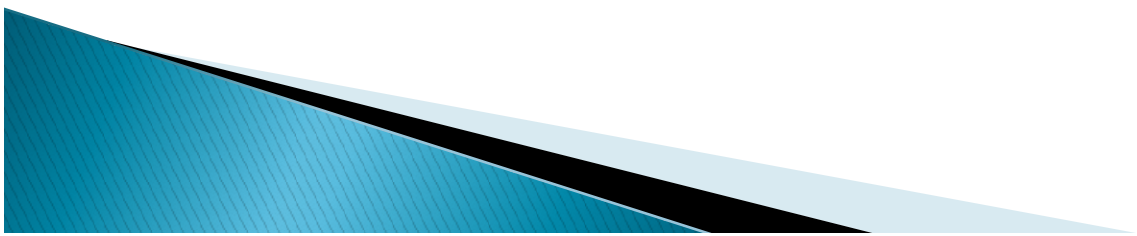
Collections in Java

By
Sunil Kumar(Master of Sc.)
Bangalore, India

1. Agenda

- What are Collections
- Role of Collections
- Collections and Type Safety
- Common Collection Methods
- Collection and Entry Equality
- Converting between Collections and Array
- Common Collection Interfaces and Classes
- Sorting Behavior
- Map Collections



What are Collections

- When we build our applications, something that we commonly have to do is manage groups of data.
  - Ex. Building an application to manage passengers on a flight. Group of Passenger data
  - Ex. In case of Social Media App. Group of Messages
- To store these groups of commonly typed data, the most basic solution is to use arrays.
  - Arrays works for simple cases
  - Arrays have limitations like Statically Sized, Requires Explicit position management
- Collections Provide more powerful Options

What are Collections Contd..

- Collections hold and organize values
  - Iterable
  - Can provide type safety
  - Tend to dynamically size
- A wide variety of collections are available
  - May be a simple list of values
  - Can provide optimization or sophistication
    - Ordering
    - Prevent Duplicates
    - Manage Data as name/value pair.

A simple collection of Objects

```
ArrayList list = new ArrayList();
list.add(“Foo”);
list.add(“Bar”);
System.out.println(“Elements : ”+list.size());
for(Object o:list)
    System.out.println(o.toString());
String s = (String) list.get(0);
SomeClassIMadeUp c = new SomeClassIMadeUp();
list.add(c);
```

- Here we can see that in the defined collection, we have added the String of our choice.
- Later we have tried adding the class object and it didn’t throw any error.
- We will see, how we can actually restrict a given collection to a specific type.

Collections and Type Safety

- By default collections hold Object types.
  - So any return values that we got back from the collection were of type object, which means we had to actually convert them to our desired type and the collection didn't restrict what type we could add to the collection.
- Collection can be type restricted.
  - Uses the java concept called generics
  - Type specified during collection creation
- Collection type restriction is pervasive
  - It means when we associate a collection with a particular type, the return values that you get back from that collection will be typed appropriately.
  - That will be typed based on what you have associated the collection with, and the collection will enforce those types when you add things to it. So the compiler will check to make sure that anything we are adding to collection is compatible with the type associated with the collection.

Collection Interface

- Java provides a number of different collection types, and each of those collections have their own features, but many of the features are common across collections.
- These common features mostly come from the collection interface, and most collections implement that interface.
- Map collection are notable exception. They don't implement the collection interface, and we'll talk more about them a little bit later in this module.

Collection Interface

- Collection interface extends the iterable interface, which is why we can use things like full reach to walk through collections, but of course a collection interface provides a number of methods that are specific to collections.

Common Collection Methods

Method	Description
size	Return number of elements
clear	Removes all elements
isEmpty	Return true if no elements
add	Add a single elements
addAll	Add all member of another collection

Example on Methods

```
ArrayList<String> list1 = new ArrayList();
list1.add("Foo");
list1.add("Bar");
LinkedList<String> list2 = new LinkedList<>();
list2.add("Baz");
list2.add("Boo");
list1.addAll(list2);
for(String s:list1){
   System.out.println(s);
}
System.out.println("Elements : "+list1.size());
System.out.println("List-1 is empty or not ? "+list1.isEmpty());
System.out.println("List-2 is empty or not ? "+list2.isEmpty());
list2.clear();
System.out.println("After clear List-2 is empty or not ? "+list2.isEmpty());
```

Common Equality-Based Methods

Method	Description
contains	Returns true if contains elements
containsAll	Return true if contains all member of another collection
remove	Remove elements
removeAll	Remove all elements contained in another collection
retainAll	Remove all elements not contained in another collection

- Each one of these require some kind of an equality test, and that equality test is based on the classes equals method. So it doesn't do a reference equals, it doesn't look and see if it's the exact same object, it uses the return value of the equals method.

Example

```
ArrayList<MyClass> list1 = new ArrayList<>();
MyClass v1 = new MyClass("v1", "abc");
MyClass v2 = new MyClass("v2", "efg");
MyClass v3 = new MyClass("v3", "abc");
list1.add(v1);
list1.add(v2);
list1.add(v3);
for(MyClass m:list1)
   System.out.println(m.getLebel());
list1.remove(v3);   //Walk through the list and removes the 1st member which equals to v3
for(MyClass m:list1)
   System.out.println(m.getLebel());
ArrayList<MyClass> list2 = new ArrayList<>();
MyClass v4 = new MyClass("v4", "abc");
list2.add(v4);
for(MyClass m:list1)
   System.out.println(m.getLebel());
```

LAMBDA Expression

- Java 8 introduces lambda expressions
  - Simplify passing code as arguments
- Collection methods that leverage lambdas
  - forEach
    - Perform code for each member
  - removeIf
    - Remove elements if test is true

Using For Each Method

```
public class lambda_example1 {
    public static void main(String args[]){
        ArrayList<MyClass> list = new ArrayList<>();
        MyClass v1 = new MyClass("v1", "abc");
        MyClass v2 = new MyClass("v2", "xyz");
        MyClass v3 = new MyClass("v3", "abc");

        list.add(v1);
        list.add(v2);
        list.add(v3);

        list.forEach(m -> System.out.println(m.getLebel()));
    }
```

Using For Each Method

- The format of the lambda expressions first provide a parameter name, and that name could be anything you want, I'm just using the name m here.
- Then we use the arrow symbol, which is hyphen, greater than. So we're basically saying we're passing m into this code.
- m will represent the current member in the collection, so m will represent each member in the collection once.
- Then the code we want to run is this System.out.println and then m.getLabel. So that means is that this method, System.out.println will be called once for each member, for that member will call getLabel, which then in turn will print out the value of the label, so it prints the first member's label, second member's label, and the third member's label.

Using removeIf Method

```
ArrayList<MyClass> list = new ArrayList<>();
        MyClass v1 = new MyClass("v1", "abc");
        MyClass v2 = new MyClass("v2", "xyz");
        MyClass v3 = new MyClass("v3", "abc");

        list.add(v1);
        list.add(v2);
        list.add(v3);

        list.removeIf(m -> m.getValue().equals("abc"));
        list.forEach(m -> System.out.println(m.getLebel()));
```

Converting Between Collection & Array

- Sometime APIs require an array
  - Often due to legacy or library code
- Collection interface can return an array
  - toArray() method
    - Return Object array
  - toArray(T[] array) method
    - Retrun array of type T
- Array content can be retrieved as collection
  - Use Array class asList method

Converting Between Collection & Array

- Now there are times in our code where we need to be able to convert between collections and arrays, and what it comes down to is just very often maybe dealing with a particular API that requires an array, even though you may be working in a collection, and often that's because you're dealing with legacy code or just some library code that you don't control. Now thanks to the methods on the collection interface, we can easily get an array from our collections. Now we use the toArray method. Now there's one version of toArray that takes no parameters. If you use that version, you'll get back an array of type Object containing the members from the collection.
- There's another version of toArray that accepts an array as a parameter, and that version of toArray will return back an array of the same type as the array passed in. It's also worth noting though is that if you have an array and you need it as a collection, the arrays class has a static method called asList that will give us a collection containing the members of a particular array.

Retrieving An Array

```
ArrayList<MyClass> list = new ArrayList<>();
      list.add( new MyClass("v1", "abc"));
      list.add( new MyClass("v2", "xyz"));
      list.add( new MyClass("v3", "abc"));
      Object[] objArray = list.toArray();
      MyClass[] a1 = list.toArray(new MyClass[0]);
      MyClass[] a2 = new MyClass[3];
      MyClass[] a3 = list.toArray(a2);

      if(a2 == a3) {
         System.out.println("a2 and a3 reference the same array");
      }
      else{
         System.out.println("a2 and a3 reference the different array");
      }
```

Retrieving a collection from an Array

```
MyClass[] myArray= {
        new MyClass("val1", "abc"),
        new MyClass("val2", "xyz"),
        new MyClass("val3", "abc")
    };

    Collection<MyClass> list =
Arrays.asList(myArray);
    list.forEach(c ->
System.out.println(c.getLebel()));
```

Collection Type

- Java provide a wide variety of collections
  - Each with Specific behaviour
- Collection Interfaces
  - Provide contract for collection behaviour
- Collection Classes
  - Provide collection implementation
  - Implement 1 or more collection interfaces

Common Collection Interfaces

Interface	Description
Collection	Basic collection operations
List	Collection that maintains a particular order
Queue	Collection with the concept of order and specific "head" element
Set	Collection that contains no duplicate values
SortedSet	A Set whose members are sorted

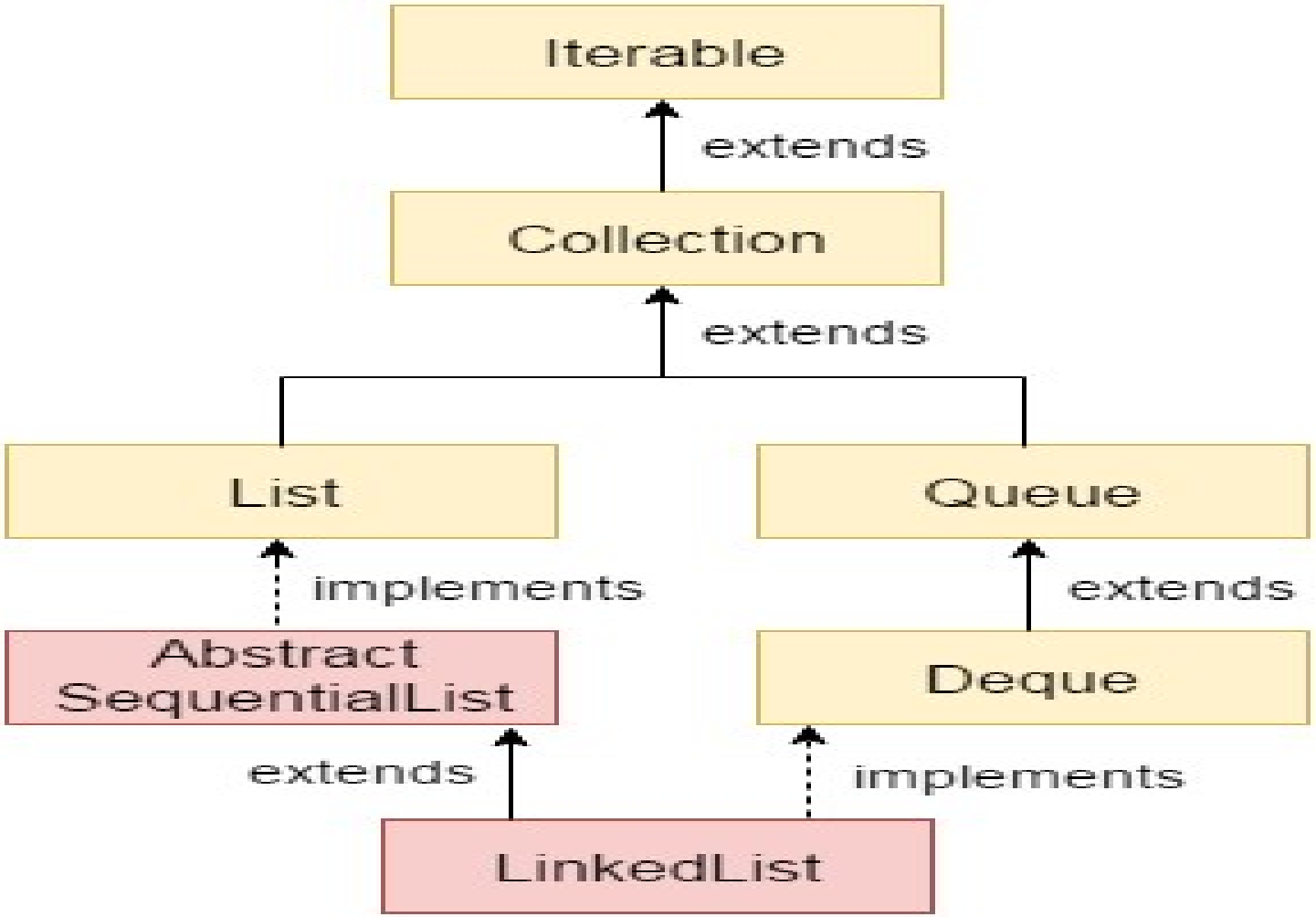
Collection Classes

Class	Description
ArrayList	A *List* backed by a resizable array Efficient random access but inefficient random inserts
LinkedList	A *List* and *Queue* backed by a doubly-linked list Efficient random insert but inefficient random access
HashSet	A *Set* implemented as a hash table Efficient general purpose usage at any size
TreeSet	A *SortedSet* implemented as a balanced binary tree Members accessible in order but less efficient to modify and search than a HashSet

LinkedList

- LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.
- The important points about Java LinkedList are:
  - Java LinkedList class can contain duplicate elements.
  - Java LinkedList class maintains insertion order.
  - Java LinkedList class is non synchronized.
  - In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
  - Java LinkedList class can be used as list, stack or queue.

LinkedList Contd..



LinkedList Contd..

```
public static void main(String args[]){

 LinkedList<String> al=new LinkedList<String>();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ravi");
 al.add("Ajay");

 Iterator<String> itr=al.iterator();
 while(itr.hasNext()){
  System.out.println(itr.next());
 }
```

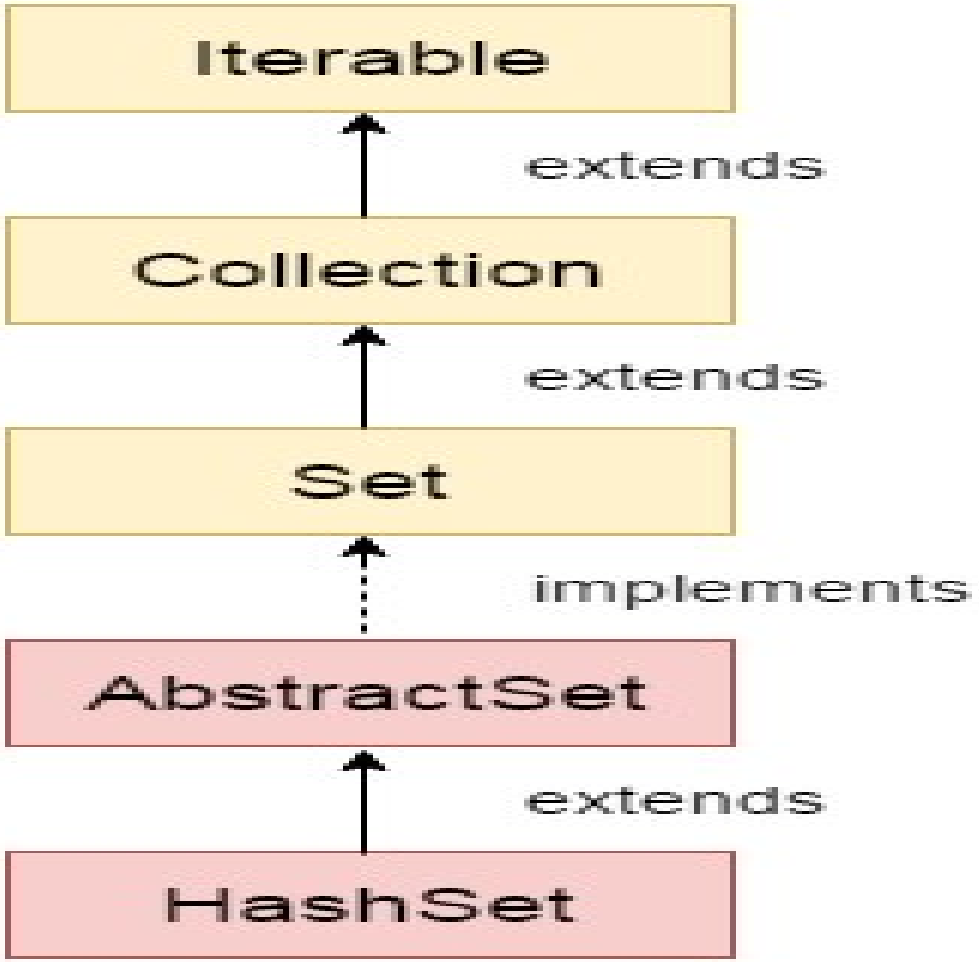
LinkedList vs ArrayList

ArrayList	LinkedList
ArrayList internally uses **dynamic array** to store the elements	LinkedList internally uses **doubly linked list** to store the elements
Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
  - HashSet stores the elements by using a mechanism called **hashing.**
  - HashSet contains unique elements only.

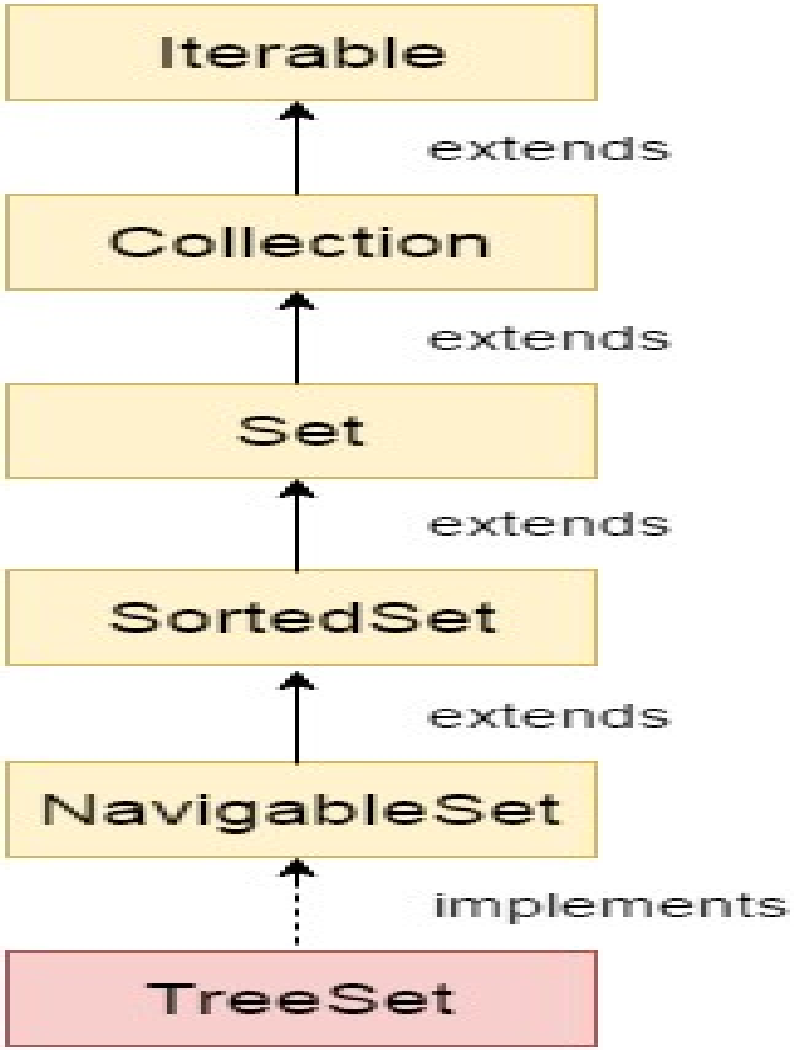
HashSet Contd..



TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.
- The important points about Java TreeSet class are:
  - Contains unique elements only like HashSet.
  - Access and retrieval times are quiet fast.
  - Maintains ascending order.

TreeSet Contd..



Sorting

- Some collection rely on sorting
  - Two ways to specify sort behavior
- Comparable Interface
  - Implemented by the type to be sorted
  - Type specifies own sort behavior
    - Should be consistent with equals
- Comparator Interface
  - Implemented by type to perform sort
  - Specifies sort behavior for another type

Using TreeSet with Comparable

```
TreeSet<MyClassWithComparale> tree = new TreeSet<>();
        tree.add(new MyClassWithComparale("2222", "ghi"));
        tree.add(new MyClassWithComparale("3333", "abc"));
        tree.add(new MyClassWithComparale("1111", "def"));
        tree.forEach(m -> System.out.println(m));
```

Output will be:

3333|abc

1111|def

2222|ghi

Using TreeSet with Comparator

```
public int compare(MyClassWithComparale x, MyClassWithComparale y){
        return x.getLabel().compareToIgnoreCase(y.getLabel());
   }
```

```
TreeSet<MyClassWithComparale> tree = new TreeSet<>(new MyComparator());
      tree.add(new MyClassWithComparale("2222", "ghi"));
      tree.add(new MyClassWithComparale("3333", "abc"));
      tree.add(new MyClassWithComparale("1111", "def"));

      tree.forEach(m -> System.out.println(m));
```

Map Collection

- Map store key/value pairs
  - Key used to identify/locate values
  - Keys are unique
  - Values can be duplicated
  - Values can be NULL

- Map is useful if you have to search, update or delete elements on the basis of key

Common Map Types

Interface	Description
Map	Basic Map Operations
SortedMap	Map whose keys are sorted

Class	Description
HashMap	Efficient general purpose Map Implementation
TreeMap	SortedMap implemented as a self-balancing tree. Supports Comparable and Comparator Sorting

Common Map Method

Method	Description
put	Add key and value
putIfAbsent	Add key and value if key not contained or value null
get	Return value for key, if key not found return null
getOrDefault	Return value for key, if key not found return the provided default value
values	Return a collection of the contained values
keySet	Return a set of the contained keys
forEach	Perform action for each entry
replaceAll	Perform action for each entry replacing the each key's value with the action's result

Map Method Example

```
Map<Integer, String> map = new HashMap<Integer, String>();
      map.put(100,"Ravi");
      map.put(200,"Vijay");
      map.put(300,"Ravi");
      map.put(400,"Ajay");
      System.out.println("Print using For Loop");
      for(Map.Entry s:map.entrySet()){
         System.out.println(s.getKey()+"  "+s.getValue());
      }
      System.out.println("Print using Lambda Expression");
      map.forEach((k, v) -> System.out.println(k+"  "+v));

      map.replaceAll((k,v) -> v.toUpperCase());
      System.out.println("Print after using replaceAll method");
      map.forEach((k, v) -> System.out.println(k+"  "+v));
   }
```

Common SortedMap Methods

Method	Description
firstKey	Return first key
lastKey	Return last key
headMap	Return a map for all keys that are less than the specified key
tailMap	Return a map for all keys that are greater than the specified key
subMap	Return a map for all keys that are greater than or equal to the starting key and less than the ending key

SortedMap Method Example

```
SortedMap<Integer, String> map = new TreeMap<>();
        map.put(100,"Ravi");
        map.put(200,"Vijay");
        map.put(300,"Ravi");
        map.put(400,"Ajay");
        map.put(500,"Nyasha");
        map.put(600,"Sunil");
        map.put(700,"Dane");

        System.out.println("Print using Lambda Expression");
        map.forEach((k, v) -> System.out.println(k+"  "+v));
        System.out.println("Print after applying headMap");
        SortedMap<Integer, String> hMap = map.headMap(300);
        hMap.forEach((k, v) -> System.out.println(k+"  "+v)); //headMap is exclusing of the key passed
        System.out.println("Print after applying tailMap");
        SortedMap<Integer, String> tMap = map.tailMap(400);
        tMap.forEach((k, v) -> System.out.println(k+"  "+v)); //tailMap is inclusing of the key passed
```

Summary

- Collections hold and organize values.
  - Collections are iterable
  - Tend to dynamically size
  - Can provide optimizations or certain sophistication.
- Collection will hold its entries as object, but collections can be type restricted
  - Uses the Java generics to specify type
  - Return values appropriately typed
  - Typing enforced on added values

Summary

- Can convert between collections and arrays
  - Collections interface provides the toArray method to get back the array as an array of object or as a typed array,
  - Array class provide toList method to convert back to collection type.
- Some collections provide sorting
  - Support Comparable interface
    - Type define own sort
  - Support Comparator interface
    - Specifies sort for another type
- Map Collection
  - Stores key/value pairs
  - Keys are unique
  - Some maps sort keys