

Object - Oriented Programming 2

Week 13. Legacy Classes, HashTable Class, Collection Class, Comparable Interface, and Comparator Interface

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

[Email: josebulinda@gmail.com](mailto:josebulinda@gmail.com)

jose@kumiuniversity.ac.ug

Agenda

1. Legacy Classes
2. HashTable Class,
3. Collection Class
4. Comparable interface,
5. Comparator Interface

Legacy Classes

Legacy Classes and Legacy interface

Legacy classes are classes that had the features of collection framework before collection was added. Collection frame was added on Java 2 SE. However, when the collection was added these classes were re-engineered.

All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them

Legacy Classes and Legacy interface+

The following are the legacy classes defined by **java.util** package

1.Dictionary

2.HashTable

3.Properties

4.Stack

5.Vector

NOTE: All the legacy classes are synchronized

There is only one legacy interface called **Enumeration**.

Dictionary class

- 1.Dictionary is an abstract class.
- 2.It represents a key/value pair and operates much like Map.
- 3.Although it is not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map class, which reason we will not talk much about it here.

HashTable Class

HashTable class

HashTable class is one of the legacy classes that was reengineered in JDK 5 when Java collection framework was introduced.

Important features

1. Like HashMap, HashTable also stores key/value pair. However neither **keys** nor **values** can be **null**.
2. It contains unique elements.
3. It doesn't allow null key or value.
4. The initial default capacity of Hashtable is 11.
5. is synchronized while HashMap is not.
6. Hashtable has following four constructors

Difference between HashMap and Hashtable

HashTable	HashMap
Hashtable class is synchronized.	HashMap is not synchronized.
Because of Thread-safe, HashTable is slower than HashMap	HashMap works faster.
Neither key nor values can be null	Both key and values can be null
Order of table remain constant over time.	does not guarantee that order of map will remain constant over time.

Java Collection Framework

HashTable

HashTable class stores elements in key-value pair. It does not allow null key and null values. It is synchronized version of HashMap.

It extends Dictionary class and implements Map interface.

Declaration

```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable
```

HashTable class Constructors

Constructor	Description
Hashtable()	creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	accepts an integer parameter and creates a hash table that contains a specified initial capacity.
Hashtable(int capacity, float loadFactor)	is used to create a hash table having the specified initial capacity and loadFactor.
Hashtable(Map<? extends K, ? extends V> t)	creates a new hash table with the same mappings as the given Map.

HashTable Class Methods

Method	Description
void clear()	emptys the hash table.
Object clone()	returns a shallow copy of the Hashtable.
V compute (K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	computes a mapping for the specified key and its current mapped value.
V computeIfAbsent (K key, Function<? super K, ? extends V> mappingFunction)	computes its value using the given mapping function.
Enumeration elements()	returns an enumeration of the values in the hash table.

HashTable Class Methods+

Method	Description
Set<Map.Entry<K,V>> entrySet()	returns a set view of the mappings contained in the map.
boolean equals (Object o)	compares the specified Object with the Map.
void forEach (BiConsumer<? super K,? super V> action)	performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V getOrDefault (Object key, V defaultValue)	returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode ()	returns the hash code value for the Map

HashTable Class Methods++

Method	Description
Enumeration<K> keys()	returns an enumeration of the keys in the hashtable.
Set<K> keySet()	returns a Set view of the keys contained in the map.
V merge (K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	if the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V put (K key, V value)	inserts the specified value with the specified key in the hash table.
void putAll (Map<? extends K,? extends V> t))	is used to copy all the key-value pair from map to hashtable.

HashTable Class Methods+++

Method	Description
V putIfAbsent (K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
boolean remove (Object key, Object value)	removes the specified values with the associated specified keys from the hashtable.
V replace (K key, V value)	replaces the specified value for a specified key.
String toString ()	returns a string representation of the Hashtable object.
Collection values ()	returns a collection view of the values contained in the map.

HashTable Class Methods++++

Method	Description
boolean contains (Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsValue (Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsKey (Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty ()	This method returns true if the hash table is empty; returns false if it contains at least one key.

HashTable Class Methods+++++

Method	Description
protected void rehash()	is used to increase the size of the hash table and rehashes all of its keys.
V get (Object key)	This method returns the object that contains the value associated with the key.
V remove (Object key)	is used to remove the key and its value. This method returns the value associated with the key.
int size ()	returns the number of entries in the hash table.

(Studytonight.com)


Creating and Adding Elements into HashTable

In this example, we will create a HashTable that takes elements of string and integer type pair. To create HashTable we will utilize the following code.

```
Hashtable<String,Integer> ht = new Hashtable<String,Integer>();
```

To insert elements into the HashTable, we are using `put()` method that adds new elements. It takes two argument: **First** is key and **Second** is value.

```
ht.put("A", 700);  
ht.put("B", 200);
```



We have just added two elements into the HashTable called **ht**.

Creating and Adding Elements into HashTable code

```
import java.util.*;
class HTPro1 {
    public static void main(String args[]) {
        // Creating Hashtable
        Hashtable<String,Integer> ht = new
        Hashtable<String,Integer>();
        // Adding elements
        ht.put("A",700); ht.put("N",677); ht.put("T",700);
        ht.put("B",200); ht.put("C",799);
        // Displaying Hashtable
        System.out.println(ht);
    } }
```

Creating and Adding Elements into HashTable code and output

```
1  package CollectionClass;
2  import java.util.*;
3  class HTPro1 {
4      public static void main(String args[]) {
5          // Creating Hashtable
6          Hashtable<String,Integer> ht = new Hashtable<String,Integer>();
7          // Adding elements
8          ht.put("A",700); ht.put("N",677); ht.put("T",700);
9          ht.put("B",200); ht.put("C",799);
10         // Displaying Hashtable
11         System.out.println(ht);
12     } }
```

run:

{A=700, T=700, C=799, N=677, B=200}

Example: Program to Search for a key or value in a HashTable

HashTable provides various methods such as `contains()`, `containsKey()` etc to search for an element in the HashTable. `contains()` method search for specified value while `containsKey()` method search for specified key. We will therefore add the following code into our program above to be able to search for items in our HashTable created above.

```
// Search for a value
boolean val = ht.contains(400);
System.out.println("is 400 present: "+val);
// Search for a key
val = ht.containsKey("K");
System.out.println("is K present: "+val);
```

Example: Program to Search for a key or value in a HashTable+Code

```
import java.util.*;
class HTPro1 {
    public static void main(String args[]) {
        // Creating Hashtable
        Hashtable<String,Integer> ht = new Hashtable<String,Integer>();
        // Adding elements
        ht.put("A",700); ht.put("N",677); ht.put("T",400);
        ht.put("B",200); ht.put("C",799);
        // Displaying Hashtable
        System.out.println(ht);
        // Search for a value
        boolean val = ht.contains(400);
        System.out.println("is 400 present: "+val);
        // Search for a key
        val = ht.containsKey("K");
        System.out.println("is K present: "+val);
    } }
```

Example: Program to Search for a key or value in a HashTable+Code and output

```
1  package CollectionClass; import java.util.*;
2  class HTPro1 {
3      public static void main(String args[]) {
4          // Creating Hashtable
5          Hashtable<String,Integer> ht = new Hashtable<String,Integer>();
6          // Adding elements
7          ht.put("A",700); ht.put("N",677); ht.put("T",400);
8          ht.put("B",200); ht.put("C",799);
9          // Displaying Hashtable
10         System.out.println(ht);
11         // Search for a value
12         boolean val = ht.contains(400);
13         System.out.println("is 400 present: "+val);
14         // Search for a key
15         val = ht.containsKey("K");
16         System.out.println("is K present: "+val);
17     } }
```

run:

```
{A=700, T=400, C=799, N=677, B=200}
is 400 present: true
is K present: false
```

Adding Elements to Hashtable

To insert elements into the HashTable, we have use put() method that adds new elements. It takes two argument: first is key and second is value.

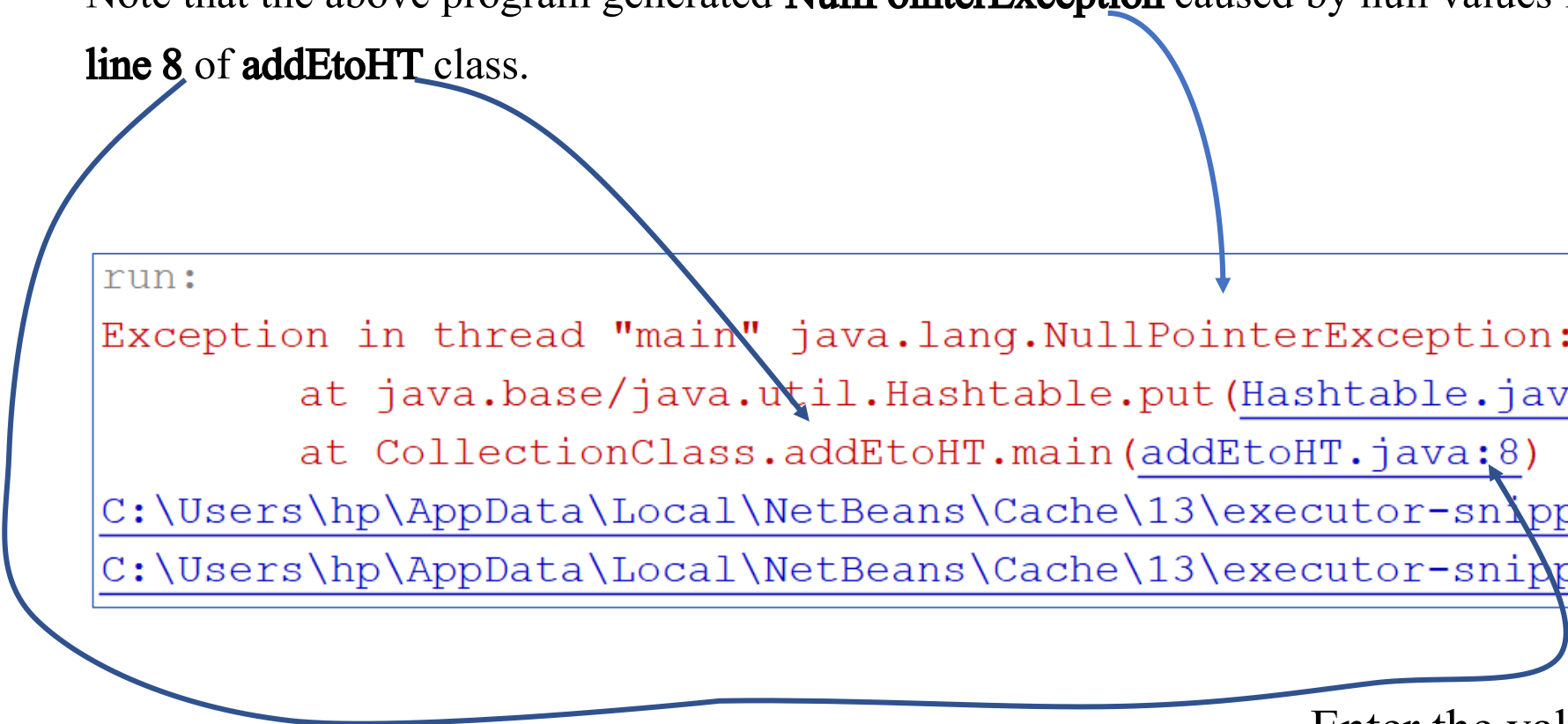
Note: This class does not allow Null values. In the example below, we will try to insert elements into a Hastable with one null value.

Adding elements into a HashTable with null values

```
import java.util.*;
class addEtoHT {
public static void main(String args[]) {
    Hashtable<String,Integer> ht = new
    Hashtable<String,Integer>();
    ht.put("A",100);ht.put("B",500);
    ht.put("C",300); ht.put("D",400);
    ht.put(null, 0); // error: no null allowed //
    Displaying Hashtable
        System.out.println(ht);
}
}
```

Adding elements into a HashTable with null value-output error.

Note that the above program generated **NullPointerException** caused by null values in the HashTable list on **line 8** of **addEtoHT** class.



```
run:
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "C
    at java.base/java.util.Hashtable.put(Hashtable.java:481)
    at CollectionClass.addEtoHT.main(addEtoHT.java:8)
C:\Users\hp\AppData\Local\NetBeans\Cache\13\executor-snippets\run.xml:111:
C:\Users\hp\AppData\Local\NetBeans\Cache\13\executor-snippets\run.xml:68: 5
```

Enter the value to solve this problem

Properties class

Properties class

Properties class is one of the legacy classes that supported some of the functionalities of collection framework before the said framework was added to java. The following are some of the important features of this class.

- 1.**Properties** class extends **Hashtable** class.
- 2.It is used to maintain list of value in which both key and value are **String**
- 3.**Properties** class define two constructor

```
Properties() //creates a Properties object that has no default values  
Properties(Properties propdefault) // creates an object that uses  
propdefault for its default values.
```

Properties class+

4. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

Note: In both cases, the property list is empty

5. In Properties class, you can specify a default property that will be returned if no value is associated with a certain key.

Example of Properties class program

```
import java.util.*;
public class PropTest {
    public static void main(String[] args) {
        Properties pt = new Properties();
        pt.put("Java", "James Gosling");
        pt.put("C++", "Bjarne Stroustrup");
        pt.put("C", "Dennis Ritchie");
        pt.put("C#", "Microsoft Inc.");
        Set< ?> creator = pt.keySet();
        for(Object ob : creator) {
            System.out.println(ob+" was created by "+
                pt.getProperty((String)ob) );
        }
    }
}
```

Example of Properties class program+code and output

```
1  package CollectionClass;
2  import java.util.*;
3  public class PropTest {
4      public static void main(String[] args) {
5          Properties pt = new Properties();
6          pt.put("Java", "James Ghosling");
7          pt.put("C++", "Bjarne Stroustrup");
8          pt.put("C", "Dennis Ritchie");
9          pt.put("C#", "Microsoft Inc.");
10         Set< ?> creator = pt.keySet();
11         for(Object ob: creator) {
12             System.out.println(ob+" was created by "+
13                 pt.getProperty((String)ob) );
14         }
15     } }
```

run:
C# was created by Microsoft Inc.
Java was created by James Ghosling
C++ was created by Bjarne Stroustrup
C was created by Dennis Ritchie

Stack class

Stack class is a legacy class.

1. Stack class extends Vector.
2. It follows last-in, first-out principle for the stack elements.
3. It defines only one default constructor

```
Stack() //This creates an empty stack
```

4. If you want to put an object on the top of the stack, call **push()** method. If you want to remove and return the top element, call **pop()** method. An EmptyStackException is thrown if you call **pop()** method when the invoking stack is empty.
5. You can use **peek()** method to return, but not remove the top object. The **empty()** method returns true if nothing is on the stack. The **search()** method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack.

Example of Stack program

```
import java.util.*;
class StackPro {
    public static void main(String args[]) {
        Stack sk = new Stack();
        sk.push(11); sk.push(22);
        sk.push(33); sk.push(44);
        sk.push(55);
        Enumeration e1 = sk.elements();
        while(e1.hasMoreElements())
            System.out.print(e1.nextElement()+" ");
        sk.pop(); sk.pop();
        System.out.println("\nAfter popping out two elements");
        Enumeration e2 = sk.elements();
        while(e2.hasMoreElements())
            System.out.print(e2.nextElement()+" ");
    } }
```

Example of Stack program code and output

```
1  package CollectionClass;
2  import java.util.*;
3  class StackPro {
4      public static void main(String args[]) {
5          Stack sk = new Stack();
6          sk.push(11); sk.push(22);
7          sk.push(33); sk.push(44);
8          sk.push(55);
9          Enumeration e1 = sk.elements();
10         while(e1.hasMoreElements())
11             System.out.print(e1.nextElement()+" ");
12         sk.pop(); sk.pop();
13         System.out.println("\nAfter popping out two elements");
14         Enumeration e2 = sk.elements();
15         while(e2.hasMoreElements())
16             System.out.print(e2.nextElement()+" "); System.out.println();
17     } }
```

Note: pop() has deleted two elements since it was called twice.

run:
11 22 33 44 55
After popping out two elements
11 22 33

Vector class

Vector is one of the legacy classes

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. There are two differences between **Vector** and **ArrayList**; **First**, Vector is synchronized while ArrayList is not, and **Second**, it contains many legacy methods that are not part of the Collections Framework.
3. With the release of JDK 5, Vector also implements Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the for-each loop.

Vector class Methods

Vector defines several legacy methods. Lets see some important legacy methods defined by **Vector** class.

Method	Description
void addElement (E element)	adds element to the Vector
E elementAt (int index)	returns the element at specified index
Enumeration elements ()	returns an enumeration of element in vector
E firstElement ()	returns first element in the Vector
E lastElement ()	returns last element in the Vector
void removeAllElements ()	removes all elements of the Vector

Vector class constructors


Vector class has following 4 constructor as seen below.

```
Vector() //This creates a default vector, which has an  
initial size of 10.  
Vector(int size) //This creates a vector whose initial  
capacity is specified by size.  
Vector(int size, int incr) //This creates a vector whose  
initial capacity is specified by size and whose increment  
is specified by incr. The increment specifies the number of  
elements to allocate each time when a vector is resized for  
addition of objects.  
Vector(Collection c) //This creates a vector that contains  
the elements of collection c.
```

Example of Vector Program

```
import java.util.*;
public class TestVector {
    public static void main(String[] args) {
        Vector<Integer> ve = new Vector<Integer>();
        ve.add(70); ve.add(20); ve.add(98);
        ve.add(40); ve.add(50); ve.add(60);
        Enumeration<Integer> en = ve.elements();
        while(en.hasMoreElements()) {
            System.out.println(en.nextElement()+" ");
        }
    }
}
```

Example of Vector Program code and output

```
1  package CollectionClass;
2
3  import java.util.*;
4  public class TestVector {
5      public static void main(String[] args) {
6           Vector<Integer> ve = new Vector<Integer>();
7          ve.add(70); ve.add(20); ve.add(98); ve.add(40);
8          ve.add(50); ve.add(60);
9          Enumeration<Integer> en = ve.elements();
10         while(en.hasMoreElements()) {
11             System.out.print(en.nextElement()+" ");
12         }
13     } }
```

run:
70 20 98 40 50 60

Collection Class

Collection Class

is designed to provide methods for searching, sorting, copying etc. It consists exclusively of built-in static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections.

This class is located into **java.util** package. The declaration of this class is given below.

Declaration

```
public class Collections extends Object
```

It inherits Object class and all the methods of this class throw a **NullPointerException** if the object is null.

Collections Methods

Method	Description
addAll()	adds all of the specified elements to the specified collection.
binarySearch()	searches the list for the specified object and returns their position in a sorted list.
copy()	copies all the elements from one list into another list.
disjoint()	returns true if the two specified collections have no elements in common.
emptyEnumeration()	fetches an enumeration that has no elements.
emptyIterator()	fetches an Iterator that has no elements.

Collections Methods+++

Method	Description
<code>emptyList()</code>	fetches a List that has no elements.
<code>emptyListIterator()</code>	fetches a List Iterator that has no elements.
<code>emptyMap()</code>	returns an empty map which is immutable.
<code>emptyNavigableMap()</code>	returns an empty navigable map which is immutable.
<code>emptyNavigableSet()</code>	returns an empty navigable set which is immutable in nature.
<code>emptySet()</code>	returns the set that has no elements.

Collections Methods++

Method	Description
emptySortedMap()	returns an empty sorted map which is immutable.
emptySortedSet()	is used to get the sorted set that has no elements.
enumeration()	is used to get the enumeration over the specified collection.
fill()	is used to replace all of the elements of the specified list with the specified elements.
list()	is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration.
max()	is used to get the maximum value of the given collection.

Collections Methods++

Method	Description
min()	is used to get the minimum value of the given collection.
nCopies()	is used to get an immutable list consisting of n copies of the specified object.
replaceAll()	is used to replace all occurrences of one specified value in a list with the other specified value.
reverse()	is used to reverse the order of the elements in the specified list.
reverseOrder()	is used to get the comparator that imposes the reverse of the natural ordering on a collection.
rotate()	is used to rotate the elements in the specified list by a given distance.

Collections Methods+

Method	Description
shuffle()	is used to randomly reorders the specified list elements using a default randomness.
sort()	is used to sort the elements presents in the specified list of collection in ascending order.
swap()	is used to swap the elements at the specified positions in the specified list.
synchronizedCollection()	is used to get a synchronized (thread-safe) collection backed by the specified collection.
synchronizedList()	is used to get a synchronized (thread-safe) collection backed by the specified list.
synchronizedMap()	is used to get a synchronized (thread-safe) map backed by the specified map.

Collections Methods+

Method	Description
synchronizedNavigableMap()	is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map.
synchronizedNavigableSet()	is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set.
synchronizedSet()	is used to get a synchronized (thread-safe) set backed by the specified set.
synchronizedSortedMap()	is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map.
synchronizedSortedSet()	is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set.
synchronizedNavigableMap()	is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map.

Example: Finding min and max elements and sorting elements

The collections class provides two methods `max()` and `min()` that can be used to fetch max and min values from a collection. Therefore to do this, we will have to include the following lines of code into our program called MinMax.

```
int min = Collections.min(list); // Find max element
int max = Collections.max(list); // Displaying data
System.out.println("Minimum element : "+ min);
System.out.println("Maximum element : "+ max);
```


Example: Finding min and max elements and sorting elements + code

```
import java.util.*;
public class MinMax {
    public static void main(String a[]) { // Creating List
        ArrayList<Integer> list = new ArrayList<>();
        list.add(32); list.add(45); list.add(66);          list.add(22);
        list.add(10); list.add(54);
        System.out.println(list); // Sorting list
        Collections.sort(list); // Displaying sorted list
        int min = Collections.min(list); // Find max element
        int max = Collections.max(list); // Displaying data
        System.out.println("Minimum element : "+ min);
        System.out.println("Maximum element : "+ max);
        System.out.println("Sorted List\n"+list);
    }
}
```

Example: Finding min and max elements and sorting elements + code and output

```
1  package CollectionClass; import java.util.*;
2  public class MinMax {
3      public static void main(String a[]){ // Creating List
4          ArrayList<Integer> list = new ArrayList<>();
5          list.add(32); list.add(45); list.add(66);
6          list.add(22); list.add(10); list.add(54);
7          System.out.println(list); // Sorting list
8          Collections.sort(list); // Displaying sorted list
9          int min = Collections.min(list); // Find max element
10         int max = Collections.max(list); // Displaying data
11         System.out.println("Minimum element : "+ min);
12         System.out.println("Maximum element : "+ max);
13         System.out.println("Sorted List\n"+list);
14     }
15 }
```

run:

```
[32, 45, 66, 22, 10, 54]
Minimum element : 10
Maximum element : 66
Sorted List
[10, 22, 32, 45, 54, 66]
```

Swapping Elements and reversing the list

Collections class provides built-in swap method that can be used to swap elements from one position to another in a collection. The `swap()` method takes three arguments: **First** is reference of object, **Second** is index of first elements and **Third** is index of second elements to be swapped. See the below example.

```
Collections.swap(list, 0, 4); // 10 is swapped with 32 in  
progam MinMax class.  
System.out.println("List after swapping : "+ list);
```

Collections class provides a static method `reverse()` that is used to get a collection in reverse order. In the below example, we are getting list in reverse order using the `reverse()` method.

```
Collections.reverse(list);  
System.out.println("List in reverse order "+list);
```

Swapping Elements and reversing the list+code

```
import java.util.*;
public class swapReverse {
    public static void main(String a[]){ // Creating List
        ArrayList<Integer> list = new ArrayList<>();
        list.add(32); list.add(45); list.add(66);          list.add(22);
        list.add(10); list.add(54);
        System.out.println("List before swap\n"+list);
        Collections.swap(list, 0, 4);
        System.out.println("List after swapping : "+ list);
        Collections.sort(list); // Displaying sorted list
        System.out.println("Sorted List\n"+list);
        Collections.reverse(list);
        System.out.println("List in reverse order "+list);
    } }
```

Swapping Elements and reversing the list+code and output

```
1 package CollectionClass; import java.util.*;  
2 public class swapReverse {  
3     ArrayList<Integer> main(String a[]){ // Creating List  
4         list = new ArrayList<>();  
5         list.add(32); list.add(45); list.add(66);  
6         list.add(22); list.add(10); list.add(54);  
7         System.out.println("List before swap\n"+list);  
8         Collections.swap(list, 0, 4);  
9         System.out.println("List after swapping : "+ list);  
10        Collections.sort(list); // Displaying sorted list  
11        System.out.println("Sorted List\n"+list);  
12        Collections.reverse(list);  
13        System.out.println("List in reverse order "+list);  
14    }  
}
```

```
run:  
List before swap  
[32, 45, 66, 22, 10, 54]  
List after swapping : [10, 45, 66, 22, 32, 54]  
Sorted List  
[10, 22, 32, 45, 54, 66]  
List in reverse order [66, 54, 45, 32, 22, 10]
```

Comparator Interface

Comparator Interface

In Java, Comparator interface is used to order(sort) the objects in the collection in your own way. It gives you the ability to decide how elements will be sorted and stored within collection and map.

Comparator Interface defines **compare()** method. This method has two parameters.

This method compares the two objects passed in the parameter. It returns **0** if two objects are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is returned. The method can throw a **ClassCastException** if the type of object are not compatible for comparison.

Rules for using Comparator interface

1. if you want to sort the elements of a collection, you need to implement Comparator interface.
2. If you do not specify the type of the object in your Comparator interface, then, by default, it assumes that you are going to **sort the objects** of type **Object**. Thus, when you override the compare() method, you will need to specify the type of the parameter as Object only.
3. If you want to sort the user-defined type elements, then while implementing the Comparator interface, you need to specify the user-defined type generically. If you do not specify the user-defined type while implementing the interface, then by default, it assumes Object type and you will not be able to compare the user-defined type elements in the collection

Rules for using Comparator interface

For Example

If you want to sort the elements according to roll number, defined inside the class Student, then while implementing the Comparator interface, you need to mention it generically as follows:

```
class MyComparator implements Comparator<Student>{ }
```

Then it assumes, by default, data type of the compare() method's parameter to be Object, and hence you will not be able to compare the Student type(user-defined type) objects.

Example program to compare objects

In this example, we will create three classes; Student, deCompare and CompareTest (with main method)

```
class Student{
    int roll;
    String name;
    Student(int r,String n) {
        roll = r;
        name = n;
    }
    public String toString(){
        return roll+" "+name;
    }
}
```

Comparing two objects using compare() method of Comparator interface:

This class defines the comparison logic for Student class based on their roll. Student object will be sorted in ascending order of their roll.

```
import java.util.*;
class deCompare implements Comparator<Student> {
public int compare(Student s1, Student s2) {
    if(s1.roll == s2.roll) {
        return 0;
    }
    else if(s1.roll > s2.roll) {
        return 1;
    }
    else {return -1; }
} }
```

Comparison Execution using a class called CompareTest

Now let's create an execution class called CompareTest with main() function, to run the two classes above.

```
public class CompareTest {  
    public static void main(String[] args) {  
        TreeSet< Student> ts = new TreeSet< Student>(new  
            deCompare());  
        ts.add(new Student(100, "Amoiti"));  
        ts.add(new Student(300, "Caroline"));  
        ts.add(new Student(200, "Solume"));  
        System.out.println(ts);  
    }  
}
```

Output

When roll numbers of two or more elements are the same, then one element will be omitted. However, when all elements are having unique roll numbers then all elements are outputted. This is because duplicates are not allowed here.

NO Duplicate roll=100;

```
1 package CollectionClass;
2 import java.util.*;
3 public class CompareTest {
4     public static void main(String[] args) {
5         TreeSet< Student> ts = new TreeSet< Student>(new deCompare());
6         ts.add(new Student(100, "Amoite"));
7         ts.add(new Student(300, "Caroline"));
8         ts.add(new Student(200, "Solume"));
9         System.out.println(ts);
10    }
11 }
12
13 }
```

run:

[100 Amoite, 200 Solume, 300 Caroline]

Note Duplicated roll=100;

```
1 package CollectionClass;
2 import java.util.*;
3 public class CompareTest {
4     public static void main(String[] args) {
5         TreeSet< Student> ts = new TreeSet< Student>(new deCompare());
6         ts.add(new Student(100, "Amoite"));
7         ts.add(new Student(100, "Caroline"));
8         ts.add(new Student(200, "Solume"));
9         System.out.println(ts);
10    }
11 }
```

run:

[100 Amoite, 200 Solume]

Important Note

As you can see in the output, Student objects are stored in ascending order of their **roll**.

Note:

1. When we are sorting elements in a collection using Comparator interface, we need to pass the class object that implements Comparator interface.
2. To sort a TreeSet collection, this object needs to be passed in the constructor of TreeSet.
3. If any other collection, like ArrayList, was used, then we need to call sort method of Collections class and pass the name of the collection and this object as a parameter.

For example, If the above program used ArrayList collection, the public class CompareTest would be as follows:

Example of Sorting elements of other collection classes other than TreeSet e.g. ArrayList code

```
public class CompareArraySort {  
    public static void main(String[] args) {  
        ArrayList< Student> ts = new ArrayList< Student>();  
        ts.add(new Student(100, "Amo it"));  
        ts.add(new Student(300, "Caroline"));  
        ts.add(new Student(200, "Solume"));  
        Collections.sort(ts, new deCompare()); /*passing the  
name of the ArrayList and the object of the class that  
implements Comparator in a predefined sort() method in  
Collections class*/  
        System.out.println(ts);  
    }  
}
```

Example of Sorting elements of other collection classes other than TreeSet e.g. ArrayList code and output

```
1  package CollectionClass;
2  import java.util.*;
3  public class CompareArraySort {
4      public static void main(String[] args) {
5          ArrayList< Student> ts = new ArrayList< Student>();
6          ts.add(new Student(100, "Amo it"));
7          ts.add(new Student(300, "Caroline"));
8          ts.add(new Student(200, "Solume"));
9          Collections.sort(ts, new deCompare());
10         System.out.println(ts);
11     }
12 }
```

run:

```
[100 Amo it, 200 Solume, 300 Caroline]
```


Comparable Interface

Comparable Interface

is a member of collection framework which is used to compare objects and sort them according to the natural order.

The natural ordering refers to the behavior of **compareTo()** method which is defined into Comparable interface. Its sorting technique depends on the type of object used by the interface. If object type is string then it sorts it Lexicographically.

If object type is wrapper class object like: integer or list then it sorts according to their values.

Comparable Interface+

if object type is custom object like: user defined object then sorts according to the defined compareTo() method.

Classes that implements this interface can be sorted automatically by calling Collections.sort() method. Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

Declaration

```
public interface Comparable<T>
```

Comparable Method

This interface contains single method `compareTo()` that is given below.

1. It compares object with the specified object for order.
2. compares the current object with the provided object. This function is already implemented for default wrapper classes and primitive data types but, this method also needs to be implemented for user-defined classes.
3. It returns positive integer, if the current object is greater than the provided object.
4. If the current object is less than the provided object then it returns negative integer.
5. If the current object is equal to the provided object then it returns zero.
6. This method returns `NullPointerException`, if the specified object is null and `ClassCastException` if the specified object's type prevents it from being compared to this object..

Example program for Sorting list

Lets take an example to sort an ArrayList that stores integer values. We are using sort() method of Collections class that sort those object which implements Comparable interface. Since integer wrapper class implements Comparable so we are able to get sorted objects.

Example program for Sorting list+

```
import java.util.*;
public class CompSorter {
    public static void main(String a[]) { // Creating List
        ArrayList<Integer> list = new ArrayList<>();
        list.add(32); list.add(45); list.add(66);
        list.add(22); list.add(10); list.add(54);
        // Displaying list
        System.out.println(list); // Sorting list
        Collections.sort(list); // Displaying sorted list
        System.out.println("Sorted List\n"+list);
    }
}
```

Example program for Sorting list+Code and output

```
1  package CollectionClass;
2  import java.util.*;
3  public class CompSorter {
4      public static void main(String a[]){ // Creating List
5          ArrayList<Integer> list = new ArrayList<>();
6          // Adding elements
7          list.add(32); list.add(45); list.add(66); list.add(22);
8          list.add(10); list.add(54);
9          // Displaying list
10         System.out.println(list); // Sorting list
11         Collections.sort(list);
12         // Displaying sorted list
13         System.out.println("Sorted List : "+list);
14     } }
```

run:

[32, 45, 66, 22, 10, 54]

Sorted List

[10, 22, 32, 45, 54, 66]

Assignment

Sorting String objects

1. While sorting string objects, the comparable sorts it lexicographically. It means a dictionary like sorting order. Write a program that will sort the following ArrayList called lister.

```
lister.add("D"); lister.add("L");  
lister.add("A"); lister.add("Z");  
lister.add("C");
```

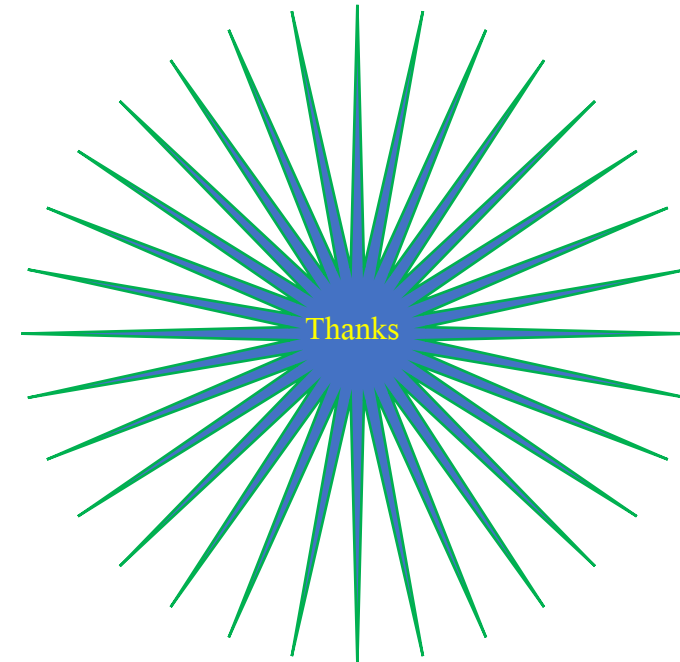

Assignment

2. Having realized that HashTable class does not allow either Null Key or Null values, Write a program that will attempt to insert null key and null value into a HashTable called ErrorT. Run the code to view the output.
3. Write a program that will be able to traverse through a HashTable called NT containing the following elements. {A=10, B=34, C=34, KT=399} using for-each- loop method.

Summary

1. Legacy Classes
2. HashTable Class,
3. Collection Class
4. Comparable interface,
5. Comparator Interface

Thank you for
Listening



References

Accessing a Java collection using iterators. Studytonight.com. (n.d.). Retrieved November 9, 2022, from <https://www.studytonight.com/java/iterator-in-collection.php>

Java Collection Classes. Studytonight.com.(n.d.). Retrieved November 12, 2022, from <https://www.studytonight.com/java/collections-in-collection-framework.php>

Java comparable interface. Studytonight.com. (n.d.). Retrieved November 12, 2022, from <https://www.studytonight.com/java/comparable-in-collection-framework.php>

Comparator interface - java collections. Studytonight.com. (n.d.). Retrieved November 12, 2022, from <https://www.studytonight.com/java/comparators-interface-in-java.php>

Legacy classes - java collections. Studytonight.com. (n.d.). Retrieved November 12, 2022, from <https://www.studytonight.com/java/legacy-classes-and-interface.php>

Java Collection Framework Hashtable. Studytonight.com. (n.d.). Retrieved November 12, 2022, from <https://www.studytonight.com/java/hashtable-in-collection-framework.php>

<https://www.c-sharpcorner.com/article/legacy-classes-and-legacy-interface-of-collections-api/>