Chapter 16
Java Collection and Generics

The division of the chapter is as follows:

Chap 16 Java Collections and Generics

Introduction:

- Can you imagine trying to write object-oriented applications without using data structures like hash tables or linked lists?
- What would you do when you needed to maintain a sorted list.
- Obviously you can do it yourself;
- But with the kind of schedules programmers are under today, it's almost too painful to consider.

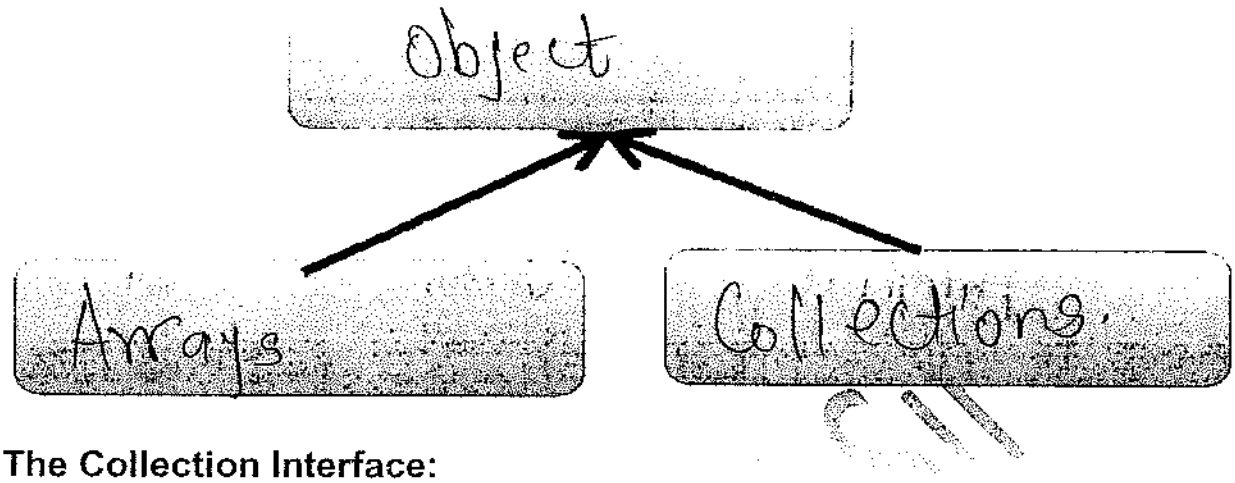
Collections Framework:

- The Collections Framework in Java, gives you
  - lists,
  - sets,
  - maps, and
  - queues
- to satisfy most of your coding needs.
- They've been tried and tested.
- The Collections Framework in the java.util package is loaded with interfaces and utilities.

What all can we do with Collections ?

- There are a few basic operations you'll normally use with collections:
- Add objects to the collection.
- Remove objects from the collection.
- Find out if an object (or group of objects) is in the collection.
- Retrieve an object from the collection (without removing it).
- Iterate through the collection, looking at each element (object) one after another.

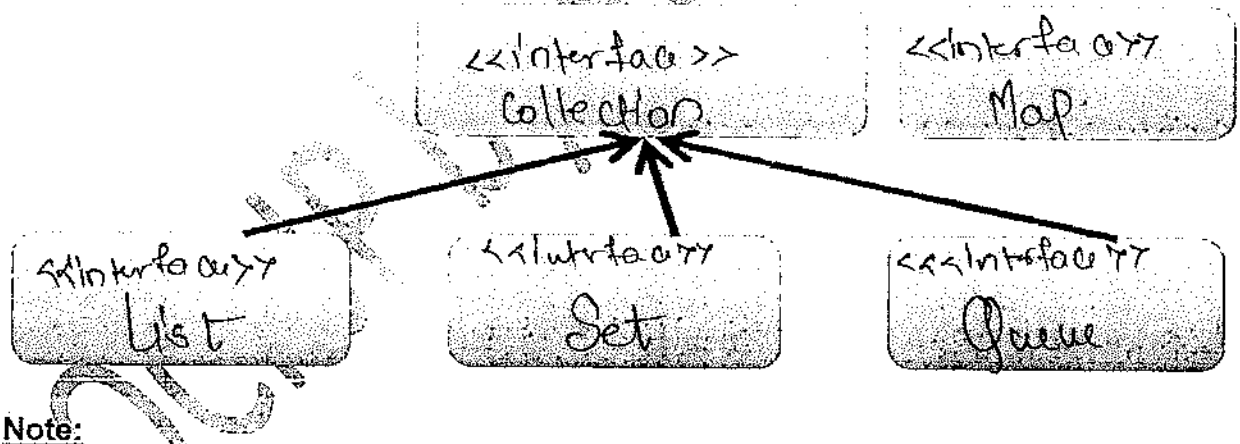
Class Hierarchy:



The Collection Interface:

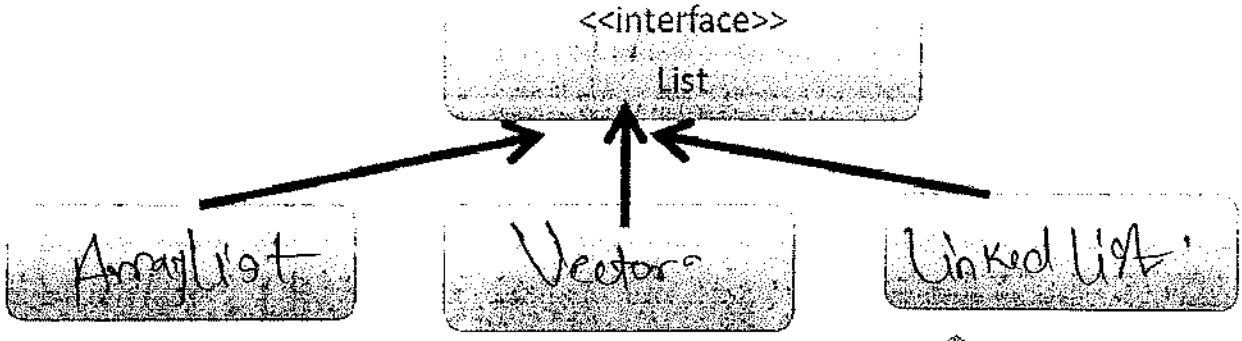
- The root of the hierarchy of the collections interfaces is the Collection interface, also referred to as the superinterface of the collections.
- There is another kind of collections called maps, which are represented by the superinterface Map, which is not derived from the Collection interface.

Collection Hierarchy:



Note:
Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including add(), remove(), contains(), size(), and iterator().

List Hierarchy:



List	Ordered	Sorted
ArrayList	By Index	No
Vector	By Index	No
LinkedList	By Index	No

List Interface:

- A List cares about the index.
- All three List implementations are ordered by index position—a position that you determine either by setting an object at a specific index or by adding it without specifying position, in which case the object is added to the end.

ArrayList:

- It as a growable array.
- It gives you fast iteration and fast random access. it is an ordered collection (by index), but not sorted.
- Some of the advantages ArrayList has over arrays are
  - It can grow dynamically.
  - It provides more powerful insertion and search mechanisms than arrays.

Program to illustrate ArrayList:

```
import java.util.*;
public class ALE1a
{
public static void main(String[] args)
{
ArrayList al = new ArrayList();
al.add("Java");
al.add("C++");
al.add("Java");
al.add("Oracle");
al.add("C");
al.add("AJAX");
System.out.println("Contents of array list: " +  al );
Collections.sort(al);
System.out.println("Contents of array list: " +  al );
}
}
```

Output:

```
>javac ALE1a.java
Note: ALE1a.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

>java ALE1a
Contents of array list: [Java, C++, Java, Oracle, C, AJAX]
Contents of array list: [AJAX, C, C++, Java, Java, Oracle]
```

Note: we are getting note coz with ArrayList we have not specified the type of Object we are going to store.

Limitations of Nongeneric Collections:

- The ArrayList can contain any Object , even though we only want it to store Strings.
- We can add Integer and Boolean which is a problem with nongenerics because the compiler cannot stop us from putting a Integer and Boolean object in the ArrayList .
- We will use generics to specify the data type of the elements in the ArrayList .

```
import java.util.*;
public class ALE1aa
{
public static void main(String[] args)
{
ArrayList al =new ArrayList();
al.add("Java");
al.add("C++");
al.add("Java");
al.add(123);
al.add("C");
al.add(true);
System.out.println("Contents of array list: " +  al );
Collections.sort(al);
System.out.println("Contents of array list: " +  al );
}
}
```

Output:

```
>javac ALE1aa.java
Note: ALE1aa.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
>java ALE1aa
Contents of array list: [Java, C++, Java, 123, C, true]
Exception in thread "main"
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
    at java.lang.Integer.compareTo(Unknown Source)
    at java.util.ComparableTimSort.binarySort(Unknown Source)
    at java.util.ComparableTimSort.sort(Unknown Source)
    at java.util.ComparableTimSort.sort(Unknown Source)
    at java.util.Arrays.sort(Unknown Source)
    at java.util.Collections.sort(Unknown Source)
```

Using Generic Collections:

- As of J2SE 5.0, the class declaration of ArrayList is
  - public class ArrayList < E >
- The < E > represents a generic element.

Program using Generics:

```
import java.util.*;
public class ALE1b
{
public static void main(String[] args)
{
ArrayList<String> al = new ArrayList<String>();
al.add("Java");
al.add("C++");
al.add("Java");
al.add("Oracle");
al.add("C");
al.add("AJAX");
System.out.println("Contents of array list: " +  al );
Collections.sort(al);
System.out.println("Contents of array list: " +  al );
}
}
```

Output:

```
>javac ALE1b.java
>java ALE1b
Contents of array list:
[Java, C++, Java, Oracle, C, AJAX]
Contents of arraylist:[AJAX,
 C, C++, Java, Java, Oracle]
```

Advantage of Using Generics:

- The ArrayList can only contain String objects, and the compiler enforces this rule.
- The add method of ArrayList only accepts String references. Notice how generics allow issues like this one to be discovered at compile time.
- The other benefit of generics is that you do not need to cast the data when accessing elements in the collection which improves both the readability and reliability of the code.

ArrayList Methods	Description
add(Object o)	Appends the specified element to the end of the list.
add(int index, Object o)	Inserts the specified element at the specified position in the list.
remove(int index)	Removes the element at the specified position in the list.
remove(Object o)	Removes the object from the list.
contains(Object element)	Returns true if this list contains the specified element.
get(int index)	Returns the element at the specified position in the list.
size()	Returns the number of elements in the list.

Program:

```
import java.util.*;
public class ALE1c {
 public static void main(String[] args) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al :  " + al.size());
al.add("C");
al.add("C++");
al.add("Java");
al.add("Oracle");
al.add("PHP");
al.add(1,"AJAX");
System.out.println("Size of array list after
addition : " + al.size());
System.out.println("Contents of array list : " +  al );

al.remove("PHP");
al.remove(2);

System.out.println("Size of array list after
deletions : " + al.size());

for(String s : al)
System.out.print(s + " ");
System.out.println();

System.out.println("al contains Java : "
+ al.contains("Java"));
System.out.println("al position 2 : " + al.get(2));

Iterator i = al.iterator();
while(i.hasNext())
        System.out.print(i.next() + " " );
}
}
```

Output:

```
>javac ALE1c.java
>java E1c
```

Q1) Can we determine the output of ArrayList?
Ans: Yes, We Can. Bcoz the data is arranged index-wise.

Q2) Can we add same value more than once?
Ans: Yes. We Can.

Q3) Can we store null value to ArrayList?
Ans: Yes. with ArrayList We Can Store null (more than one) value also

Q4) Can we sort ArrayList with null values?
Ans: JVM raises NullPointerException.

Q5) Can we write: List<String> al = new ArrayList<String>();
Ans: Yes we Can. This is Called Polymorphic reference

Q6) Can we write: List<Object> al = new ArrayList<String>();
Ans: CF.

Vector:

- Vector is basically the same as an ArrayList, but **Vector** methods are **synchronized** for thread safety.
- Use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need.

LinkedList:

- A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.
- This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue.

Program:

```
import java.util.*;
public class LLE1a {
 public static void main(String[] args)  {
LinkedList<String> al = new LinkedList<String>();
al.add("C");
al.add("C++");
al.addFirst("Java");
al.addLast("Oracle");
al.add("PHP");
al.add(1,"AJAX");
al.remove("PHP");
al.remove(2);
al.removeFirst();
al.removeLast();
System.out.println("Contents of Linked List : " +  al );
}}
```

Output:

```
>javac LLE1a.java
>java LLE1a
Java AJAX
C++
```

Sorting and Searching in Lists:

- **Collections.binarySearch(l ,k):**
  Searches the specified list for the specified object using the binary search algorithm and returns an int indicating the index in the list.
- **Collections.sort(l):**
  Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
- **Collections.reverse(l):**
  Reverses the order of the elements in the specified list.

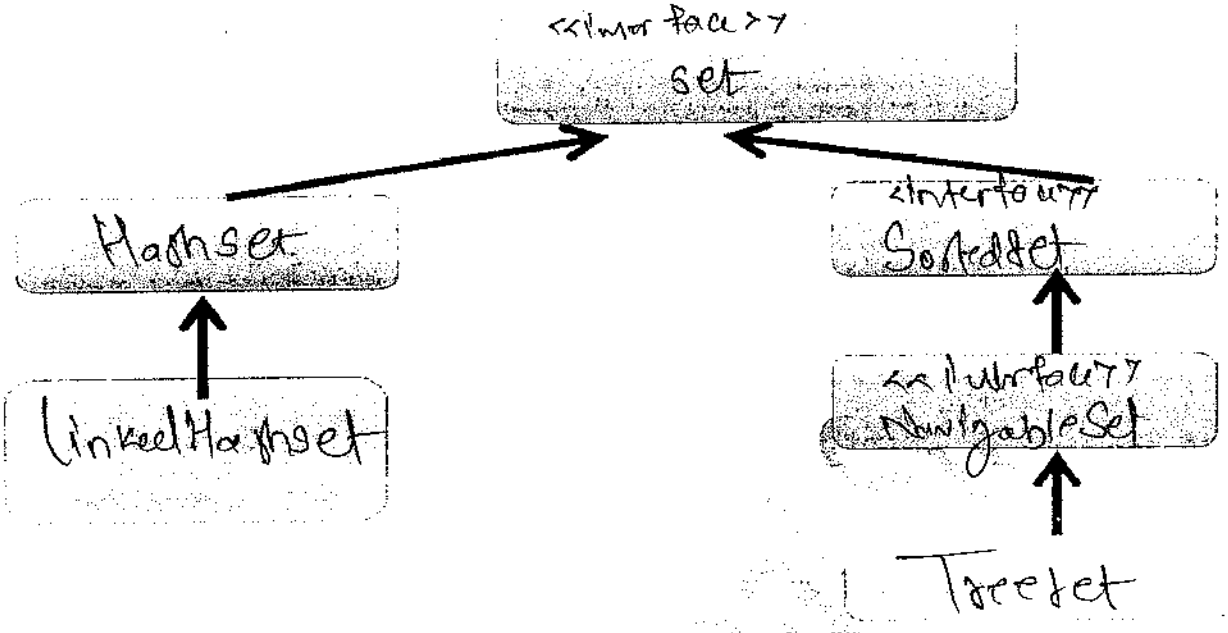
Program:

```
import java.util.*;
public class ALE1bc
{
public static void main(String[] args)
{
ArrayList<String> al = new ArrayList<String>();
al.add("Java");
al.add("C++");
al.add("Oracle");
al.add("C");
al.add("AJAX");

Collections.sort(al);

for(Object s : al)
        System.out.print(s + " " );
System.out.println();
al.add("Android");

Collections.reverse(al);
for(Object s : al)
        System.out.print(s + " " );
System.out.println();

System.out.println("Java is present at " +
Collections.binarySearch(al, "Java"));
}
}
```

Output:

```
>javac ALE1bc.java

>java ALE1bc

AJAX
 C
 C++
Java
Oracle.

Android
 Oracle
 Java
  C++
   C
 AJAX.

Java is present at
  2
```

Set Hierarchy:



Set	Ordered	Sorted
HashSet	No	No
LinkedHashSet	By Insertion Order	No
TreeSet	Sorted	By Natural Order or custom Comparison rules.

Set Interface:

- A Set cares about uniqueness—it doesn't allow duplicates.
- The **equals()** method determines whether two objects are identical (in which case only one can be in the set).

HashSet:

- A HashSet is an **unsorted, unordered Set**.
- It uses the **hashcode** of the object being **inserted**, so the more efficient your hashCode() implementation the better access performance you'll get.
- Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

Program:

```
import java.util.*;
public class HSC1a {
public static void main(String [] args) {
HashSet<String>  carData = new  HashSet<String>();
carData.add("Santro");
carData.add("Esteem");
carData.add("Accent");
carData.add("Xylo");
System.out.println("Size :" + carData.size() );
System.out.print("Car data: " + carData);
System.out.println();
System.out.println("Car data empty: " +  carData.isEmpty());
carData.remove("Accent");
carData.remove(2);
System.out.print("Car data: " + carData);
System.out.println();
carData.clear();
System.out.print("Car data: " + carData);
System.out.println();
}
}
```

Output:

4
[Accent, Esteem, Santro, Xylo]
false
[Esteem, Santro, Xylo]
[]

Q1) Can we determine the output of HashSet program?
Ans: No. Hashset is unordered and unsorted.

Q2) What if we add the same element more than once?
Ans: No. Duplicates. equals() checks for the duplicates.

Q3) Can we add more than one null value?
Ans: Only one null can be added.

Q4) On what basis the unique data is added to HashSet?
Ans: The equals() Method determines whether two objects are identical.

Q5) Which methods must be overridden to add objects to HashSet?
Ans: ~~Hash~~ hashcode() and equals() method.

LinkedHashSet :

- A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.
- Use this class instead of HashSet when you care about the iteration order.
- When you iterate through a **HashSet the order is unpredictable**, while a LinkedHashSet lets you iterate through the elements in the **order in** which they **were inserted**.

Note:

- When using HashSet or LinkedHashSet, the objects you add to them must override hashCode().
- If they don't override hashCode(), the default Object.hashCode() method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.

Program:

```
import java.util.*;
public class LHSC1a {
public static void main(String [] args) {
LinkedHashSet<String>  carData = new
LinkedHashSet<String>();
carData.add("Santro");
carData.add("Esteem");
carData.add("Accent");
carData.add("Xylo");
carData.add("Esteem");
System.out.print("Car data: " + carData);
System.out.println();
}}
```

Output:

```
>javac LHSC1a.java
>java LHSC1a
Car data: [Santro, Esteem, Accent, Xylo]
```

Program:

```
import java.util.*;
public class LHSC1b {
public static void main(String [] args) {
LinkedHashSet<Integer>  carData = new
LinkedHashSet<Integer>();
carData.add(123);
carData.add(57);
carData.add(null);
carData.add(-23);
carData.add(null);
carData.add(4573);
carData.add(56);
carData.add(-23);
System.out.print("Car data: " + carData);
System.out.println();
}}
```

Output:

```
>java

Car data: [123, 57, null, -23, 4573, 56].
```

TreeSet:

- The TreeSet guarantees that the elements will be in ascending order, according to natural order.
- Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator.
- As of Java 6, TreeSet implements NavigableSet.

Program:

```
import java.util.*;
public class TSC1a {
public static void main(String [] args) {
TreeSet<String> carData = new TreeSet<String>();
carData.add("Santro");
carData.add("Esteem");
carData.add("Accent");
carData.add("Xylo");
carData.add("Esteem");
System.out.print("Car data: " + carData);
System.out.println();
}}
```

Output:

Cardata [Accent, Esteem, Santro, Xylo]

Program:

```
import java.util.*;
public class TSC1c {
public static void main(String [] args) {
TreeSet<Integer> carData = new TreeSet<Integer>();
carData.add(123);
carData.add(57);
carData.add(-23);
carData.add(4573);
carData.add(56);
System.out.print("Car data: " + carData);
System.out.println();
}}
```

Output:

Cardata [-23, 56, 57, 123, 4573]

Program:

```
import java.util.*;
public class TSC1C1 {
public static void main(String [] args) {
TreeSet<Integer> carData = new TreeSet<Integer>();
carData.add(123);
carData.add(57);
carData.add(null);
carData.add(4573);
carData.add(null);
System.out.print("Car data: " + carData);
System.out.println();
}}
```

Output:

Cardata:
NullPointerException

Navigating TreeSets:

- There is another set of collections that can be searched and sorted. These collections are TreeSets and TreeMaps.
- In Java 6, two new interfaces, java.util.NavigableSet and java.util.NavigableMap, have been introduced.
- Now we will learn, how the TreeSet() and TreeMap() implement these interfaces.

Methods of NavigableSet Interface	
Iterator<E> iterator()	Returns an iterator over the elements in this set, in ascending order.
Iterator<E> descendingIterator()	Returns an iterator over the elements in this set in descending order.
NavigableSet<E> descendingSet()	Returns the elements contained in this set in the reverse order (descending order).
E floor(E element)	Returns the greatest element in this set less than or equal to the given element, or null if the element is not found.
E ceiling(E element)	Returns the least element in this set greater than or equal to the given element, or null if the specified element is not found.
E higher(E element)	Returns the least element in this set strictly greater than the given element, or null if the element is not found.
E lower(E element)	Returns the greatest element in this set strictly less than the given element, or null if the element is not found.
E pollFirst()	Retrieves and removes the first(lowest) element, or returns null if there is no element in the set.
E pollLast()	Retrieves and removes the last(highest) element or returns null if this set is empty.
SortedSet<E> headset(E thisElement)	Returns a view of the portion of this set whose elements are strictly less than thisElement.
SortedSet<E> tailset(E thisElement)	Returns a view of the portion of this set whose elements are greater than or equal to thisElement.
SortedSet<E> subset(E startElement, E endElement)	Returns a view of the portion of this set whose elements range from startElement inclusive, to endElement, exclusive.

Program:

```
import java.util.*;
 public class TS20a {
 public static void main(String[] args) {
 TreeSet<Integer> s = new TreeSet<Integer>();
 TreeSet<Integer> subs = new TreeSet<Integer>();
 s.add(10);
 s.add(30);
 s.add(20);
 s.add(50);
 s.add(40);
 subs = (TreeSet<Integer>)s.subSet(20, false, 40, true);
System.out.println(s + " " + subs);

 s.add(32);
System.out.println(s + " " + subs);

 s.add(53);
System.out.println(s + " " + subs);

 subs.add(38);
System.out.println(s + " " + subs);

 subs.add(42);
System.out.println(s + " " + subs);
}
}
```

Output:

[10 20 30 40 50] [30 40]

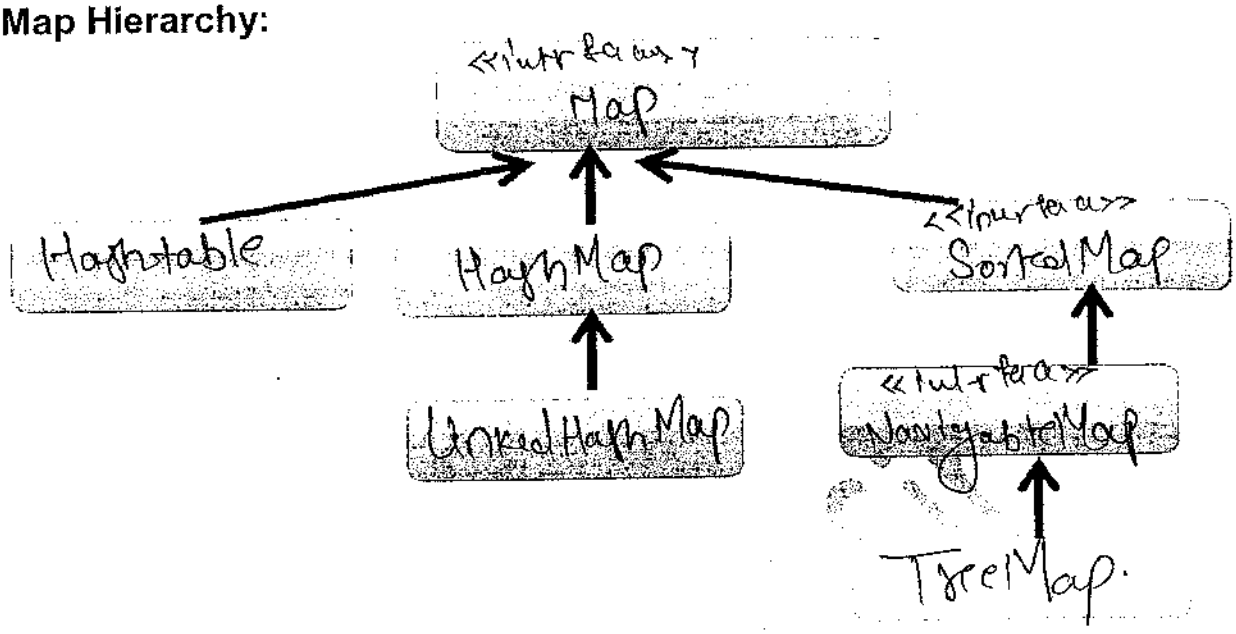
[10, 20, 30, 32, 40, 50] [30, 32, 40]

[10, 20, 30, 32, 40, 50, 53] [30, 32, 40]

[10, 20, 30, 32, 38, 40, 50, 53]
[30, 32, 38, 40]

java.lang.IllegalArgument Exception:
key out of range

Map Hierarchy:



Map	Ordered	Sorted
HashMap	No	No
Hashtable	NO	NO
LinkedHashMap	By insertion order	NO
TreeMap	Sorted	By Natural order or custom Comparison order.

Map Interface:

- A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects.
- The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys.
- Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

HashMap:

- The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go;
- Where the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your hashCode() implementation, the better access performance you'll get.
- HashMap allows one null key and multiple null values in a collection.

HashMap Methods	Description
put(k Key, v Value)	Associates the specified value with the specified key in this map.
containsKey(Object key)	Returns true if this map contains a mapping for the specified key
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
isEmpty()	Returns true if this map contains no key-value mappings.
size()	Returns the number of key-value mappings in this map.

Program:

```
import java.util.*;
class MC1a {
public static void main(String args[ ]) {
HashMap<Integer, String> hm = new HashMap<Integer, String>();
hm.put(11, "Alice");
hm.put(15, "Tom");
hm.put(14, "Bob");
System.out.println("Employee data" + hm);
System.out.println("Data contains key 15  " + hm.containsKey(15) );
System.out.println("Data contains value Alice  " + hm.containsValue("Alice") );
hm.clear();
System.out.println("Data is empty" + hm.isEmpty() );
}}
```

Output:

Employee data{11=Alice, 14=Bob, 15=Tom}
Data contains key 15 true
Data contains value Alice true
Data is empty true.

Note:— In map we don't add if we put.

Q1) Can we determine the output of HashMap?
Ans: No we cannot
HashMap is unsorted unordered.

Q2) Can we add same value more than once?
Ans: Yes We can
K,V Pair same, does not add.
K same and V not, then latest V.
K not and V same then adds new K,V pair.

Q3) Can we add null values to HashMap?
Ans: Hash map allows one null key and multiple null values in a Collection

Hashtable :
Hashtable is the synchronized counterpart to HashMap.

LinkedHashMap :

- The LinkedHashMap collection maintains insertion order (or, optionally, access order).
- Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

Program:

```
import java.util.*;
class LMC1a {
public static void main(String args[ ]) {
LinkedHashMap<Integer, String> hm = new LinkedHashMap<Integer,
String>();
hm.put(11, "Alice");
hm.put(15, "Tom");
hm.put(14, "Bob");
System.out.println("Employee data" + hm);
hm.put(17, "Bob");
hm.put(null, "Alex");
hm.put(15, "Tim");
hm.put(null, null);
hm.put(11, null);
hm.put(null, null);
System.out.println("Employee data" +hm);
}
}
```

Output:

TreeMap:

- TreeMap is a sorted Map, this means "sorted by the natural order of the elements."
- Like TreeSet, TreeMap lets you define a custom sort order (via a Comparable or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered.
- As of Java 6, TreeMap implements NavigableMap.

Program:

```
import java.util.*;
class TMC1b {
public static void main(String args[ ]) {
TreeMap<Integer, String> hm = new TreeMap<Integer, String>();
hm.put(11, "Alice");
hm.put(15, "Tom");
hm.put(14, "Bob");
System.out.println("Employee data" + hm);
hm.put(17, "Bob");
hm.put(15, "Tim");
hm.put(11, null);
System.out.println("Employee data" + hm);
}}
```

Output:

Employee data
11 = Alice
14 = Bob
15 : Tom}.
employee data:
{11 = null, 14 = Bob,
15 = Tim, 17 = Bob}

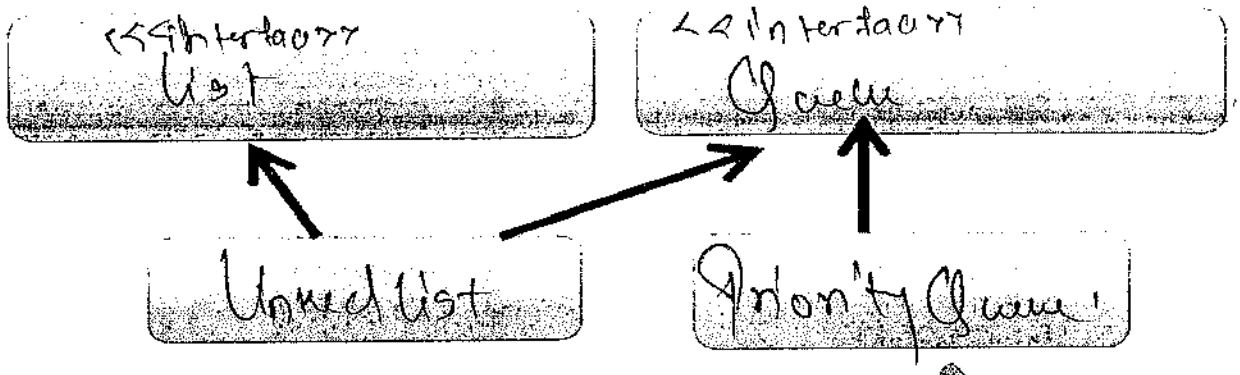
Program:

```
import java.util.*;
class TMC1a {
public static void main(String args[ ]) {
TreeMap<Integer, String> hm = new TreeMap<Integer, String>();
hm.put(11, "Alice");
hm.put(15, "Tom");
hm.put(14, "Bob");
System.out.println("Employee data" + hm);
hm.put(17, "Bob");
hm.put(null, "Alex");
hm.put(15, "Tim");
hm.put(null, null);
hm.put(11, null);
hm.put(null, null);
System.out.println("Employee data" + hm);
}}
```

Output:

Employee data
{11 = Alice, 14 = Bob, 15 = Tom}.
{11 = null, 14 = Bob,
15 = Tim
Null pointer Exception
Because of null in key.

Queue Hierarchy:



Queue	Ordered	Sorted
LinkedList	By index	No
PriorityQueue	Sorted	By to-do order [Natural order]

Queue Interface :

- A Queue is designed to hold a list of "to-dos," or things to be processed in some way.
- Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

PriorityQueue:

- PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue.
- A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator.
- In either case, the elements' ordering represents their relative priority.

Important Methods of PriorityQueue:	
add(E e)	Inserts the specified element in the queue.
offer(E e)	Inserts the specified element in the queue.
clear()	Removes all of the elements from this queue.
size()	Returns the number of elements in this queue.
poll()	Retrieves and removes the head of this queue, or returns null (if empty)
peek()	Retrieves but does not remove the head of this queue, or returns null (if empty)

Program:

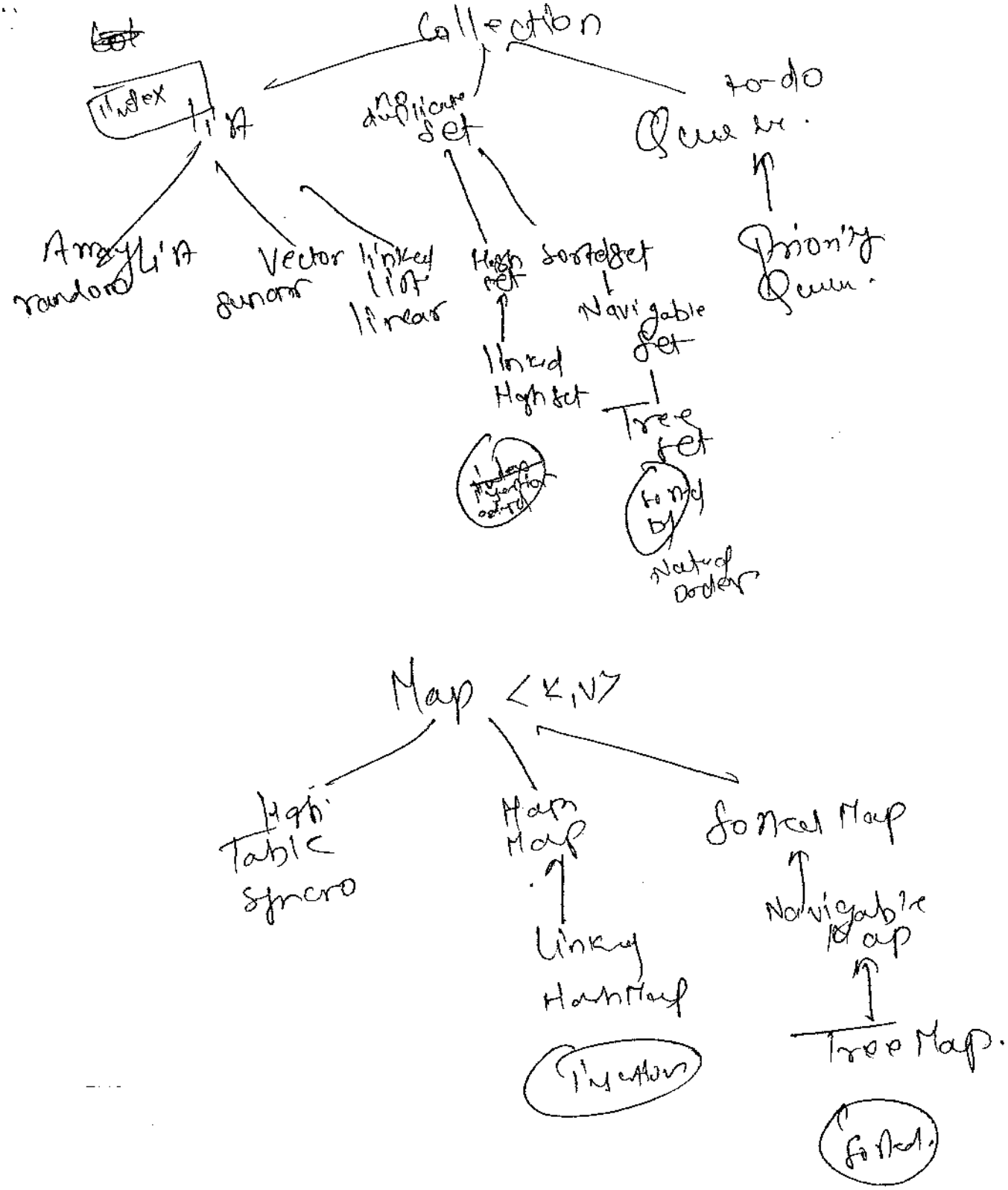
```
import java.util.*;
class PQ1a {
public static void main(String args[ ]) {
PriorityQueue<String> pq = new  PriorityQueue<String>();
pq.add("Alex");
pq.add("Tom");
pq.offer("Peter");
pq.add("Bob");
System.out.println("Queue size : "+ pq.size() );
System.out.println("Queue data : "+ pq);
System.out.println(pq.poll() );
System.out.println(pq.peek() );
System.out.println(pq.poll() );
System.out.println(pq.peek() );
System.out.println("Queue size : "+ pq.size() );
System.out.println("Queue data : "+ pq);
pq.clear();
System.out.println("Queue size : "+ pq.size() );
System.out.println("Queue data : "+ pq);
}
}
```

Output:

Q1) Can we determine the output of PriorityQueue program?
Ans:

Q2) Can we add same value more than once?
Ans:

Q3) Can we add null values to PriorityQueue?
Ans:


Collection
List
duplicate set
Queue
todo
ArrayList
random
Vector
synch
LinkedList
linear
HashSet
SortedSet
NavigableSet
TreeSet
Priority Queue
Map <K,V>
HashTable
synch
HashMap
LinkedHashMap
SortedMap
NavigableMap
TreeMap.

Test Paper:

Q1)
Which of these are core interfaces in the collections framework?
Select the three correct answers.

Options:
(a) Set<E>
(b) Bag<E>
(c) LinkedList<E>
(d) Collection<E>
(e) Map<K,V>

Solution: a d & e.

Q2)
Which of these implementations are provided by the java.util package?
Select the two correct answers.

Options:
(a) HashList<E>
(b) HashMap<K,V>
(c) ArraySet<E>
(d) ArrayMap<K,V>
(e) TreeMap<K,V>

Solution: b & e

Q3)
Which of the following statements is true? (Choose three)

Options:
A. A Set is a collection that does not allow duplicates.
B. A Map can store duplicate values.
C. A List is a collection that is ordered by index.
D. A List is a collection that cannot have duplicates.
E. The JDK provides a direct implementation of the Collection interface.

Solution: A, B & C

Q4) Which of the following classes implement the List interface? (Select two)

Options:

A. Vector

B. HashList

C. ArrayList

D. StackList

Solution: ABC

Q5) Which of the following is the most significant difference between ArrayList and Vector.

Options:

A. ArrayList is sychronized.

B. Vector is synchronized.

C. Only one of them is implemented in java.util.* package.

D. None of these.

Solution: B

Q6) A programmer has an algorithm that requires a java.util.List that provides an efficient implementation of add(0, object), but does NOT need to support quick random access. What supports these requirements?

Options:

A. java.util.Queue

B. java.util.ArrayList

C. java.util.LinearList

D. java.util.LinkedList [has linear access]

Solution: D

Q7) What is the result of the following statements?

```
6. List list = new ArrayList();
7. list.add("one");
8. list.add("two");
9. list.add(7);
10. for(String s : list) {
11. System.out.print(s);
12. }
```

// for(objects the ans is B.

Options:

A. onetwo

B. onetwo7

C. onetwo followed by an exception

D. Compiler error on line 9

E. Compiler error on line 10

Solution: E

Q8) What is the result of the following statements?

```
6. List < String > list = new ArrayList < String > ();
7. list.add("one");
8. list.add("two");
9. list.add(7);
10. for(String s : list) {
11. System.out.print(s);
12. }
```

Options:

A. onetwo
B. onetwo7
C. onetwo followed by an exception
D. Compiler error on line 9
E. Compiler error on line 10

Solution:

Q9) What is the result of the following statements?

```
3. ArrayList < Integer > values = new ArrayList < Integer > ();
4. values.add(4);
5. values.add(5);
6. values.set(1, 6);
7. values.remove(0);
8. for(Integer v : values) {
9. System.out.print(v);    10. }
```

Options:

A. 4
B. 5
C. 6
D. 46
E. 45

Solution:

Q10) Given:

```
11. public void genNumbers() {
12. ArrayList numbers = new ArrayList();
13. for (int i=0; i<10; i++) {
14. int value = i * ((int) Math.random());
15. Integer intObj = new Integer(value);
16. numbers.add(intObj);
17. }
18. System.out.println(numbers);
19. }
```

Which line of code marks the earliest point that an object referenced by intObj becomes a candidate for garbage collection?

Options:

A. Line 16
B. Line 17
C. Line 18
D. Line 19
E. The object is NOT a candidate for garbage collection.

Solution:

Q11)

```
1. class Pizza {
2. java.util.ArrayList toppings;
3. public final void addTopping(String topping) {
4. toppings.add(topping);
5. }
6. }
7. public class PepperoniPizza extends Pizza {
8. public void addTopping(String topping) {
9. System.out.println("Cannot add Toppings");
10. }
11. public static void main(String[ ] args) {
12. Pizza pizza = new PepperoniPizza();
13. pizza.addTopping("Mushrooms");
14. }
15. }
```

What is the result?

Options:

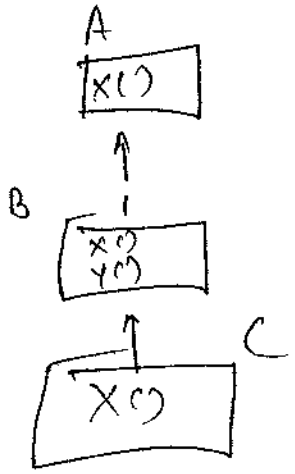
A. Compilation fails.
B. Cannot add Toppings
C. The code runs with no output.
D. A NullPointerException is thrown in Line 4.

Solution: A

Q12) Given:

```
10. interface A { void x(); }
11. class B implements A { public void x() {} public void y() {} }
12. class C extends B { public void x() {} }
And:
20. java.util.List<A> list = new java.util.ArrayList<A>();
21. list.add(new B());
22. list.add(new C());
23. for (A a : list) {
24. a.x();
25. a.y();
26. }
```

What is the result?



Options:

A. The code runs with no output.
B. An exception is thrown at runtime.
C. Compilation fails because of an error in line 20.
D. Compilation fails because of an error in line 21.
E. Compilation fails because of an error in line 23.
F. Compilation fails because of an error in line 25.

Solution: F

Q13) Given:

```
11. public static Collection get() {
12. Collection sorted = new LinkedList();
13. sorted.add("B"); sorted.add("C"); sorted.add("A");
14. return sorted;
15. }
16. public static void main(String[] args) {
17. for (Object obj: get()) {
18. System.out.print(obj + ", ");
19. }
20. }
```

What is the result?

Options:
A. A, B, C,
B. B, C, A,
C. Compilation fails.
D. B C A
E. B, C, A

Solution:

Q14) Given:

```
11. public static Iterator reverse(List list) {
12. Collections.reverse(list);
13. return list.iterator();
14. }
15. public static void main(String[ ] args) {
16. List list = new ArrayList();
17. list.add("1"); list.add("2"); list.add("3");
18. for (Object obj: reverse(list))
19. System.out.print(obj + ", ");
20. }
```

What is the result?

Options:
A. 3, 2, 1,
B. 1, 2, 3,
C. Compilation fails.
D. The code runs with no output.
E. An exception is thrown at runtime.

Solution:

Q15) Given:

```
5. import java.util.*;
6. public class SortOf {
7. public static void main(String[ ] args) {
8. ArrayList<Integer> a = new ArrayList<Integer>();
9. a.add(1); a.add(5); a.add(3);
11. Collections.sort(a);
12. a.add(2);
13. Collections.reverse(a);
14. System.out.println(a);
15. }
16. }
```

What is the result?

Options:

A. [1, 2, 3, 5]
B. [2, 1, 3, 5]
C. [2, 5, 3, 1]
D. [5, 3, 2, 1]
E. [1, 3, 5, 2]
F. Compilation fails.
G. An exception is thrown at runtime.

Solution:

Q16) Given:

```
1. import java.util.*;
2.
3. public class LetterASort{
4. public static void main(String[ ] args) {
5. ArrayList<String> strings = new ArrayList<String>();
6. strings.add("aAaA");
7. strings.add("AaA");
8. strings.add("aAa");
9. strings.add("AAaa");
10. Collections.sort(strings);
11. for (String s : strings) { System.out.print(s + " "); }
12. }
13. }
```

What is the result?

Options:

A. Compilation fails.
B. aAaA aAa AAaa AaA
C. AAaa AaA aAa aAaA
D. AaA AAaa aAaA aAa
E. aAa AaA aAaA AAaa
F. An exception is thrown at runtime.

Solution:

Q17) Which identifiers, when inserted in appropriate places in the program, will result in the output 911?

```
Collection<Integer> myItems = new ArrayList<Integer>();
myItems.add(9); myItems.add(1); myItems.add(1);
Iterator<Integer> iterator = myItems.iterator();
while (iterator.hasNext()) {
System.out.print(iterator.next());
}
```

Select the five correct answers.

Options:

(a) hasNext
(b) myItems
(c) next
(d) Integer
(e) int
(f) Collection
(g) iterator

Solution: a, b, c, d, g.

Q18) Which of the following statement differentiate the HashSet from the LinkedHashSet

Options:

A. HashSet permits the elements to be stored in the insertion order.
B. LinkedHashSet permits the elements to be stored in the insertion order.
C. Only one of the above are implemented in java.util package.
D. None of these.

Solution: D

Q19) Given:

```
3. import java.util.*;
4. public class Mapit {
5. public static void main(String[ ] args) {
6. Set<Integer> set = new HashSet<Integer>();
7. Integer i1 = 45;
8. Integer i2 = 46;
9. set.add(i1);
10. set.add(i1);
11. set.add(i2); System.out.print(set.size() + " ");
12. set.remove(i1); System.out.print(set.size() + " ");
13. i2 = 47;
14. set.remove(i2); System.out.print(set.size() + " ");
15. }  16. }
```

What is the result?

Options:

A. 2 1 0
B. 2 1 1
C. 3 2 1
D. 3 2 2
E. Compilation fails.

Solution: B

Q20) Given:

```
1. import java.util.*;
2. public class Example {
3. public static void main(String[ ] args) {
4. // insert code here
5. set.add(new Integer(2));
6. set.add(new Integer(1));
7. System.out.println(set);
8. }
9. }
```

Which code, inserted at line 4, guarantees that this program will output [1, 2]?

Options:

A. Set set = new TreeSet();

B. Set set = new HashSet(); // HashSet is unset and unsorted.

C. Set set = new SortedSet();

D. List set = new SortedList();

E. Set set = new LinkedHashSet();

Solution:

Q21) Given:

```
1. import java.util.*;
2. public class WrappedString {
3. private String s;
4. public WrappedString(String s) { this.s = s; }
5. public static void main(String[ ] args) {
6. HashSet<Object> hs = new HashSet<Object>();
7. WrappedString ws1 = new WrappedString("aardvark");
8. WrappedString ws2 = new WrappedString("aardvark");
9. String s1 = new String("aardvark");
10. String s2 = new String("aardvark");
11. hs.add(ws1); hs.add(ws2); hs.add(s1); hs.add(s2);
12. System.out.println(hs.size()); } }
```

What is the result?

Options:

A. 0
B. 1
C. 2
D. 3
E. 4
F. Compilation fails.
G. An exception is thrown at runtime.

Boz WrappedString is Programmer defined class and it did not override equals and hashCode methods.

Solution: D

Q22) Given:

```
12. import java.util.*;
13. public class Explorer1 {
14. public static void main(String[ ] args) {
15. TreeSet<Integer> s = new TreeSet<Integer>();
16. TreeSet<Integer> subs = new TreeSet<Integer>();
17. for(int i = 606; i < 613; i++)
18. if(i%2 == 0) s.add(i);
19. subs = (TreeSet)s.subSet(608, true, 611, true);
20. s.add(609);
21. System.out.println(s + " " + subs);
22. } 23. }
```

What is the result?

Options:

A. Compilation fails.
B. An exception is thrown at runtime.
C. [608, 609, 610, 612] [608, 610]
D. [608, 609, 610, 612] [608, 609, 610]
E. [606, 608, 609, 610, 612] [608, 610]
F. [606, 608, 609, 610, 612] [608, 609, 610]

Solution: F

6
[606 608 610 612]
Subs
[608 610]

Q23) Given:

```
12. import java.util.*;
13. public class Explorer2 {
14. public static void main(String[ ] args) {
15. TreeSet<Integer> s = new TreeSet<Integer>();
16. TreeSet<Integer> subs = new TreeSet<Integer>();
17. for(int i = 606; i < 613; i++)
18. if(i%2 == 0) s.add(i);
19. subs = (TreeSet)s.subSet(608, true, 611, true);
20. s.add(629);
21. System.out.println(s + " " + subs);
22. } 23. }
```

What is the result?

Options:

A. Compilation fails.
B. An exception is thrown at runtime.
C. [608, 610, 612, 629] [608, 610]
D. [608, 610, 612, 629] [608, 610, 629]
E. [606, 608, 610, 612, 629] [608, 610]
F. [606, 608, 610, 612, 629] [608, 610, 629]

Solution: E

[606 608 610 612]
Subs
[608 610]

Q24) Given:

```
12. import java.util.*;
13. public class Explorer3 {
14. public static void main(String[ ] args) {
15. TreeSet<Integer> s = new TreeSet<Integer>();
16. TreeSet<Integer> subs = new TreeSet<Integer>();
17. for(int i = 606; i < 613; i++)
18. if(i%2 == 0) s.add(i);
19. subs = (TreeSet)s.subSet(608, true, 611, true);
20. subs.add(629);
21. System.out.println(s + " " + subs);
22. } 23. }
```

What is the result?

Options:

A. Compilation fails.
B. An exception is thrown at runtime.
C. [608, 610, 612, 629] [608, 610]
D. [608, 610, 612, 629] [608, 610, 629]
E. [606, 608, 610, 612, 629] [608, 610]
F. [606, 608, 610, 612, 629] [608, 610, 629]

Subs range is 608 to 610
illegal argument Exception

Solution:

Q25) Given the proper import statement(s), and:

```
13. TreeSet<String> s = new TreeSet<String>();
14. TreeSet<String> subs = new TreeSet<String>();
15. s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");
16.
17. subs = (TreeSet)s.subSet("b", true, "d", true);
18. s.add("g");
19. s.pollFirst();
20. s.pollFirst();
21. s.add("c2");
22. System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose two)

Options:

A. The size of s is 4
B. The size of s is 5
C. The size of s is 7
D. The size of subs is 1
E. The size of subs is 2
F. The size of subs is 3

Solution: B & 3

Q26) Given the following declaration:

Map < String, Double > map = new HashMap < String, Double > ();

which of the following statements are valid? Choose all that apply.

Options:

A. map.add(" pi " , 3.14159);

B. map.add(" e " , 2.71828D);

C. map.add(" log(1) " , new Double(0.0));

D. map.add('x', new Double(123.4));

E. None of the above.

Solution: B

Q27) What is the result of the following statements?

```
3. Map < Integer, Integer > map = new HashMap < Integer, Integer > ();
4. for(int i = 1; i < = 10; i++) {
5. map.put(i, i * i);
6. }
7. System.out.println(map.get(4));
```

Options:

A. Compiler error on line 3

B. Compiler error on line 5

C. Compiler error on line 7

D. 16

E. 25

Solution: D

Q28) Given:

```
11. public class Key {
12. private long id1;
13. private long id2;
14.
15. // class Key methods
16. }
```

A programmer is developing a class Key, that will be used as a key in a standard java.util.HashMap. Which two methods should be overridden to assure that Key works correctly as a key? (Choose two.)

Options:

A. public int hashCode()

B. public boolean equals(Key k)

C. public int compareTo(Object o)

D. public boolean equals(Object o)

E. public boolean compareTo(Key k)

Solution: A & D.

Q29) Given that the elements of a PriorityQueue are ordered according to natural ordering, and:

```
2. import java.util.*;
3. public class GetInLine {
4. public static void main(String[ ] args) {
5. PriorityQueue<String> pq = new PriorityQueue<String>();
6. pq.add("banana");
7. pq.add("pear");
8. pq.add("apple");
9. System.out.println(pq.poll() + " " + pq.peek());
10. }
11. }
```

What is the result?

Options:

A. apple pear

B. banana pear

C. apple apple

D. apple banana

E. banana banana

Solution:

Q30) Given:

```
1. import java.util.*;
2. class Priorities {
3. public static void main(String[ ] args) {
4. PriorityQueue toDo = new PriorityQueue();
5. toDo.add("dishes");
6. toDo.add("laundry");
7. toDo.add("bills");
8. toDo.offer("bills");
9. System.out.print(toDo.size() + " " + toDo.poll());
10. System.out.print(" " + toDo.peek() + " " + toDo.poll());
11. System.out.println(" " + toDo.poll() + " " + toDo.poll());
12. } }
```

What is the result?

Options:

A. 3 bills dishes laundry null null

B. 3 bills bills dishes laundry null

C. 3 dishes dishes laundry bills null

D. 4 bills bills dishes laundry null

E. 4 bills bills bills dishes laundry

F. 4 dishes laundry laundry bills bills

G. Compilation fails.

Solution: