the first element.
* pollLast():- it will get the last element and remove the last element.

Queue: It is an interface which is present in java.util package.

* Queue store duplicates.
* Queue has index.
* Queue supports multiple null values.
* Queue preserve insertion order.

Priority Queue: It is a implementation class for queue interface which is present in java.util package.

1. Priority Queue store duplicate Values.
2. Priority Queue doesn’t stores null value (not even single null value).
3. Priority Queue doesn’t have an index.
4. Priority Queue doesn’t preserve insertion order.
5. Priority queue does not allow heterogeneous data.
6. The underlying data structure of Priority Queue is BinaryHeap.

Constructor:

PriorityQueue();

PriorityQueue(Collection c);

PriorityQueue(Comparator c);

PriorityQueue(int initialCapacity);

PriorityQueue(int initialCapacity, Comparator c);

Note:

* In the priority queue to retrieve the elements we have a method called poll();
* Poll() method will get the element and remove the element from the collection object
* Peek() method is used to get 1st element present.
* Element() method will get the 1st element.

HashMap: Is a implementation class of Map present in java.util package.

Properties:

* HashMap do not store duplicate key.
* HashMap stores duplicate values
* HashMap do not have index.
* Key and values are added into HashMap by using put(key,value) method.
* Values from HashMap is retrieved by using get(key) method.
* If you try to add new value with old key then old value will be re-initialized.
* Only one key can be null.
* Multiple values in HashMap can be null.
* HashMap do not preserve insertion order.
* Underlying data structure of HashMap is HashTable.

package com.oop;

import java.util.HashMap;

import java.util.Set;

public class Test {

public static void main(String[] args) {

HashMap<Integer, String> t = new HashMap<Integer, String>();

t.put(101, "java");

t.put(102, "web");

t.put(103, "sql");

t.put(104, "mannual");

t.put(102, "react");

// System.out.println(t);

//to get data from Map

System.*out*.println(t.containsKey(104));

System.*out*.println(t.containsValue("react"));

Set<Integer> s = t.keySet();

for (Integer i : s) {

System.*out*.println(i+"-->"+t.get(i));

}

}

}

LinkedHashMap: Is a implementation class of Map present in java.util package.

Properties:

* LinkedHashMap do not store duplicate key.
* LinkedHashMap stores duplicate values
* LinkedHashMap do not have index.
* Key and values are added into LinkedHashMap by using put(key,value) method.
* Values from LinkedHashMap is retrieved by using get(key) method.
* If you try to add new value with old key then old value will be re-initialized.
* Only one key can be null.
* Multiple values in LinkedHashMap can be null.
* LinkedHashMap preserve insertion order.
* Underlying data structure of LinkedHashMap is HashTable and LinkedList.

TreeMap: Is a implementation class of Map present in java.util package.

Properties:

* TreeMap do not store duplicate key.
* TreeMap stores duplicate values
* TreeMap do not have index.
* Key and values are added into TreeMap by using put(key,value) method.
* Values from TreeMap is retrieved by using get(key) method.
* If you try to add new value with old key then old value will be re-initialized to the given key.
* Not even single key can be null.
* Multiple values in TreeMap can be null.
* TreeMap do not preserve insertion order.
* Underlying data structure of TreeMap is BinarySearch Tree
* The values in TreeMap sorted according to keys
* The key in TreeMap should be homogeneous.
* The Values in TreeMap can be Heterogeneous.

**Thread:**

* Thread is a pre-defined class present in java.lang package.
* A thread is a independent part of same program which will get its own stack and cpu time for execution.
* Thread are used for performing multi-tasking in program or application.
* Thread creation and execution is costly with respect to memory and cpu time and hence unwanted thread creation should be totally avoided.
* Whenever JVM starts execution it creates 3 threads by default:
  + - 1. Main Thread
      2. Garbage collector
      3. Thread scheduler
* By default all the programs in java will be executed in main thread.
* In java Thread can be created in 2 ways.
* By extending thread class
* By implementing Runnable interface

**Multi-Threading:**

* Two or more thread are accessing same object is called as multi-threading.
* **Problem:** Whenever multiple threads trying to access same object at the same time is called as **race condition**
* Race condition always leads to in-consistant data.
* Race condition can be overcome by using thread synchronization.

**Thread Synronization:**

* Thread synchronization is executing the threads in sequential order where one thread will execute only after the other thread complete the execution

**Thread safe class:**

* If a class consists of synchronized methods then the class is known as thread safe class.
* To perform all multi-threading operations we have to use thread safe classes.

**Thread synchronization can be achieved in 2ways**

* Using synchronized methods
* Using synchronized blocks

**Synchronized methods:**

* Any method which are declared by using synchronized keyword is known ad synchronized method.
* If a thread calls the synchronized method of the class then the thread will acquire the lock on given object and other threads will not be able to access the same object at the same time.
* Hence race condition is avoided

**Synchronized blocks:**

* Synchronized blocks will lock the given object until the thread complete its execution using the given object. This prevents other thread accessing the same object at the same time and hence race condition is avoided.

**Syntax:**

**synchronized(Object ref)**

**{**

Statements…

**}**

Synchronized blocks are used to perform thread safe operation in those objects whose class is not thread safe.

**Note:**

* Synchronized methods are used to make a class as thread safe.
* Synchronized blocks are used to make object as thread safe.