ntroduction

ollection

There are really three overloaded uses of the word **"collection"**:

1. collection (lowercase c), is a data structure in which objects are stored and iterated over. In simple words, a collection is basically a collection of objects.
2. Collection (capital C), is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)
3.
4. Collections (capital C and ends with s) is the java.util.Collections class that holds a pile of static utility methods for use with collections.

Uses of collections:

There are few basic operations we will 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.

Collections Framework

The Java collections framework (JCF) is a set of classes and interfaces using which we can create collections.

Collections come in four basic flavors:

* **Lists** - Lists of things (classes that implement List)
* **Sets** - Unique things (classes that implement Set)
* **Maps** - Things with a unique ID (classes that implement Map)
* **Queues** - Things arranged by the order in which they are to be processed.

The following are the specific classes which we need to know for exam:

Lists	Sets	Maps	Queues	Utilities
ArrayList	HashSet	HashMap	PriorityQueue	Collections
Vector	LinkedHashSet	Hashtable		Arrays
LinkedList	TreeSet	LinkedHashMap		
		TreeMap		

List interface

1. A List cares about the index. The one thing that List has that non-Lists don't is a set of methods related to the index.
2. Those key methods include get(int index), indexof (object o), add(int index, object obj), and so on.
3. 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 positions.
4. The three List implementations are described in the following sections.

[This tables are very Imp in all concepts]

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

[index hai ki nahi?] [Sorted hai ki nahi?]

I) ArrayList

[obj of any class can be inserted in ArrayList]

1. An ArrayList as a growable array.
2. It gives you fast iteration and fast random access.
3. To state the obvious: It is an ordered collection (by index), but not sorted.
4. The java.util.ArrayList class is one of the most commonly used classes in the Collections Framework. Some of the advantages ArrayList has over arrays are
   * It can grow dynamically.
   * It provides more powerful insertion and search mechanisms than arrays.

Example 1: Basic operations

```
import java.util.*;
class  ArrayList1  {
public static void main(String args[ ]) {
       ArrayList a1 = new ArrayList();

       a1.add("red");
       a1.add("white");
       a1.add("green");
       a1.add(1,"blue");
       a1.add("green");
       System.out.println(a1.size());
       System.out.println(a1);

       System.out.println(a1.remove("bluu"));
       System.out.println(a1.remove("blue"));
       System.out.println(a1.remove(1));
       System.out.println(a1);

       System.out.println(a1.contains("white"));

(I)    Object s1 = a1.get(1);
       System.out.println(s1);
       a1.set(0,"black");
       System.out.println(a1);
}}
```

Note ① method add(), does automatic conversion to data type "Object"

② ArrayList 1.java uses unchecked or unsafe operations.

Recompile with -Xlint: unchecked for details

O/P :-

comes in o/p

5
[red, blue, white, green, green]
false
true
white
[red, green, green]
false
green
[black, green, green]

Note: ArrayList can have duplicates. Order of elements in ArrayList can be predicted.

(replaces)

When we write this stat, "s1" is first converted to object of class Object, So we have to make data type of s1 to class object in statement (I)

Example 2: Adding objects of different types

```
import java.util.*;
class Box
{ }
class  ArrayList2
{
public static void main(String args[ ])
{
        ArrayList a1 = new ArrayList();

        a1.add("red");
        a1.add(24);          // Autoboxing takes place
        a1.add(true);        // Autoboxing takes place
        a1.add(new Box());

        System.out.println(a1.size());
        System.out.println(a1);
}
}
```

Output:

```
Note: ArrayList2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
4
[red, 24, true, Box@6d06d69c]
```

Note:

1. We have objects of different types in ArrayList.
2. Since ArrayList can contain only objects, that's why autoboxing takes place when we insert 24 and true.

Example 3: Using Generics

```
import java.util.*;
class  ArrayList3
{
public static void main(String args[ ])
{
        ArrayList<String>  a1 = new  ArrayList<String>();

        a1.add("red");
        a1.add("white");
        a1.add("green");

        // a1.add(24);  COMPILE-TIME ERROR

        System.out.println(a1.size());
        System.out.println(a1);
}
}
```

See now that note is not displayed.

Example 4: Usage of polymorphic reference, for-each and iterator

```
import java.util.*;
class  ArrayList4
{
public static void main(String args[ ])
{
      List<String>  a1 = new  ArrayList<String>();  // note the difference

      a1.add("red");
      a1.add("white");
      a1.add("green");

   for(String s : a1)
         System.out.print(s+" ");

      System.out.println();

      Iterator itr = a1.iterator();
      while(itr.hasNext())
         System.out.print(itr.next()+" ");
}
}
```

Output:

```
red white green
red white green
```

The following is the summary of the important methods of class ArrayList:

Methods	Meaning
void add(Object obj)	Appends obj at the end of array list
void add(int index, Object obj)	Inserts obj at specified index. Any pre-existing elements at or beyond the point of insertion are shifted ahead. Hence, no overwriting takes place.
void remove(Object obj)	Removes the first occurrence of obj from the array list.
void remove(int index)	Removes from the array list that object which is at position index.
Object get(int index)	Returns the object at position index
void set(int index , Object obj)	The existing element at position index is replaced by obj.

II) Vector

A vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of vector because the synchronized methods add a performance hit you might not need.

III) LinkedList

1. A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly linked to one another.
2. This linkage gives you new methods for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue.
3. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion.
4. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods peek() , poll() and offer().

Example 1: Basic LinkedList operations

```
import java.util.*;
class  LinkedList1
{
public static void main(String args[ ])
{
      LinkedList<String> a1 = new LinkedList<String>();

      a1.add("red");
      a1.add("white");
      System.out.println(a1);

      a1.addFirst("blue");
      a1.addLast("black");
      System.out.println(a1);

      a1.add(0,"green");

      a1.removeFirst();
      a1.removeLast();
      System.out.println(a1);
}
}
```

Set interface

1. A set cares about uniqueness – **it doesn't allow duplicates.**
2. The equals() method determines whether two objects are identical (in which case, only one can be in the set).
3. The three Set implementations are described in the following sections.

Set	Ordered	Sorted
HashSet	No	No
LinkedHashSet	By Insertion Order	No
TreeSet	Sorted	Yes

[Order hai ki nahi hai?] [Sorted hai ki nahi hai?]

I) HashSet

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

Example 1:

```
import java.util.*;
class  HashSet1
{
public static void main(String args[ ])
{
      HashSet<String> a1 = new HashSet<String>();

      a1.add("may");
      a1.add("april");
      a1.add("june");
      a1.add("may");
      System.out.println(a1.size());
      System.out.println(a1);

      a1.remove(2);
      System.out.println(a1);
}
}
```

[no index is set in Set interface]

O/P: 3
[june, may, april]
[june, may, april]

Note:

1) "may" added only once.
2) remove(2) does not report an error, but does not work either.
3) Set<String> a1 = new HashSet<String>(); is also valid – Polymorphic reference
4) See the order of output is not same as order of insertion.

When using HashSet or LinkedHashSet , we should ideally override method hashcode() and equals().

If we don't override hashCode() and equals(), the default object.hashCode() method will allow addition of multiple objects that we might consider **"meaningfully equal"**.

Example 2: Duplicate objects allowed

```
import java.util.*;

class Complex
{
        private  int  x , y ;

        Complex(int  a , int  b)
        { x=a ; y=b; }
}

class  HashSet2
{
public static void main(String args[ ])
{
        HashSet<Complex> a1 = new HashSet<Complex>();

        Complex  c1 = new Complex(2,4);
        Complex  c2 = new Complex(2,4);

        a1.add(c1);
        a1.add(c2);

        System.out.println(a1.size());
        System.out.println(a1);
}
}
```

Cannot detect

[Here the HashSet ~~does not~~ considers 2 objs as duplicates, bcoz class Complex is a user defined data type. If it was Java defined datatype, the HashSet would detect duplicates]

Output:

```
2
[Complex@7852e922, Complex@6d06d69c]
```

Note: Objects duplicates, but still inserted.

II) LinkedHashSet

1. A LinkedHashSet is an ordered version of HashSet that maintains a doubly linked List across all elements.
2. Use this class instead of HashSet when you care about the iteration order.
3. 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.

```
import java.util.*;
class  LinkedHashSet1
{
public static void main(String args[ ])
{
        LinkedHashSet<Integer> a1 = new LinkedHashSet<Integer>();

        a1.add(24);
        a1.add(-58);
        a1.add(46);
        a1.add(-58);

        System.out.println(a1);
}
}
```

O/P: [24, -58, 46]

III) TreeSet

1. The TreeSet is one of two sorted collections (the other being TreeMap).
2. It guarantees that the elements will be in ascending order, according to natural order.
3. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be by using a comparator.
4. As of java 6, TreeSet implements NavigableSet.

```
import java.util.*;
class  TreeSet1
{
public static void main(String args[ ])
{
        TreeSet<Integer> a1 = new TreeSet<Integer>();
        a1.add(24);
        a1.add(-58);
        a1.add(46);
        System.out.println(a1);

        TreeSet<String> a2 = new TreeSet<String>();
        a2.add("may");
        a2.add("april");
        a2.add("june");
        System.out.println(a2);
}
}
```

O/P :-
[-58, 24, 46]
[april, june, may]

Map interface

1. A map is a collection of key-value pairs.
2. The map lets 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.
3. A map cannot have duplicate keys. Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

Map	Ordered	Sorted
HashMap	No	No
Hashtable	No	No
LinkedHashMap	By Insertion Order	No
TreeMap	Sorted	Yes

I) HashMap

1. The HashMap gives you an unsorted, unordered Map.
2. 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; the other maps add a little more overhead.
3. Where the keys land in the Map is based on the key's hashcode, so,like HashSet, the more efficient your hashCode() implementations, the better access performance you'll get.
4. HashMap allows one null key and multiple null values in a collection.

```
import java.util.*;
class  HashMap1
{
public static void main(String args[ ])
{
        HashMap<Integer , String>  a1 = new HashMap<Integer , String>();

        a1.put(546,"Jack");
        a1.put(126,"Jill");
        a1.put(284,"Tim");

        a1.put(284,"Tim");
        a1.put(546,"Jon");
        a1.put(759,"Jack");

        System.out.println(a1);

        System.out.println(a1.get(126));
        System.out.println(a1.get("Jill"));

        System.out.println(a1.containsKey(546));
        System.out.println(a1.containsValue("Tim"));
        System.out.println(a1.containsKey("Tim"));

        System.out.println(a1.isEmpty());
}
}
```

Note: Output unpredictable.

II) Hashtable

1. Hashtable is the synchronized counterpart to HashMap.
2. Another difference, though, is that while HashMap lets you have null values as well as one null key, a Hashtable doesn't let you have anything that's null.

III) LinkedHashMap

1. Like its Set counterpart, LinkedHashSet, the LinkedHashMap collection maintains insertion order (or, optionally, access order).
2. Although it will be somewhat slower than HashMap for adding and removing elements, you can except faster iteration with a LinkedHashMap.

```
import java.util.*;
class  LinkedHashMap1
{
public static void main(String args[ ])
{
      LinkedHashMap<Integer , String> a1 = new LinkedHashMap<Integer , String>();

      a1.put(546,"Jack");
      a1.put(126,"Jill");
      a1.put(284,"Tim");

      System.out.println(a1);
}
}
```

IV) TreeMap

1. A TreeMap is a sorted Map. By default, this means "sorted by the natural order of the elements".
2. Like TreeSet, you can construct a TreeMap with a constructor that lets you give the collection your own rules for what the order should be by using a comparator.
3. As of java 6, TreeMap implements NavigableMap.

```
import java.util.*;
class  TreeMap1
{
public static void main(String args[ ])
{
      TreeMap<Integer , String> a1 = new TreeMap<Integer , String>();
      a1.put(546,"Jack");
      a1.put(126,"Jill");
      a1.put(284,"Tim");
      System.out.println(a1);

      TreeMap<String , Integer> a2 = new TreeMap<String , Integer>();
      a2.put("Jill",128);
      a2.put("Tim",346);
      a2.put("Jack",549);
      System.out.println(a2);
}
}
```

Note: Output in ascending order of key.

Queue interface

(Predicted and Sorted)

1. A Queue is designed to hold a list of "to-dos" or things to be processed in some way.
2. Although other orders are possible, queues are typically thought of as FIFO (first-in first-out).
3. Queues support all off the standard Collection methods and they also have methods to add and subtract elements and review queue elements.

PriorityQueue

1. This class is new as of java 5. Since the LinkedList class has been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList.
2. The purpose of a PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue.
3. A PriorityQueue's elements are ordered either by natural ordering or according to a Comparator. In either case, the elements ordering represents their relative priority.

```
import java.util.*;
class  PriorityQueue1
{
public static void main(String args[ ])
{
      PriorityQueue<Integer> a1 = new PriorityQueue<Integer>();

      a1.add(4);
      a1.add(2);
      a1.offer(6);
      a1.offer(9);
      System.out.println(a1);

      System.out.println(a1.poll());
      System.out.println(a1);

      System.out.println(a1.peek());
      System.out.println(a1);

      a1.clear();
      System.out.println(a1.poll());
}
}
```

add() and offer() work same

[2, 4, 6, 9]

remove and return the 1st element as per sorted order (ascending)

only fetch the 1st element as per sorted order

O/P : [2, 4, 6, 9]
2
[4, 6, 9]
4
[4, 6, 9]
null

Note : After removing element for 1st time, O/P is unpredictable

Sandeep J. Gupta (9821882868)

Sorting Collections and Arrays

Example 1: Basic Sorting and reversing

```
import java.util.*;
class  ALSort1
{
public static void main(String args[ ])
{
        ArrayList a1 = new ArrayList();

        a1.add("red");
        a1.add("white");
        a1.add("green");
        System.out.println(a1);

       Collections.sort(a1);
       System.out.println(a1);

       Collections.reverse(a1);   // Note that sort() and reverse() is in class Collections
       System.out.println(a1);
 }
 }
```

Output:

```
Note: ALSort1java.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[red, white, green]
[green, red, white]
[white, red, green]
```

Note: reverse() can also be used without sort().

Example 2: Incorrect sorting

```
import java.util.*;
class Employee
{
       private  String  name ;
       private  int  code ;
       private  double salary ;

       Employee(String n, int c, double s)
       { name = n;  code = c; salary = s; }

      String getName()
      { return name;}

     int getCode()
     { return code;}

    double getSalary()
    { return salary;}
}
```

```
class ALSort2
{
public static void main(String args[ ])
{
        ArrayList<Employee> a1 = new ArrayList<Employee>();

        Employee e1 = new Employee("Arya",246,45000.5);
        Employee e2 = new Employee("Sansa",915,35000.6);
        Employee e3 = new Employee("Jon",128,55000.7);

        a1.add(e1);
        a1.add(e2);
        a1.add(e3);

        Collections.sort(a1);
        System.out.println(a1);
}
}
```

(Collections.sort() cannot sort user defined objects. It sorts only Java defined & Wrapper objects. If we want to sort User defined obj, we should do follow] [on what basis Sorting should be done?]

Note:
Compilation fails. We get the following message:
error: no suitable method found for sort(ArrayList<Employee>)
Collections.sort(a1);

I) Sorting using Comparable Interface (User defined Sorting)

The Comparable interface is used by the Collections.sort() method and the Arrays.sort() method to sort Lists and arrays of objects.

To implement Comparable, a class must implement a single method, compareTo().

Here's an invocation of compareTo() : int x = thisObject.compareTo(anotherObject);

The compareTo() method returns an int with the following characteristics:

* **Negative** - If thisObject < anotherObject
* **Zero** - If thisObject == anotherObject
* **Positive** - If thisObject > anotherObject

The sort() method uses compareTo() to determine how the List or object array should be sorted. Since you get to implement compareTo() for your own classes, you can use whatever weird criteria you prefer to sort instances of your classes.

```
import java.util.*;
class Employee implements Comparable<Employee>
{
        private String name ;
        private int code ;
        private double salary ;

        Employee(String n, int c, double s)
        { name = n;  code = c; salary = s; }

        String getName()
        { return name;}

        int getCode()
        { return code;}

        double getSalary()
        { return salary;}

        public int compareTo(Employee  e)
        { return  code - e.getCode() ; }
}
class  Comparable1
{
public static void main(String args[ ])
{
        ArrayList<Employee>  a1 = new ArrayList<Employee>();

        Employee e1 = new Employee("Arya",246,45000.5);
        Employee e2 = new Employee("Sansa",195,35000.6);
        Employee e3 = new Employee("Jon",428,55000.7);

        a1.add(e1);
        a1.add(e2);
        a1.add(e3);

        Collections.sort(a1);

        for(Employee  a : a1)
          System.out.println(a.getName()+" "+a.getCode()+" "+a.getSalary());
}
}
```

Output:

```
Sansa 195 35000.6
Arya 246 45000.5
Jon 428 55000.7
```

Some other options for method compareTo():

1) public int compareTo(Employee e)
{ return e.getCode() - code ; }

2) public int compareTo(Employee e)
{ return name.compareTo(e.getName()) ; }

3) public int compareTo(Employee e)
{ return e.getName().compareTo(name) ; }

4) public int compareTo(Employee e)
{ return (int) (e.getSalary() - salary); }

II) Sorting using Comparator Interface

1. The limitation of compareTo() is that we can only implement compareTo() once in a class. So then how do we go about sorting our classes in an order different from what we specify in our compareTo() method?
2. The answer is Comparator interface. The comparator interface gives you the capability to sort a given collection any number of different ways.
3. The other handy thing about the Comparator interface is that you can use it to sort instances of any class - even classes you can't modify. The comparable interface forces you to change the class whose instances you want to sort. This is not the case with Comparator.
4. The comparator interface is also very easy to implement, having only one method: compare().

```
import java.util.*;
class Employee
{
      private String name ;
      private int code ;
      private double salary ;

      Employee(String n, int c, double s)
      { name = n;  code = c; salary = s; }

      String getName()
      { return name;}

    int getCode()
      { return code;}

    double getSalary()
      { return salary;}
}
// See class Employee is not altered.
```

```
class  CodeAsc implements Comparator<Employee>
{
        public int compare(Employee  e1,Employee  e2)
        { return  e1.getCode() - e2.getCode() ; }
}

class  SalaryDsc implements Comparator<Employee>
{
        public int compare(Employee  e1,Employee  e2)
        { return  (int) ( e2.getSalary() - e1.getSalary() ) ; }
}

class  NameAsc implements Comparator<Employee>
{
        public int compare(Employee  e1,Employee  e2)
        { return  e1.getName().compareTo( e2.getName() ) ; }
}

class  Comparator1
{
public static void main(String args[ ])
{
        ArrayList<Employee>  a1 = new ArrayList<Employee>();

        Employee e1 = new Employee("Arya",246,45000.5);
        Employee e2 = new Employee("Sansa",195,35000.6);
        Employee e3 = new Employee("Jon",428,55000.7);

        a1.add(e1);
        a1.add(e2);
        a1.add(e3);

        CodeAsc c = new CodeAsc();
        Collections.sort(a1,c);
        for(Employee  a : a1)
          System.out.println(a.getName()+" "+a.getCode()+" "+a.getSalary());

        System.out.println();

        SalaryDsc  s = new SalaryDsc();
        Collections.sort(a1,s);
        for(Employee  a : a1)
          System.out.println(a.getName()+" "+a.getCode()+" "+a.getSalary());

        System.out.println();

        NameAsc  n = new NameAsc();
        Collections.sort(a1,n);
        for(Employee  a : a1)
          System.out.println(a.getName()+" "+a.getCode()+" "+a.getSalary());
}
}
```

Output:

```
Sansa 195 35000.6
Arya 246 45000.5
Jon 428 55000.7

Jon 428 55000.7
Arya 246 45000.5
Sansa 195 35000.6

Arya 246 45000.5
Jon 428 55000.7
Sansa 195 35000.6
```

Comparing Comparable & Comparator

No	java.lang.Comparable	java.util.Comparator
1	int objOne.compareTo(objTwo)	int compare(objOne , objTwo)
2	Returns negative if objOne < objTwo zero if objOne == objTwo positive if objOne > objTwo	same as Comparable
3	You must modify the class whose instances you want to sort.	You build a class separate from the class whose instances you want to sort.
4	only one sort sequence can be created	many sort sequences can be created.
5	Implemented frequently in the API by: String, Wrapper classes, Date, Calendar etc	Meant to be implemented to sort instances of third-party classes.

Sorting with the Arrays Class

We've been using the java.util.Collections class to sort collections; now let's look at using the java.util.Arrays class to sort arrays.

The good news is that sorting arrays of objects is just like sorting collections of objects.

The Arrays.sort() method can be used in two ways:

* Arrays.sort (arrayToSort)
* Arrays.sort (arrayToSort , comparator)

Searching Collections and Arrays

Example:

```
import java.util.*;
class  ALSearch1
{
public static void main(String args[ ])
{
        ArrayList  a1 = new  ArrayList();
        int  i ;

        a1.add("red");
        a1.add("white");
        a1.add("green");
        a1.add("blue");

        i = Collections.binarySearch(a1,"green");
        System.out.println(i);

        Collections.sort(a1);
        System.out.println(a1);

        i = Collections.binarySearch(a1,"green");
        System.out.println(i);

        i = Collections.binarySearch(a1,"orange");
        System.out.println(i);
}
}
```

Output:

```
Note: ALSearch1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
-1
[blue, green, red, white]
1
-3
```

Note: Arrays.binarySearch() can be used with arrays also.

How binarySearch works ??

The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:

* Searches are performed using the binarySearch() method.

* Successful searches return the int index of the element being searched.

* Unsuccessful searches return an int index that represents the insertion point.

The insertion point is the place in the collection/array where the element would be inserted to keep the collection/array properly sorted. Because positive values and 0 indicate successful searches, the binarySearch() method uses negative numbers to indicate insertion points. Since 0 is a valid result for a successful search, the first available insertion point is -1. Therefore, the actual insertion point is represented as **(-(insertion point) -1).**
For instance, if the insertion point of a search is at element 2, the actual insertion point returned will be -3.

* The collection/array being searched must be sorted before you can search it.

* If you attempt to search an array or collection that has not already been sorted, the results of the search will not be predictable.

* If the collection/array you want to search was sorted in natural order, it must be searched in natural order Usually, this is accomplished by NOT sending a comparator as an argument to the binarySearch() method.

* If the collection/array you want to search was sorted using a comparator, it must be searched using the same comparator, which is passed as the second argument to the binarySearch() method. Remember that Comparators cannot be used when searching arrays of primitives.

Example:

```
import java.util.*;
class  Employee {
       private  String  name ;
       private  int  code ;
       private  double salary ;

       Employee(String n, int c, double s)
       { name = n;  code = c; salary = s; }

       String getName()
       { return name;}

       int getCode()
       { return code;}

       double getSalary()
       { return salary;}
}
```

```
class  CodeAsc implements Comparator<Employee>
{
        public int compare(Employee  e1,Employee  e2)
        { return  e1.getCode() - e2.getCode() ; }
}

class  ComparatorSearch
{
public static void main(String args[ ])
{
        ArrayList<Employee>  a1 = new ArrayList<Employee>();
        int i ;

        Employee e1 = new Employee("Arya",246,45000.5);
        Employee e2 = new Employee("Sansa",195,35000.6);
        Employee e3 = new Employee("Jon",428,55000.7);

        a1.add(e1);
        a1.add(e2);
        a1.add(e3);

        // i = Collections.binarySearch(a1,e2); COMPILE-TIME ERROR

        CodeAsc c = new CodeAsc();
        Collections.sort(a1,c);
        for(Employee  a : a1)
          System.out.println(a.getName()+" "+a.getCode()+" "+a.getSalary());

        System.out.println();

        i = Collections.binarySearch(a1,e2,c);
        System.out.println(i);
}
}
```

Output:

```
Sansa 195 35000.6
Arya 246 45000.5
Jon 428 55000.7

0
```

Navigating (Searching) TreeSets and TreeMaps

As of Java 6, TreeSet is implemented from interface java.util.NavigableSet and TreeMap is implemented from interface java.util.NavigableMap.

Accordingly TreeSet and TreeMap have following methods for purpose of **"navigation"**.

Method	Description
TreeSet Navigation Methods	
TreeSet.ceiling(e)	Returns the lowest element >= e
TreeSet.higher(e)	Returns the lowest element > e
TreeSet.floor(e)	Returns the highest element <= e
TreeSet.lower(e)	Returns the highest element < e
TreeSet.pollFirst()	Returns and removes the first entry
TreeSet.pollLast()	Returns and removes the last entry
TreeSet.descendingSet()	Returns a NavigableSet in reverse order
TreeMap Navigation Methods	
TreeMap.ceilingKey(K)	Returns the lowest key >= k
TreeMap.higherKey(k)	Returns the lowest key > k
TreeMap.floorkey(k)	Returns the highest key <= k
TreeMap.lowerkey(k)	Returns the highest key < k
TreeMap.pollFirstEntry()	Returns and removes the first key/value pair
TreeMap.pollLastEntry()	Returns and removes the last key/value pair
TreeMap.descendingMap()	Returns a NavigableMap in reverse order

Example 1: TreeSet Navigation

```
import java.util.*;
class  TreeSet2
{
public static void main(String args[ ])
{
        TreeSet<Integer> a1 = new TreeSet<Integer>();

        for(int i=2;i<=7;i++)
              a1.add(i);
        System.out.println(a1);                          // [2, 3, 4, 5, 6, 7]

        System.out.println(a1.lower(5));                 // 4
        System.out.println(a1.floor(5));                 // 5

        System.out.println(a1.higher(5));                // 6
        System.out.println(a1.ceiling(1));               // 2

        System.out.println(a1.descendingSet());          // [7, 6, 5, 4, 3, 2]

        System.out.println(a1.pollFirst());              // 2
        System.out.println(a1.pollLast());               // 7
        System.out.println(a1);                          // [3, 4, 5, 6]
}
}
```

Example 2: TreeMap Navigation

```
import java.util.*;
class  TreeMap2
{
public static void main(String args[ ])
{
       TreeMap<Integer , String> a1 = new TreeMap<Integer , String>();
       a1.put(546,"Jack");
       a1.put(126,"Jill");
       a1.put(284,"Tim");
       System.out.println(a1);                       // {126=Jill, 284=Tim, 546=Jack}

       System.out.println(a1.lowerKey(284));         // 126
       System.out.println(a1.floorKey(650));         // 546

       System.out.println(a1.higherKey(284));        // 546
       System.out.println(a1.ceilingKey(284));       // 284

       System.out.println(a1.descendingMap());       // {546=Jack, 284=Tim, 126=Jill}

       System.out.println(a1.pollFirstEntry());      // 126=Jill
       System.out.println(a1.pollLastEntry());       // 546=Jack
       System.out.println(a1);                       // {284=Tim}
}
}
```

Backed Collections

There are some methods in Java which return collections which are backed by the other i.e. changes to one collection affects the other as well. To understand this consider the following examples:

Example 1: Usage of subMap()

```
import java.util.*;
class  BackedCollection1
{
public static void main(String args[ ])
{
1.      TreeMap<String , String> a1 = new TreeMap<String , String>();
2.      a1.put("a","ant");
3.      a1.put("d","dog");
4.      a1.put("f","fish");
5.      a1.put("h","horse");
6.      System.out.println(a1);

7.      System.out.println();

8.      SortedMap<String , String> a2 ;
9.      a2=a1.subMap("d","h");

        /* SortedMap<String , String> a2  = new SortedMap<String , String>();
         is invalid since we cannot use new with interface SortedMap.
         However, following is valid
         SortedMap<String , String> a2 = new TreeMap<String , String>(); */

10.     System.out.println(a2);

11.     System.out.println();

12.     a1.put("g","goat");
13.     a2.put("e","elephant");
14.     System.out.println(a1);
15.     System.out.println(a2);

16.     System.out.println();

17.     a1.put("p","parrot");
18.     // a2.put("t","tiger");
19.     // Exception in thread "main" java.lang.IllegalArgumentException: key out of range
20.     System.out.println(a1);
21.     System.out.println(a2);

22.     System.out.println();

23.     SortedMap<String , String> a3 ;
24.     a3=a1.subMap("d", false, "h", true);
25.     System.out.println(a3);
}
```

Output:

```
{a=ant, d=dog, f=fish, h=horse}

{d=dog, f=fish}

{a=ant, d=dog, e=elephant, f=fish, g=goat, h=horse}
{d=dog, e=elephant, f=fish, g=goat}

{a=ant, d=dog, e=elephant, f=fish, g=goat, h=horse, p=parrot}
{d=dog, e=elephant, f=fish, g=goat}

{e=elephant, f=fish, g=goat, h=horse}
```

Explanation:

1. By the end of statement 5, we have created a TreeMap a1 which is {a=ant, d=dog, f=fish, h=horse}

2. Statement 8 creates another collection a2 which is of type SortedMap. The most important statement is **a2=a1.subMap("d","h");** which uses a method subMap().

3. This method returns a submap of a1 starting at "d", but ending before "h". Hence the **first argument is inclusive but second argument is exclusive**. Now a2 can have any key from "d" to "g". The output of statement 11 confirms this: {d=dog, f=fish}

4. Statement 12 adds "goat" to a1 and statement 13 adds "elephant" to a2. What is surprising is that "goat" and "elephant" were added in both a1 and a2. This is the main concept of "backed collections" – changing one collection will change another too. The output of statements 14 and 15 confirms this:
   {a=ant, d=dog, e=elephant, f=fish, g=goat, h=horse}
   {d=dog, e=elephant, f=fish, g=goat}

5. Statement 17 adds "parrot" to a1 which will not be added to a2 since "parrot" is not within range defined for a2. The output of statements 20 and 21 confirms this:
   {a=ant, d=dog, e=elephant, f=fish, g=goat, h=horse, p=parrot}
   {d=dog, e=elephant, f=fish, g=goat}

6. Statement 18, if not a comment, will throw exception since "tiger" cannot be added to a2 because that would be out of range.

7. Look at statement 23. The method subMap() can also be used as:
   a3=a1.subMap("d", **false**, "h", **true**);

   The boolean arguments are optional.
   false after "d" tells the compiler to NOT include "d".
   true after "h" tells the compiler to include "h".
   Hence now a3 can have any key from "e" to "h". The output of statement 25 shows this : {e=elephant, f=fish, g=goat, h=horse}

Note that a1 was {a=ant, d=dog, e=elephant, f=fish, g=goat, h=horse, p=parrot}

Example 2: Usage of headMap() and tailMap()

```
import java.util.*;
class  BackedCollection2
{
public static void main(String args[ ])
{
        TreeMap<String , String> a1 = new TreeMap<String , String>();
        a1.put("a","ant");
        a1.put("d","dog");
        a1.put("f","fish");
        a1.put("h","horse");
        System.out.println(a1);

        System.out.println();

        SortedMap<String , String> a2 ;
        a2=a1.headMap("d");

        System.out.println(a2);

        a2=a1.headMap("d",true);
        System.out.println(a2);

        a2=a1.tailMap("d");
        System.out.println(a2);

        a2=a1.tailMap("d",false);
        System.out.println(a2);
 }
 }
```

Output:

{a=ant, d=dog, f=fish, h=horse}

{a=ant}
{a=ant, d=dog}
{d=dog, f=fish, h=horse}
{f=fish, h=horse}

Note:

1. The statement a2=a1.headMap("d"); returns a submap from first element of a1 and ends just before "d". That's why o/p is : {a=ant}

2. The statement a2=a1.headMap("d",true); returns a submap from first element of a1 and ends at "d". Hence "d" is also included because of boolean argument true. That's why o/p is : {a=ant, d=dog}

3. The statement a2=a1.tailMap("d"); returns a submap starting at "d" till the last element. That's why o/p is : {d=dog, f=fish, h=horse}

4. The statement a2=a1.tailMap("d",false); returns a submap starting after "d" till the last element. "d" is not included because of boolean argument false. That's why o/p is : {f=fish, h=horse}

Backed Collections with TreeSet

Given below is an example which shows how backed collection will work for TreeSet.

The working of methods subSet(), headSet(), tailSet() is same as subMap(), headMap() and tailMap() respectively.

```
import java.util.*;
class  BackedCollection3
{
public static void main(String args[ ])
{
        TreeSet<Integer> a1 = new TreeSet<Integer>();
        a1.add(2);
        a1.add(4);
        a1.add(6);
        a1.add(8);
        System.out.println(a1);  // [2, 4, 6, 8]

        System.out.println();

        SortedSet<Integer>  a2 ;
        a2=a1.subSet(4,8);

        System.out.println(a2); // [4, 6]

        a2=a1.headSet(6);
        System.out.println(a2); // [2, 4]

        a2=a1.headSet(6,true);
        System.out.println(a2);  // [2, 4, 6]

        a2=a1.tailSet(4);
        System.out.println(a2);  // [4, 6, 8]

        a2=a1.tailSet(4,false);
        System.out.println(a2);  // [6, 8]
}
}
// For your best understanding o/p is given adjacent Sop() statement.
```

Note:

Instead of writing:

```
SortedSet<Integer>  a2 ;
a2=a1.subSet(4,8);
```

we may create a2 as shown:

```
TreeSet<Integer> a2 = new TreeSet<Integer>();
a2=(TreeSet<Integer>)a1.subSet(4,8);
```

But if we use 2nd approach then we cant use a2 with methods like headSet() & tailSet()

How pollFirst() and pollLast() will work with Backed Collections ??

```
import java.util.*;
class  BackedCollectionPoll
{
public static void main(String args[ ])
{
        TreeSet<Integer> a1 = new TreeSet<Integer>();

        a1.add(2);
        a1.add(4);
        a1.add(6);
        a1.add(8);
        System.out.println(a1);  // [2, 4, 6, 8]

        TreeSet<Integer> a2 = new TreeSet<Integer>();
        a2=(TreeSet<Integer>)a1.subSet(2,8);

        System.out.println(a2);  //  [2, 4, 6]

        a1.pollFirst();
        System.out.println(a1);  //  [4, 6, 8]
        System.out.println(a2);  //  [4, 6]

        a1.pollLast();
        System.out.println(a1);  //  [4, 6]
        System.out.println(a2);  //  [4, 6]
}
}
```

Note:

1. a1.pollFirst() removed 2 from a1. Since 2 was also first element of a2, 2 got removed from a2 also.
2. a1.pollLast() removed 8 from a1. But 8 was not the last element of a2, So a2 remains unchanged.
3. **The working of pollFirstEntry() & pollLastEntry() is same in TreeMap.**

The following are the methods which are normally used in Backed Collections.

Method	Description
subMap (s, b*, e, b*)	Returns a submap starting at key s and ending just before key e
headMap (k, b*)	Returns a submap starting from the first element and ending before key k
tailMap (k, b*)	Returns a submap starting at key k (inclusive of k) and ending at the last element.
subset (s, b*, e, b*)	Returns a subset starting at element s and ending just before element e.
headset (e, b*)	Returns a subset starting from the first element and ending before key e.
tailSet (e, b*)	Returns a subset starting at element e (inclusive of e) and ending at the last element.
* Note: These boolean arguments are optional. If they exist, it's a Java 6 method that lets you specify whether the start point and/or end point are exclusive or inclusive. These methods return a NavigableMap or NavigableSet. If the Boolean argument(s) don't exist, the method returns either a SortedSet or a SortedMap.	

Converting Arrays to Lists and vice-versa

A couple of methods allow you to convert arrays to Lists and Lists to arrays.

The Arrays class has a method called ***asList()***.

The Arrays.asList() method copies an array into a List.When you update one of them, the other is updated automatically.

Example 1: Converting Array to List

```
import java.util.*;
class  ArrayToList
{
public static void main(String args[ ])

{
        String  s1[] = {"red","white","blue","yellow"};
        List a1 ;                          // Cannot write ArrayList  a1 ;
        a1 = Arrays.asList(s1);            // asList() converts array to List.
        System.out.println(a1);

        a1.set(2,"green");                 // Changes both List and array
        s1[0] = "black" ;                  // Changes both List and array

        System.out.println(a1);

        for(String s : s1)
          System.out.print(s+" ");
}
}
```

Output:

```
Note: ArrayToList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[red, white, blue, yellow]
[black, white, green, yellow]
black white green yellow
```

Note:

We cannot use add() and remove() methods with List a1. Actually, we cannot use any method which will increase or decrease its size. The change in its size will affect the change in array's size also. But size of array cannot change.

Notice that when we print the final state of the array and the List, they have both been updated with each other's changes.

Example 2: Converting List to Array

```
import java.util.*;
class ListToArray
{
public static void main(String args[ ])
{
        ArrayList a1 = new ArrayList() ;

        a1.add(2);
        a1.add(4);
        a1.add(6);
        System.out.println(a1);

        Object b1[] = new Object[3];
        b1 = a1.toArray();                              // Statement 1
        for(Object b : b1)
          System.out.print(b+" ");

        System.out.println();

        Object b2[] = new Object[3];
        a1.toArray(b2);                                 // Statement 2
        // We can also write b2 = a1.toArray(b2);
        for(Object b : b2)
          System.out.print(b+" ");
}
}
```

Output:

```
Note: ListToArray.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[2, 4, 6]
2 4 6
2 4 6
```

The List and set classes have ***toArray()*** method which converts List to Array.

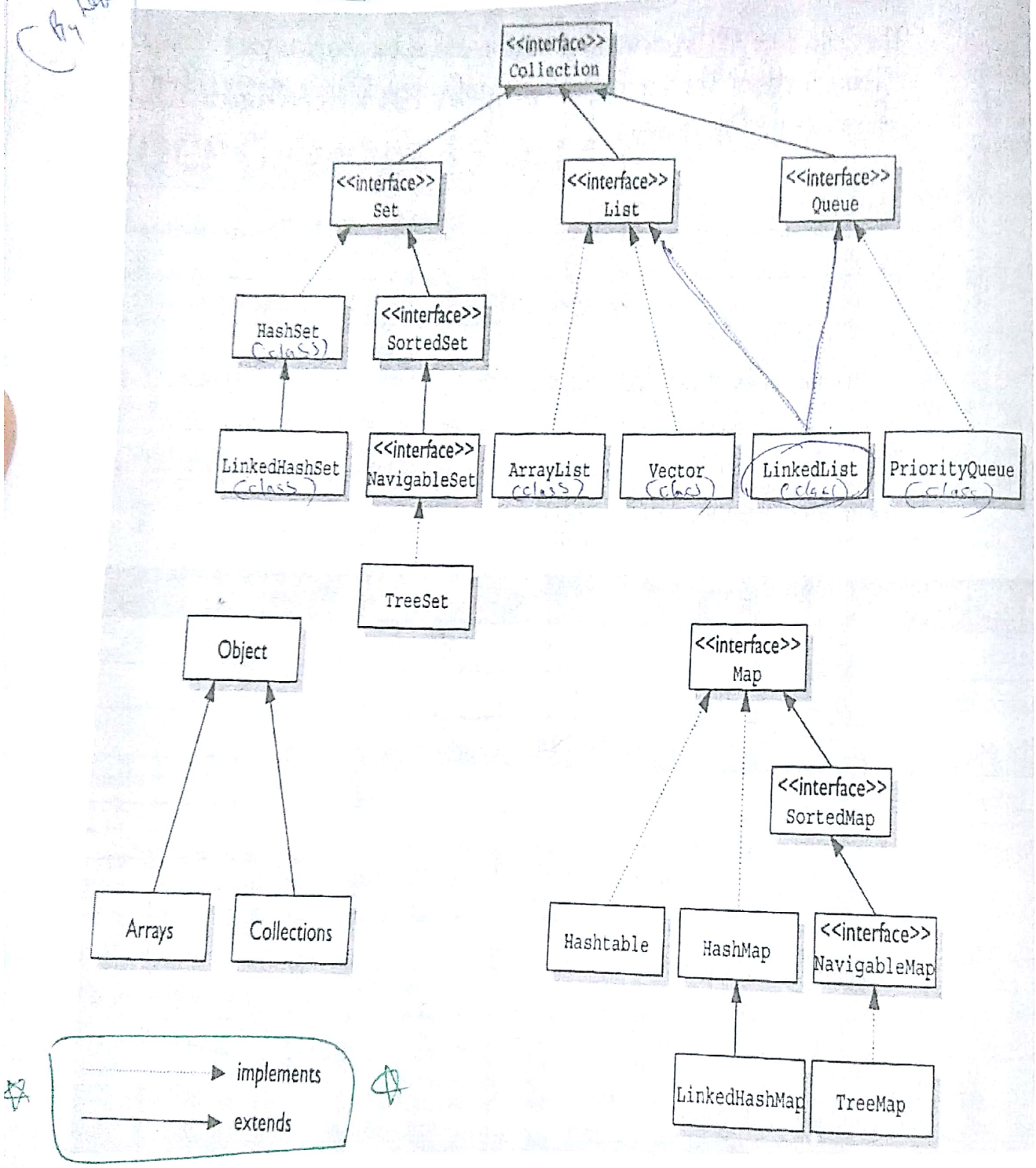
The toArray() method comes in two flavors:

1) one that returns a new Object array, and

2) another that takes the destination array as a parameter.

Statement 1 in the above program uses flavor 1 while statement 2 uses flavor 2.

The interface and class hierarchy for collections



Locale class

1. According to Java API, a locale is a "specific geographical, political and cultural region".
2. The Locale class of the package java.util is used to create objects that can be used in a specific locale.
3. For example, the format to display date, time, numbers and currency is different for different regions. In this situation we can take help of Locale class to write programs that can work in several different international environments.
4. The Locale class defines two constructors:
   Locale(String language)
   Locale(String language , String country)
   The language argument represents an ISO 639 Language code and the country argument represents an ISO 3166 country code.
5. We may create a Locale object even without using a constructor. The class Locale contains many predefined constants which we can use to create objects of commonly used locales. Some examples are:
   GERMANY GERMAN FRANCE FRENCH JAPAN JAPANESE US UK
6. The Locale class contains a method *getDefault()* that returns default locale of the JVM on that particular machine.
7. The Locale class contains methods getDisplayCountry() and getDisplayLanguage() which return the country and language of the invoking Locale object.

```
import java.util.*;
class  Locale1
{
public static void main(String args[ ])
{
Locale Loc1 = new Locale("hi","IN");
Locale Loc2 = new Locale("it");
Locale Loc3 = Locale.GERMANY; // Locale Constant
Locale Loc4 = Locale.getDefault();
Locale Loc5 = new Locale("hi","US");

System.out.println(Loc1+" "+Loc2+" "+Loc3+" "+Loc4+" "+Loc5);
System.out.println(Loc1.getDisplayCountry()+" speaks "+Loc1.getDisplayLanguage());
System.out.println(Loc2.getDisplayCountry()+" speaks "+Loc2.getDisplayLanguage());
System.out.println(Loc4.getDisplayCountry()+" speaks "+Loc4.getDisplayLanguage());
System.out.println(Loc5.getDisplayCountry()+" speaks "+Loc5.getDisplayLanguage());
}
}
```

Op: hi_IN it de_DE en_US hi_US

India speaks Hindi

Speaks Italian

US speaks English

US speaks Hindi

Date class

(Generates current date)

1. The Date class of the package java.util is used to work with date and time.
2. The constructor ***Date()*** creates a Date object representing the current date and time.
3. Commonly used methods of class Date are:

   long getTime():
   It returns the number of milliseconds elapsed since midnight of January 1, 1970.

   void setTime(long time):
   It sets the time and date as specified by argument time which represents the time elapsed in milliseconds since midnight of January 1, 1970.

4. Date also has a constructor ***Date(long millisec)*** which creates a Date object whose time and date are set to argument *millisec* which represents the time elapsed in milliseconds since midnight of January 1, 1970.

```
import java.util.*;
class  Date1
{
public static void main(String args[ ])
{
        Date d1 = new Date();
        System.out.println(d1);

        System.out.println(d1.getTime());

        Date d2 = new Date();
        d2.setTime( d1.getTime() + 24*60*60*1000 );
        System.out.println(d2);

        Date d3 = new Date(24*60*60*1000);
        System.out.println(d3);

}
}
```

o/p:

Sun Oct 02 15:55:29 IST 2016

1475403929954

Mon Oct 03 15:55:29 IST 2016

Fri Jan 02 5:30:00 IST 1970

(Format of class Date)

(Current date)

(next day)

(1 day after 1 Jan 1970)

(GMT is 5:30 more than IST)

(Indian Standard Time)

(no of ms in one day)

Sandeep J. Gupta (9821882868)

Calendar class (abstract class)

1. The Calendar class of the package java.util is used to work with date and time.
2. However, it is an abstract class. So we cannot create any object of its class.
3. In order to create a Calendar instance, we have to use one of its static factory methods called getInstance().
4. A **factory method** is a static method of a class that returns an object of subclass of that class. In case of class Calendar, the getInstance() mostly returns an object of class GregorianCalendar.

```
import java.util.* ;
class Calendar1
{
public static void main(String args[ ])
{
        // Calendar c1 = new Calendar(); --- Error because Calendar is abstract

        Calendar c1 = Calendar.getInstance();
        System.out.println(c1);
}
}
```

DateFormat class

1. The DateFormat class of the package java.text is used to format date and time according to a specific locale.
2. However, it is an abstract class. So we cannot create any object of its class.
3. In order to create a DateFormat instance, we have to use one of its static factory methods.
4. Two commonly used factory methods are:

   getInstance(): It returns date and time using the default format.

   getDateInstance(): It returns the date using the MEDIUM format.

5. We can use 4 static constants as formatting styles: SHORT, MEDIUM, LONG, FULL.
6. Two commonly used methods of DateFormat are:

   String format(Date):
   It formats(converts) a date of type Date into a string.

   Date parse(String):
   It will parse (convert) a string representing a date into a value of type Date.

   If the String cannot be converted into a Date, then method parse() throws ParseException which is a checked exception.

   Being checked, ParseException must be handled or declared.

Example 1:

```
import java.util.*;
import java.text.*;
class  DateFormat1
{
public static void main(String args[ ])
{
      Date d = new Date();

      DateFormat df1 = DateFormat.getInstance();
      System.out.println(df1.format(d));
      DateFormat  df2 = DateFormat.getDateInstance();
      System.out.println(df2.format(d));
      DateFormat  df3 = DateFormat.getDateTimeInstance();
      System.out.println(df3.format(d));
      DateFormat  df4 = DateFormat.getTimeInstance();
      System.out.println(df4.format(d));
}}
```

Note: System.out.println(df1); correct but will display address of df1.

Example 2:

```
import java.util.*;
import java.text.*;
class  DateFormat2
{
public static void main(String args[ ])
{
      Date d = new Date();

      DateFormat  df = DateFormat.getDateInstance();
      System.out.println(df.format(d));

      String  str = df.format(d); // format() returns a string
      System.out.println(str);

      DateFormat  df1 = DateFormat.getDateInstance(DateFormat.SHORT);
      System.out.println(df1.format(d));

      DateFormat  df2 = DateFormat.getDateInstance(DateFormat.MEDIUM);
      System.out.println(df2.format(d));

      DateFormat  df3 = DateFormat.getDateInstance(DateFormat.LONG);
      System.out.println(df3.format(d));

      DateFormat  df4 = DateFormat.getDateInstance(DateFormat.FULL);
      System.out.println(df4.format(d));
}}
```

Note:

DateFormat df1 = DateFormat.getInstance(DateFormat.LONG); is wrong since static field LONG, SHORT, MEDIUM, FULL cannot be used with getInstance()

Example 3: [Locale + Date]

```
import java.util.*;
import java.text.*;
class  DateFormat3
{
public static void main(String args[ ])
{
      Date d = new Date();

      DateFormat  df1 = DateFormat.getDateInstance(DateFormat.FULL);
      System.out.println(df1.format(d));
        Sunday, 2 october, 2016
      Locale Loc1 = new Locale("it");
      DateFormat  df2 = DateFormat.getDateInstance(DateFormat.FULL,Loc1);
      System.out.println(df2.format(d));
        domenica 2 ottobre 2016
     DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL,Locale.GERMAN);
      System.out.println(df3.format(d));
        Sonntag, 2. Oktober 2016
     Locale Loc2 = new Locale("hi","IN"); // Cant use Locale.INDIA
      DateFormat  df4 = DateFormat.getDateInstance(DateFormat.FULL,Loc2);
      System.out.println(df4.format(d));
}      " wild characters will be displayed" [For Hindi language]
}
```

Example 4:

parse() throws an checked exception so if try catch is removed, compilation fails.

```
import java.util.*;
import java.text.*;
class  DateParse1
{
public static void main(String args[ ])
{
 try
 {
      Date d ;
      String s1 ;

      DateFormat  df1 = DateFormat.getDateInstance(DateFormat.SHORT);

      s1 = "13/10/2016";
      d = df1.parse(s1);
      System.out.println(d);
        13/10/2016 op: Thu Oct 13 00:00:00 IST 2016
      s1 = "138/10/2016";
      d = df1.parse(s1);
      System.out.println(d);
op: Wed Feb 15 00:00:00 IST 2017
      /*
      s1 = "13-10-2016" ;
      d = df1.parse(s1);
      System.out.println(d);
      Exception will be thrown */
```

Converts String "13/10/2016" to Date format

d is of Date type Date, So Date Format

138 days are added to 13/10/16

13-10-2016 ← Wrong format. Also, string should start with a number

```
        DateFormat  df2 = DateFormat.getDateInstance(DateFormat.FULL);

        s1 = "13/10/2016" ;
        d = df2.parse(s1);
        System.out.println(d);
    }
    catch(ParseException  e)
    {
        System.out.println("Parse Failed");
    }
}
}
```

NumberFormat class

1. The NumberFormat class of the package java.text is used to format numbers, currencies etc according to a specific locale.
2. However, it is an abstract class. So we cannot create any object of its class.
3. In order to create a NumberFormat instance, we have to use one of its static factory methods.

Example 1: Use getInstance() to format numbers

```
import java.util.*;
import java.text.*;
class  NumberFormat1
{
public static void main(String args[ ])
{
        Locale Loc1 = Locale.getDefault();
        Locale Loc2 = new Locale("de","DE");

        double d = 12345678.62487 ;

        NumberFormat  nf1 = NumberFormat.getInstance(); // US Standard
        System.out.println(nf1.format(d));
        NumberFormat  nf2 = NumberFormat.getInstance(Loc1);
        System.out.println(nf2.format(d));
        NumberFormat  nf3 = NumberFormat.getInstance(Loc2);
        System.out.println(nf3.format(d));
        NumberFormat  nf4 = NumberFormat.getInstance(Locale.ITALY);
        System.out.println(nf4.format(d));
        double m = 1234.624479 ;
        NumberFormat  nf5 = NumberFormat.getInstance();
        System.out.println(nf5.format(m));  // NO COMPOUND ROUNDING
}
}
```

Example 2: Setting Maximum and Minimum

```
import java.util.*;
import java.text.*;
class  NumberFormat2
{
public static void main(String args[ ])
{
      double  d ;
      NumberFormat  nf1 = NumberFormat.getInstance();

      nf1.setMaximumIntegerDigits(4);
      nf1.setMinimumIntegerDigits(2);
      nf1.setMaximumFractionDigits(3);
      nf1.setMinimumFractionDigits(1);

      d = 12345678.62487 ;
      System.out.println(nf1.format(d));
          5678.624
      d = 678.9 ;
      System.out.println(nf1.format(d));
          678.9
      d = 6 ;
      System.out.println(nf1.format(d));
          06.0
      d = 0 ;
      System.out.println(nf1.format(d));
}}        00.0
```

Example 3: Use getNumberInstance() to format numbers [o/p same as eg 1]

```
import java.util.*;
import java.text.*;
class  NumberFormat3
{
public static void main(String args[ ])
{
      Locale Loc1 = Locale.getDefault();
      Locale Loc2 = new Locale("de","DE");

      double d = 12345678.62487 ;

      NumberFormat  nf1 = NumberFormat.getNumberInstance();
      System.out.println(nf1.format(d));

      NumberFormat  nf2 = NumberFormat.getNumberInstance(Loc1);
      System.out.println(nf2.format(d));

      NumberFormat  nf3 = NumberFormat.getNumberInstance(Loc2);
      System.out.println(nf3.format(d));

      NumberFormat  nf4 = NumberFormat.getNumberInstance(Locale.ITALY);
      System.out.println(nf4.format(d));
}}
```

Factory method is diff. Working is same as example 1

Note: Hence getNumberInstance() & getInstance() will behave same.

Sandeep J. Gupta (9821882868)

Example 4: Using getCurrencyInstance() to format currency

```
import java.util.*;
import java.text.*;
class  NumberFormat4
{
public static void main(String args[ ])
{
        Locale Loc1 = Locale.getDefault();
        Locale Loc2 = Locale.US;
        Locale Loc3 = new Locale("de","DE");

        double amount = 4586127.62487 ;

        NumberFormat  nf1 = NumberFormat.getCurrencyInstance();
        System.out.println(nf1.format(amount));

        NumberFormat  nf2 = NumberFormat.getCurrencyInstance(Loc1);
        System.out.println(nf1.format(amount));

        NumberFormat  nf3 = NumberFormat.getCurrencyInstance(Loc2);
        System.out.println(nf3.format(amount));

        NumberFormat  nf4 = NumberFormat.getCurrencyInstance(Loc3);
        System.out.println(nf4.format(amount));
}}
```

Example 5: Using parse()

```
import java.util.*;
import java.text.*;
class  NumberParse
{
public static void main(String args[ ])
{
      String s1 = "1.4  version of java" ;
      String s2 = "-64 is the result"    ;
      String s3 = "java1.4" ;

     NumberFormat  nf = NumberFormat.getInstance();
     try
     {
     System.out.println(nf.parse(s1));
     System.out.println(nf.parse(s2));
     System.out.println(nf.parse(s3));
     }
     catch(ParseException e)
     { System.out.println("Parse Failed"); }
}
}
```

Note: The string should begin with number for successful parsing.

Parsing, Tokenizing and Formatting

A) Parsing

1. Parsing basically means searching. Parsing can be considered as an activity in which we need to search through large amounts of textual data looking for some specific stuff.
2. Parsing in java is normally done using classes java.util.regex.Pattern and java.util.regex.Matcher.
3. Apart from these classes, we will need a 'regex' to facilitate our search. A regex is useful not only in parsing but also in Tokenizing and Formatting.
4. A Regex (**Reg**ular **Ex**pression) is a sequence of characters that define a search pattern. In other words, a regular expression is a special sequence of characters that helps us match or find strings or sets of strings within some given data.
5. For example, we may have a file containing addresses of all students of a particular school. We may now want to find those students who live in an area whose postal code is 400080. In that case, our regex will be "400080".
6. In order to make searching easier, regex may contain metacharacters, quantifiers and dot.

a) Metacharacters

Following are frequently used metacharacters

\d	A digit (0-9)
\D	A non-digit (anything BUT 0-9)
\s	A whitespace character (e.g. space, \t , \n , \f , \r)
\S	A non-whitespace character
\w	A word character (letters (a-z and A-Z), digits, or the "_"[underscore])
\W	A non-word character (everything else)

b) Quantifiers

+	One or more occurences
*	Zero or more occurrences
?	Zero or one occurrences

c) The Dot

1. In addition to the \s, \d, and \w metacharacters that we discussed, we have to understand the "." (dot) metacharacter.
2. When you see this character in a regex expression, it means – "any character can serve here."
3. Consider the following source and pattern:
   source: "ac abc a c"
   pattern: a.c
   will produce the output
   3 abc
   7 a c

   The "." was able to match both the "b" and the " " in the source data.

Example:

```
import java.util.regex.*;
class  Matcher1
{
public static void main(String args[ ])
{
        Pattern p = Pattern.compile(args[0]);
        Matcher m = p.matcher(args[1]);

        System.out.println("Text is "+args[1]);
        System.out.println("Pattern is "+m.pattern());

        while(m.find())
         System.out.println(m.group()+" found at  "+m.start());
}
}
```

Command: java "\\d" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \d
4 found at 2
6 found at 4
7 found at 6

Command: java "\\D" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \D
a found at 0
b found at 1
 found at 3
_ found at 5
+ found at 7
X found at 8

Command: java "\\s" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \s
found at 3

```
Command: java "\\S" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \S
a found at  0
b found at  1
4 found at  2
6 found at  4
_ found at  5
7 found at  6
+ found at  7
X found at  8

Command: java "\\w" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \w
a found at  0
b found at  1
4 found at  2
6 found at  4
_ found at  5
7 found at  6
X found at  8

Command: java "\\W" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \\W
 found at  3
+ found at  7

Command: java "\\w\\d" "ab4 6_7+X"
Output:
Text is ab4 6_7+X
Pattern is \w\d
b4 found at  1
_7 found at  5

Command: java "a.c" "ac abc a c"
Output:
Text is ac abc a c
Pattern is a.c
abc found at  3
a c found at  7

Command: java "a..c" "ac abc axyc a  c"
Output:
Text is ac abc axyc a  c
Pattern is a..c
axyc found at  7
a  c found at  12

Command: java "\\d+" "1 a12 234b"
Output:
```

```
Text is 1 a12 234b
Pattern is \d+
1 found at  0
12 found at  3
234 found at  6

Command: java "a[m-p][aou][r-td-f]" "tim amar amput anar apod anare"
Output:
Text is tim amar amput anar apod anare
Pattern is a[m-p][aou][r-td-f]
amar found at  4
anar found at  15
apod found at  20
anar found at  25

Command: java "abc"  "aecabcacabbc"
Text is aecabcacabbc
Pattern is abc
abc found at  3

Command: java "ab+c"  "aecabcacabbc"
Text is aecabcacabbc
Pattern is ab+c
abc found at  3
abbc found at  8

Command: java "ab*c"  "aecabcacabbc"
Text is aecabcacabbc
Pattern is ab*c
abc found at  3
ac found at  6
abbc found at  8

Command: java "ab?c"  "aecabcacabbc"
Text is aecabcacabbc
Pattern is ab?c
abc found at  3
ac found at  6

Command: java "0[xX][0-9a-fA-F]"  "12  0x  0xFA2  0XG  0x4CD  0X3G"
Output:
Text is 12  0x  0xFA2  0XG  0x4CD  0X3G
Pattern is 0[xX][0-9a-fA-F]
0xF found at  8
0x4 found at  20
0X3 found at  27

Command: java "0[xX]([0-9a-fA-F])+"  "12  0x  0xFA2  0XG  0x4CD  0X3G"
Output:
Text is 12  0x  0xFA2  0XG  0x4CD  0X3G
Pattern is 0[xX]([0-9a-fA-F])+
0xFA2 found at  8
0x4CD found at  20
0X3 found at  27
```

B) Tokenizing (Breaking into tokens and storing)

Tokenizing is the process of taking big pieces of source data, breaking them into little pieces, and storing the little pieces in variables.

We'll look at two classes in the API that provide tokenizing capabilities: Class String (using the split () method) and class Scanner, which has many methods that are useful for tokenizing.

Tokenizing with String.split()

The string class's split () method takes a regex expression as its argument and returns a string array populated with the tokens produced by the split (or tokenizing) process. This is handy way to tokenize relatively small pieces of data.

Everything happens all at once when the split () method is invoked. The source string is split into pieces, and the pieces are all loaded into the tokens string array.

Example:

```
class  Split1  {
public static void main(String args[ ])
{
      String str[] = new String[10] ;
      String  line ;

      line="Boy ate the  apple" ; // Two spaces before apple

      str = line.split(" "); // One Space
      for(String  s : str)
        System.out.println(s);
      System.out.println(str.length);          System.out.println();

      str = line.split("  ");        // Two Spaces
      for(String  s : str)
            System.out.println(s);
      System.out.println(str.length);          System.out.println();

      line="8pm, 9pm, 10pm, 12pm" ;
      str = line.split("pm,");
      for(String  s : str)
        System.out.println(s);
      System.out.println(str.length);          System.out.println();

      line="Apples,Oranges,Bananas" ;
      str = line.split(",");
      for(String  s : str)
        System.out.println(s);
      System.out.println(str.length);          System.out.println();

      line="192.68.246" ;
      str = line.split("\\.");
      for(String  s : str)
        System.out.println(s);
      System.out.println(str.length);          System.out.println();
}}
```

O/P:

```
Boy
ate
the
apple
5

Boy ate the
apple
2

8
9
10
12pm
4
Apples
Oranges
Bananas
3

192
68
246
3
```

Delimiter

In array length we don't write length(), In case of string we write that

Here the 2nd space is considered Delimiter as well as data

If we write only '.' it would have been considered as Parsing topic of Pg 10

Tokenizing with Scanner

The java.util.Scanner class is extensively used for tokenizing. In addition to the basic tokenizing capabilities provided by string.split(), Scanner class offers following features:

* Scanner object can be constructed using files, streams, or string as a source.
* Tokenizing is performed within a loop so that you can exit the process at any point.
* Tokens can be converted to their appropriate primitive types automatically.

Example 1:

```
import java.util.Scanner;
class  Scanner1  {
public static void main(String args[ ])
{
      String  line="20  true  560  6.5  false  42.4" ;

      int  a = 0;            double b = 0;

      Scanner s1 = new Scanner(line) ;

      while(s1.hasNext())
      {
            if(s1.hasNextInt())
              a=a+s1.nextInt();
            else if(s1.hasNextDouble())
              b=b+s1.nextDouble();
            else
              s1.next();
      }
      System.out.println(a);          System.out.println(b);
}}
```

Note:

What if we write s1.nextBoolean() instead of s1.next() ?? Jack

We will get java.util.InputMismatchException for line="20 true 560 6.5 false 42.4"

Example 2:

```
import java.util.Scanner;
class  Scanner2
{
public static void main(String args[ ])
{
      String  line ;

      line="28,456,9,1965" ;
      Scanner s1 =new Scanner(line) ;
      s1.useDelimiter(",");
      while(s1.hasNextInt())
            System.out.println(s1.nextInt());
}}
```

Note:

1) Instead of while loop we could have written s1.nextInt() multiple times.

2) If we don't write s1.useDelimiter(","); then no output is displayed since the default delimiter is whitespace.

C) Formatting

"printf and format are same"

Formatting basically means presenting data in a particular way.

The format() and printf() methods were added to java.io.PrintStream in Java 5. These two methods behave exactly the same way, so anything we say about one of these methods applies to both of them. (The rumor is that printf () was added just to make old C programmers happy.)

The general format of printf() is: **printf ("format string " , arguments);**

The format string may contain the following:

% [arg_index$] [flags] [width] [.precision] conversion character

The values within [ ] are optional.
In other words, the only required elements of a format string are the % and a conversion character.

arg_index:
An integer followed directly by a $, this indicates which argument should be printed in this position.

flags:
While many flags are available, for the exam, you'll need to know:

-	Left-justify this argument
+	Include a sign (+ or -) with this argument
0	Pad this argument with zeroes (instead of blank, add 0)
,	Use locale-specific grouping separators (i.e. , the comma in 123,456)
(	Enclose negative numbers in parentheses
width	This value indicates the minimum number of characters to print.
precision	For the exam, you'll only need this when formatting a floating point number, and in the case of floating-point numbers, precision indicates the number of digits to print after the decimal point.

Conversion Character:
It specifies the type of argument

b boolean
c char
d integer
f floating point
s string

Select 2nd arg [456] here — Select 1st argument

Example: System.out.printf("%2$d + %1$d", 123, 456) ; (These are called arguments)

In the previous example, the only optional values we used were for argument indexing. The 2$ represents the second argument, and the 1$ represents the first argument. (Notice that there's no problem switching the order of arguments .) The d after the arguments is a conversion character (type of the argument.)

Output: 456 + 123

Example 1:

```
class  Printf1
{
public static void main(String args[ ])
{
        int  a = 24;
        byte b = 125;
        float f = 8.5624f;
        String s = "ocjp";
        boolean bn = true;
        char c = '@';

        System.out.printf("a=%d b=%d f=%f s=%s bn=%b c=%c" , a, b, f, s, bn, c);
}
}
```

O/P:- a=24 b=125 & f = 8.562400 s = ocjp bn = true c = @

(V. Imp) **Example 2:** (Nuti):- [%d works for long and short also. There is no %ld in java]

```
class  Printf2
{
public static void main(String args[ ])
{
        int  a = 246;
        float f = 26.7518f;
        String s = "Jon Snow";

        System.out.printf(">%2d %3d %5d<%n" , a, a, a);
```

(Just to represent)

b stands for blank or space

[This is not 5, it is %2d, means min 2 char should be displayed]

> 246 246 bb 246 <

```
        System.out.printf(">%11f<\n" , f);
```

[Same as C rule]

>bb 26.751800<

```
        System.out.printf(">%.2f %.3f %7.2f %07.2f<%n" , f, f, f, f);
```

[max field width - double, float, string only can use it]

> 26.75 26.752 bb 26.75 0026.75<

```
        System.out.printf(">%4s %11s %.3s<%n" , s, s, s);
```

> Jon Snow bbb Jon Snow Jon <

```
        System.out.printf(">%-5d<>%-10s<%n" , a, s);
}
}
```

> 246 bb <> Jon Snow bb <

[Left Alignment of blank and value]

Note: %n and \n are equivalent

Example 3:

```
class Printf3 {
public static void main(String args[ ])
{
      int a = 240 , b=-50 ;
      float x = 25.6f ;
      boolean bn = false ;

      System.out.printf(">%+-6d<%n" , a);
      System.out.printf(">%+-5d<%n" , b);
      System.out.printf(">%(5d<%n" , b);
      System.out.printf(">%(5d<%n" , b);
      System.out.printf(">%0,7d<%n" , 12345);
}}
```

[Display no with it's sign]

(blanks = 2)

b → blank space

[Refer note below]

Note: Minus sign is not displayed when we enclose it in parentheses

Example 4:

```
class Printf4
{
public static void main(String args[ ])
{
      int n1 = 2 , n2 = 4 ;
      String s1 = "apples" , s2="oranges" ;

      System.out.printf("Jack ate %3$d %1$s and %4$d %2$s %n" , s1, s2, n1, n2);
      System.out.printf("Jack ate %3$d %1$s and %4$d" , s1, s2, n1, n2);
      // System.out.printf("Jack ate %3$d %1$s and %4$d %2$s" , s1, s2, n1);
      // java.util.MissingFormatArgumentException

      //System.out.printf("Jack ate %3$d %1$d and %4$d %2$s" , s1, s2, n1, n2);
      // java.util.IllegalFormatConversionException

      // %3$d
      // java.util.UnknownFormatConversionException
}}
```

[Same as \n]

Jack ate 2 apples and 4 oranges

Jack ate 2 apples and 4

[1 argument is missing]

[Char is Unknown]

[Format are different]

The Console class: → [Reads Username and Password]

```
import java.io.Console ;

class  Console1
{
public static void main(String args[ ])
{
      Console c = System.console();  // Cannot use new to create Console variable.

      String s1;
      char p[];

      System.out.print("Customer ID:");
      s1=c.readLine();

     // Input is echoed i.e. input will be displayed on screen

      System.out.print("Password:");
      p=c.readPassword();

     //  Input is not echoed i.e. input will be not be displayed on screen
     //  readPassword() returns a character array, not a string

}
}
```

p should be array of char

O/P :- Customer ID: Jack
Password : ____ (not echoed)