

Java: Collection and Generics

Objectives

- Evaluate different types of collection classes in Java to know when to use which class

Table of Content

Collection framework	Set
Generics	HashSet
Collection	hashCode()
List interface	LinkedHashSet
ArrayList	SortedSet and TreeSet
Type erasure	Map
Mixing generics and legacy code	HashMap and Hashtable
Vector	LinkedHashMap
Stack and Queue	Properties
LinkedList	

Collection framework

- A collection in java is an object that can hold multiple objects (like an array) .
- It grows dynamically.
- Examples of collection classes are **Stack**, **LinkedList**, **Dictionary**.
- A collection framework is a common architecture for representing and manipulating all the collections. This architecture has a set of interfaces on the top and implementing classes down the hierarchy. Each interface has specific purpose.
- Collection framework uses generics.

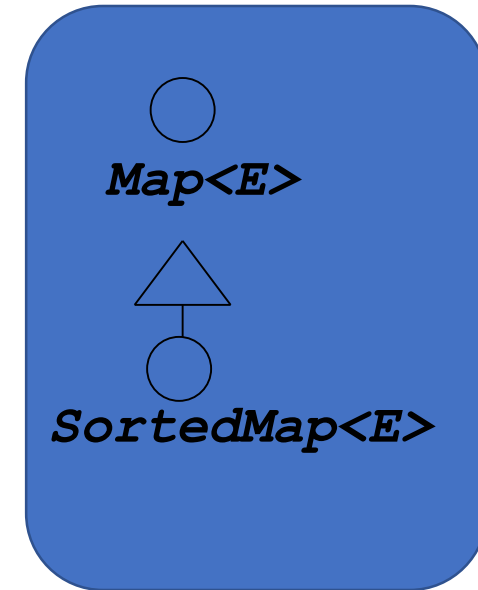
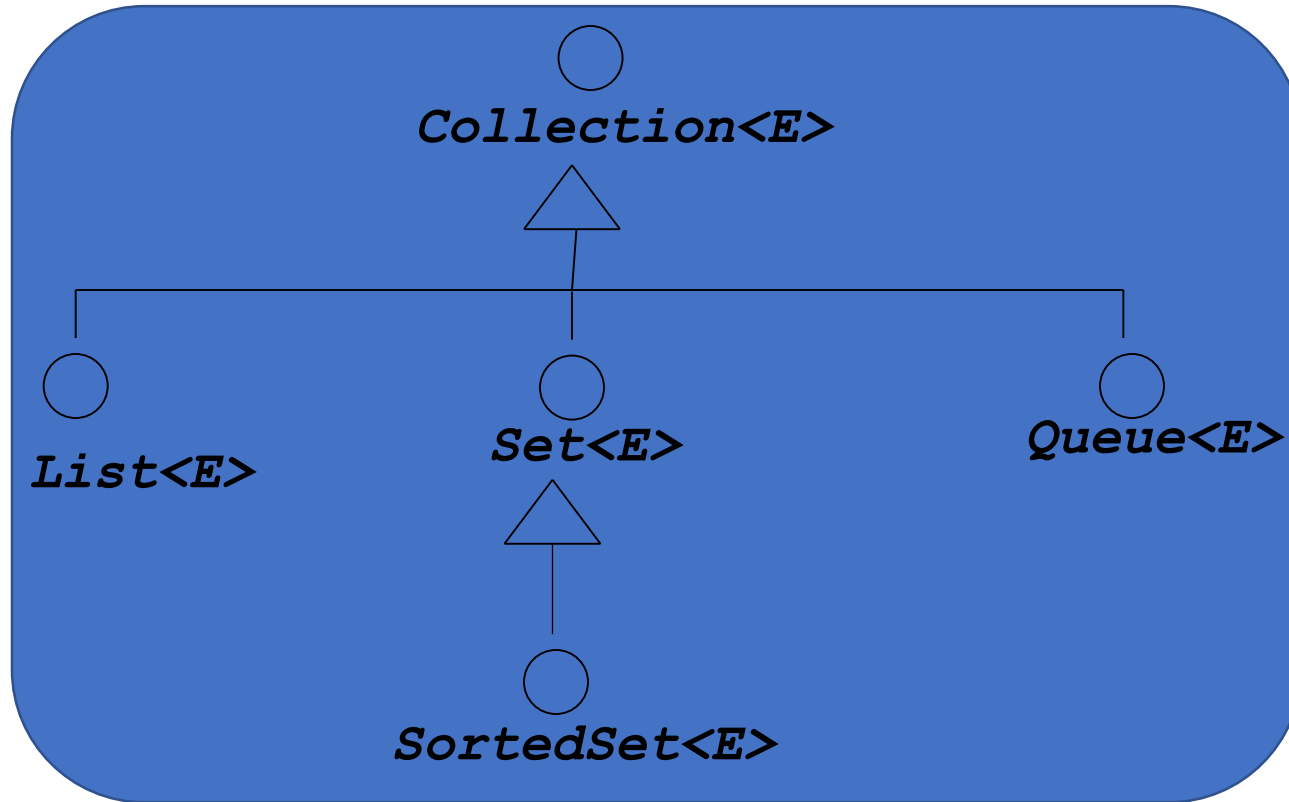
Generics

- Prior to 1.5, collection methods used **Object** in their collection classes.
- From 1.5 onwards, Java has added newer syntax to allow programmers to create type-safe collections
- The type that will be used to create the collection object is specified at the time of instantiation.
- The collection methods therefore use generic symbols (like 'E').
- Note that E can represent only classes not primitives.

Tell me why

- Why do I need a collection framework when I can create my own classes?
 - Is it not better to use the well tested code than to reinvent the wheel?
 - Advantages
 - Reduces design, coding and testing efforts and therefore saves time.
 - Variety of classes to choose from in terms of performance and memory.
 - The collection interfaces at the top layer reduces the learning effort
 - It fosters reuse when new collection classes are added.

Collection interfaces



`java.util` package

Collection Classes

Interface	Implementation Classes
1. List<E>	ArrayList<E> Vector<E> Stack<E> LinkedList<E>
2. Set<E> SortedSet<E>	HashSet<E> LinkedHashSet<E> TreeSet<E>
3. Map<E> SortedMap<E>	Hashtable<E> HashMap<E> LinkedHashMap<E> TreeMap<E>
4. Queue	LinkedList<E> PriorityQueue<E>

Collection

- It is the root interface in the collection hierarchy.
- The items in the collection is refereed to as elements.
- **Collection** interface extends another interface called **Iterable**.
- Any class that implements **Iterable** can use the enhanced for-loop to iterate through elements.
- **Iterable** has single methods that returns **Iterator**

Iterator<T> iterator()

- **Iterator** is an interface with 3 methods :
 - **boolean hasNext()**
 - **E next()**
 - **void remove()**

Methods of Collection interfaces



Go through the methods of Collection class for 5 minutes.

<? extends E> → all objects that are instances of the subclasses of E and E itself is allowed.

<?> → all objects allowed (short form of **<? extends Object>**)

If you call `remove("xyz")` on collection, will all xyz string be removed from the collection?

Which is the method that is totally opposite to `removeAll()`?

List interface

- The subclasses of **List** are ordered collection of objects.
- The elements are ordered based on user's input.
- Methods of **List** interface provide indexed access to the objects.
- The index begins from 0.
- From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.
- **List** is also called sequence



If you call `remove(1)` from the list, will the object representing 1 be removed or object in index position 1 is removed?

ArrayList

- This class implements all the methods defined **List** interface even the optional methods)

- Constructors:

ArrayList()

Creates an array with default capacity of 10

ArrayList(int initialCapacity)

No new methods added here!

Creates an array with initial capacity as specified by “initialCapacity”

- What will happen on adding 11th element in **ArrayList** which has capacity as 10?*

Elements gets added without any issues. Internally, arrays capacity increases. Remember, the collection classes grow dynamically!

- Also note that the **capacity** is different from **size**. While the size returns the number of elements in the list, capacity is the maximum allocated space for the list.

Example: Using ArrayList

```
import java.util.*;
class ArrayListTest{
public static void main(String[] s){
ArrayList<Integer> a= new ArrayList<Integer>();
a.add(1);
a.add(2);
a.add(3);
for(Integer o:a)
System.out.println(o);
}}
```

Must be the generic type that is defined.

Can also be `int`, *why?*

1	0
2	1
3	2
	9

Exercise

- *Coordinator adds the names of the participants who wish to participate in extempore. He also removes the names if the participants decides otherwise or if they don't meet the required criteria.*
- *A list is sorted and split into a list of 5 participants and a seminar room number is allocated. This information is maintained as another list. Finally the application must display the list as :*

Group 1: seminar room

participants name

Group 2: seminar room

participants name

and so on

Hint: Use the ArrayList and Arrays class.

(45 mins)

Traditional way

```
import java.util.ArrayList;
public class ArrayListEx {

    public static void main(String[] s) {
        ArrayList a= new ArrayList();
        a.add(1);
        a.add(1.78);
        double sum=0;
        for(Object o:a) {
            Number d=null;
            // cast the object based on type and use it
            if(o instanceof Number) {
                d=(Number)o;
                sum =sum+d.doubleValue();
            }
        }
        System.out.print(sum);}}
```

Warning generated by compiler

Activity

- Convert the code in the pervious slide to generic way.
- What are the advantages of this code compared to the previous code?

Type erasure

- Generic type information is present only at compile time.
- After compiler ensures all the checks are met, it *erases* all the generic information.
- As a result the code looks like the traditional code (i.e. code without generics that used raw type). Which means that the **`ArrayList<Integer>`** becomes **`ArrayList`** at runtime.
- This is done to ensure binary compatibility with older Java libraries and applications that were created before generics.

Tell me how

- What if the legacy code tries to insert a integer into String collections, if there is no generic information at runtime? How do you eliminate such dangerous insertions?
 - To make sure your collection is type-safe even at runtime, **checkXXX** methods in **Collections** class should be used.
 - We will look into this at the end of this session when we do **Collections** (not **Collection** interface!)class.

Mixing generics and legacy code

- It is recommended not to mix generis and legacy code.
- There are times when we need to do this. This is so in cases where we need to pass collection arguments to legacy code and vice versa. In such case we need to convert generics to raw type and vice versa.
- Following are the legal ways of mixing generics to raw types:

1. `ArrayList<Integer> a= new ArrayList();`

2. `ArrayList a= new ArrayList<Integer>();`

3. `List<Integer> a= new ArrayList();`

4. `List a= new ArrayList<Integer>();`

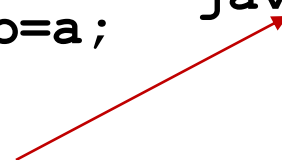
Beware!

Consequences of mixing raw types with generics. Examples -

Case 1:

```
import java.util.*;
class Test{
public static void main(String[] s){
ArrayList a=new ArrayList();
a.add(1);
a.add(1.1);
ArrayList<Integer> b=a;
for(Integer i:b)
System.out.println(i);}}
a
```

java.lang.ClassCastException:
java.lang.Double at runtime



Case 2:

```
List l= new ArrayList<Integer>();
l.add("AbC");
```

Vector

- The **Vector** class is exactly same as **ArrayList** class except that the Vector class methods are **thread-safe**.

Exercise

- *Create a Vector object that can hold any type of object : Student or Teacher or HOD. Write a java code that creates these objects and inserts them into the list. Make sure that toString() is overridden in all the classes. Print out the list that displays the string representation of the object. It should also print the object type such as Student, Teacher or HOD.*

(30 mins)

Stack and Queue

- Objects are inserted in Stack - LIFO, Queue – FIFO manner.
- Inherits from the **Vector** class.

Go through the methods for Stack and Queue classes



What is the difference between

- 1. push, peek and pop.*
- 2. poll, remove.*
- 3. peek, element.*

LinkedList

- **LinkedList** implements **List** and **Queue**.
- All the **List** classes we have seen so far used arrays internally. **LinkedList** class uses doubly-linked list.
- The methods in the **LinkedList** allow it to be used as a stack, queue, or double-ended queue.
- Note that this class is not thread-safe.



Go through the methods for 5 minutes.

Exercise

- *Implement a railway ticket counter scenario where there are two queues- one general and one for senior citizen.*
- *Tickets are issued such that for every one person in senior citizen queue, two persons in general queue are processed.*
- *Write a program that takes input for 6 people who come at various points and print the list of people in the order of their processing sequence.*

Hint: implement Queue interface

(30 mins)

Exercise

- *Use LinkedList to store list of Scores objects (name, score) that will be entered by the user. Make sure they are arranged in the descending order while they are inserted into the linked list. Display the linked list in the order of the rank of the student.*

(30 mins)

Test your understanding

When should **LinkedList** be used instead of array **ArrayList**? Both of them offer dynamic growth.

Set

- **Set** cannot contain duplicate elements.
- Two objects **o1** and **o2** are duplicates if **o1.equals(o2)** returns **true**. That is, a **Set** cannot contain both **o1** and **o2** such that **o1.equals(o2)** is **true**.
- It can contain at most one **null** element.
- **Set** does not add any new methods apart from what it gets from **Collection** interface.
- Classes implementing **Set** must
 - Not add duplicate element
 - Return **false** if an attempt is made to add duplicate element.

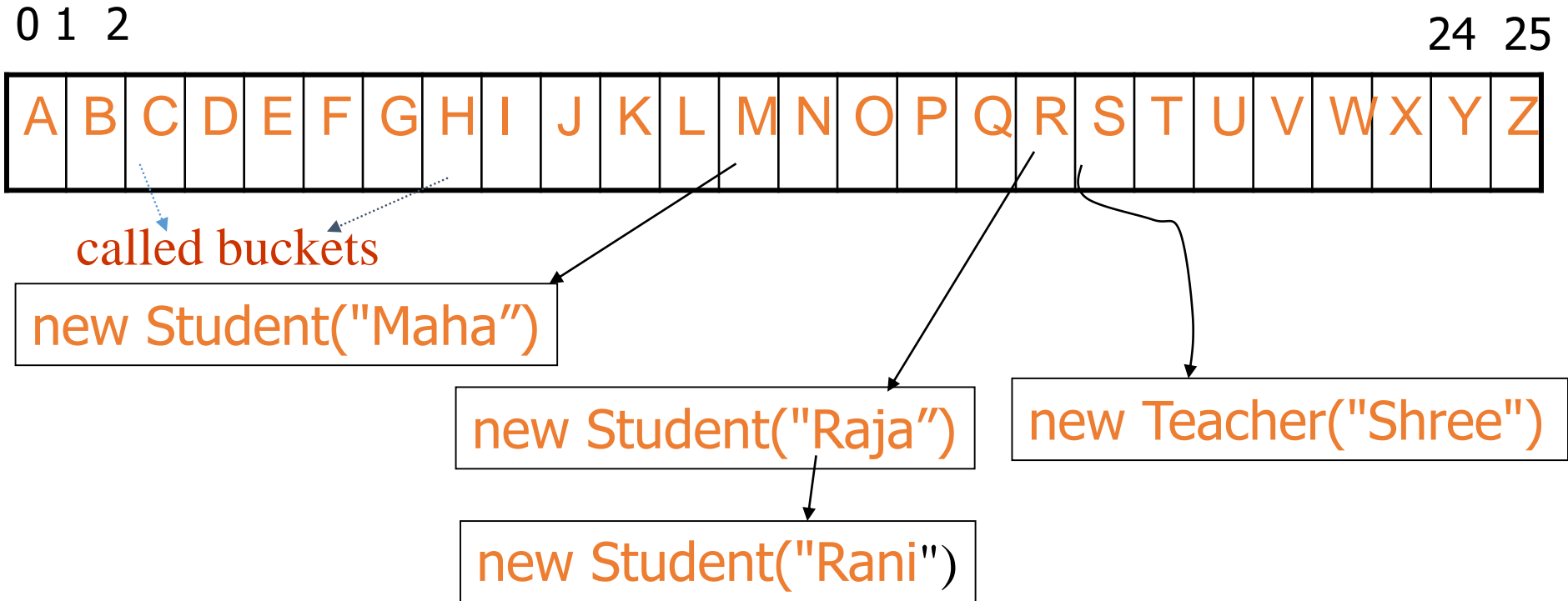
HashSet

- **HashSet** is an unordered and unsorted set that does not allow duplicates.
- Unordered and unsorted means that there is no guarantees as to the iteration order of the set; it is may not be in the order that user enters and it may not be in the sorted order.
- Also there is no guarantee that the order will remain constant over time when new entries are added.
- **HashSet** stores its elements in a hash table.
- Therefore this class offers constant time performance for the basic operations like **add**, **remove**, **contains** etc.
- This is also not a thread-safe class

hashCode ()

- This class relies heavily on **hashCode ()** method of the object that is added in **HashSet**.
- Positioning elements using **hashCode ()** helps in faster retrieval. So, more efficient the **hashCode ()**, better the performance.
- **Object** class has **hashCode ()** method.
- The implementation of **hashCode ()** provided by the **Object** class leads to a linear search because each object has a unique bucket.
- The performance would be better only if we can classify a set of objects and put them together inside a bucket and then do a linear search inside the bucket. Hence we need to override hashCode() method.

hashCode () Example



```
package general;  
public abstract class Person{  
    ...  
    public int hashCode() {  
        return name.charAt(0);    }  
    public boolean equals(Object obj) {  
        return name.equals((String)obj); }  
}
```

Example: HashSet

Adding Teachers and Students in the **HashSet**

```
import java.util.*;
public class TestHashSet {
    public static void main(String str[]) throws Exception
    {
        teacher.Teacher s1= new teacher.Teacher("Guru");
        student.Student s2= new student.Student("Shree");
        teacher.Teacher s3= new teacher.Teacher("Kumar");
        student.Student s4= new student.Student("Sheela");
        HashSet<Person> set= new HashSet<Person>();
        set.add(s1);
        set.add(s2);
        set.add(s3);
        set.add(s4);
        for(Person p:set) {
            System.out.println(p.getName());
        }
    }
}
```

Result:

Sheela
Shree
Kumar
Guru

→ Into one bucket

LinkedHashSet

- Subclass of **HashSet**, maintains the insertion-order and does not allow duplicates.
- If a duplicate element is entered, insertion order of the first one is maintained since 2nd one is not inserted at all.
- It implements a hashtable using doubly-linked list.
- Like **HashSet**, this class also has constant-time performance for the basic operations (add, contains and remove) if the hash function is implemented properly. But compared to **HashSet**, this class is slow except in case of iterating over the collection in which case **LinkedHashSet** is faster.
- Same constructor and methods like **HashSet**
- Like **HashSet** this is also not a thread-safe class

Exercise

Given an array of employee ids who were listed as outstanding for last 2 years.

Say : {1,2,6,3,4,5,6,7,9,4}

Write code that picks the employees who are listed as outstanding for 2 consecutive years.

(20 mins)

Test your understanding

- Taking the previous slide example further, how will you get employees who have only a single outstanding?
- Hint: Go back to **Collection** interface and find out if there are methods that allow you to get set difference!

SortedSet and *TreeSet*

- **SortedSet** is an interface. This interface guarantees that while traversing the order will be either
 - A. in natural order (using **compareTo()** of **Comparable** interface)
or
 - B. by using a **Comparator** provided at creation time.
- This is not a thread-safe class.



*Go through the methods of *SortedSet* and *TreeSet*.*

TreeSet Example

```
import java.util.*;

public class SortedElements {

    public static void main(String[] a){
        TreeSet<String> set = new TreeSet<String>();
        set.add("banana");
        set.add("citrus");
        set.add("apple");
        System.out.println(set);
        NavigableSet<String> n= set.descendingSet();
        System.out.println(n);
        NavigableSet<String> n1= set.headSet("banana", true);
        n1.add("apricot");
        System.out.println(n);
        n1.add("pineapple");
        System.out.println(n);}}}
```

Code displays:

```
[apple, banana, citrus]
[citrus, banana, apple]
[citrus, banana, apricot, apple]
Exception in thread "main" java.lang.IllegalArgumentException: key out of range
at java.util.TreeMap$NavigableSubMap.put(Unknown Source)
at java.util.TreeSet.add(Unknown Source)
at Tester.main(Tester.java:16)
```

Note that the code throws an exception on an attempt to insert a key outside its range

Exercise

- *10 volunteers are needed for Showcase of a new product. Write a program that will accept employees ids who will volunteer for this. Make sure that the ids are not duplicates. Display the ids in a sorted order.*

(20 mins)

Map

- A **Map** maps keys to values. So there are 2 columns in a Map : key and value.
- A map cannot contain duplicate keys; each key can map to at most one value. Therefore keys in the **Map** are like **Set**.
- Note that **Map** is not **Iterable**, therefore enhanced for loop cannot be used for **Map**.
- **Map.Entry** is an interface that is used to represent key-value pair.

HashMap and Hashtable

- There are 2 similar classes **HashMap** and **Hashtable** that implements **Map**. The only difference between **HashMap** and **Hashtable** is that **Hashtable** is thread-safe.
- Both of the classes arrange the pair of objects with respect to **hashCode ()** of the key and the keys map to a value.

Exercise

- *Write a class representing thesaurus that has many synonyms for a single word mapped. User can use this to search meaning of the words they want.*

(20 mins)

LinkedHashMap

- Subclass of **HashMap**, **LinkedHashMap** is like **LinkedHashSet** implements a hashtable using doubly-linked list.
- With **LinkedHashSet** 2 types of order can be maintained
 - Insertion order : Order in which entries get inserted into the collection is maintained. The insertion order is not affected if a key is re-inserted into the map.
 - Access order: A linked hash map can also be used to maintain iteration order in terms of entries how they were accessed. The accessed element is taken out and put at the end of the collection. This kind of map is well-suited to building LRU caches. A special constructor is provided in this class to specify this.

Example: LinkedHashMap

- This example demonstrates **LinkedHashMap** with access order.

```
import java.util.*;

public class LRU {

    public static void main(String[] args) {
        // LinkedHashMap for Caching using access-order

        Map<Integer, String> cacheMap = new
        LinkedHashMap<Integer, String>(10, 0.75f, true);
        cacheMap.put(4, "D");
        cacheMap.put(1, "A");
        cacheMap.put(5, "E");
        cacheMap.put(3, "C");
        cacheMap.put(2, "B");
        System.out.println("LinkedHashMap - not access yet: " +
        cacheMap);
    }
}
```

```
cacheMap.get(1);
cacheMap.get(5);
cacheMap.get(3);
System.out.println("LinkedHashMap after accessing some
elements : " + cacheMap);

cacheMap.get(2);
cacheMap.put(6, "F");
System.out.println("LinkedHashMap after accessing and
adding new entry : " + cacheMap);
}}
```

Result

```
LinkedHashMap - not access yet: {4=D, 1=A, 5=E, 3=C,
2=B}
LinkedHashMap after accessing some elements : {4=D, 2=B,
1=A, 5=E, 3=C}
LinkedHashMap after accessing and adding new entry :
{4=D, 1=A, 5=E, 3=C, 2=B, 6=F}
```

Test your understanding: `TreeMap`

Have you seen any other **`TreeXXX`** class?

This class is very similar to that class.

Let us make a few guesses about this class.

- **`TreeMap`** implements **`NavigableMap`** interface which extends **`SortedMap`** interface.
- **Methods**
 - `Sorted____<E> subMap(E fromKey, E toKey)`
 - `Sorted____<E> headMap(E toKey)`
 - `Sorted____<E> tail____(E fromKey)`

Can you describe what these methods do?

Iterating through a TreeMap example

```
public static void main(String[] s){
    TreeMap<String,Double> hm = new
    TreeMap<String,Double>();
    // Put elements to the map
    hm.put("Jerry", new Double(10000.00));
    hm.put("Tom", new Double(5000.22));
    hm.put("Mary", new Double(7000.00));
    hm.put("Susan", new Double(4000.00));
    // Get a set of the entries
    Set<Map.Entry<String,Double>> set = hm.entrySet();
    // Get an iterator
    Iterator<Map.Entry<String,Double>> i = set.iterator();
    // Display elements
    while(i.hasNext()) {
        Map.Entry<String,Double> me = i.next();
        System.out.print(me.getKey() + ": ");
        System.out.println(me.getValue()); }
}
```

```
Jerry:
10000.0
Mary: 7000.0
Susan: 4000.0
Tom: 5000.22
```

Exercise

- *Write a program to implement a telephone directory. Provide facilities to add, delete and search the telephone directory.*

(30 mins)

- *A shop has a list of product code , description and price. Some prices are listed in terms of kg and others are listed in terms of dozens. Customers buys the different products in different quantities. The application must display a bill with the product code , description , quantity and price per unit and total price.*

(45 mins)

Summary

- A collection in java is an object that can hold multiple objects which can grow dynamically.
- A collection framework has a set of interfaces on the top and implementing classes down the hierarchy. Collection framework uses generics.
- The subclasses of List are ordered collection of objects.
- The Vector class is exactly same as ArrayList class except that the Vector class methods are thread-safe.
- Set cannot contain duplicate elements.
- HashSet is an unordered and unsorted set that does not allow duplicates.
- A Map maps keys to values. The 2 similar classes HashMap and Hashtable implements Map. The only difference between HashMap and Hashtable is that Hashtable is thread-safe.