Object Oriented Programming
CS F213

J. Jennifer Ranjani

email: jennifer.ranjani@pilani.bits-pilani.ac.in

Chamber: 6121 P, NAB

Consultation: Appointment by e-mail

BITS Pilani

Pilani Campus

innovate achieve lead

BITS Pilani

Pilani Campus

Query asked during the previous class

Wildcard in Generics

```
class test{
    public static void main(String[] args) {
     List<Integer> list1= Arrays.asList(1,2,3);
     List<Number> list2=Arrays.asList(1.1,2.2,3.3);
     List<Double> list3=Arrays.asList(1.1,2.2,3.3);
      List<String> list4=Arrays.asList("s","j","r");

        printlist(list1);
       printlist(list2);
      printlist(list3);
    printlist(list4);
    }
     private static void printlist(List<Number> list)     {
         System.out.println(list);
    }
}
```

Output:
list1, list3, list4 – compilation error
Type not applicable for the arguements

Wildcard in Generics

```
class test{
    public static void main(String[] args) {
     List<Integer> list1= Arrays.asList(1,2,3);
     List<Number> list2=Arrays.asList(1.1,2.2,3.3);
     List<Double> list3=Arrays.asList(1.1,2.2,3.3);
      List<String> list4=Arrays.asList("s","j","r");

        printlist(list1);
       printlist(list2);
      printlist(list3);
    printlist(list4);
    }
     private static void printlist(List<?> list)     {
         System.out.println(list);
    }
}
```

Output:
[1, 2, 3]
[1.1, 2.2, 3.3]
[1.1, 2.2, 3.3]
[s, j, r]

Upper Bounded Wildcard

```
class test{
    public static void main(String[] args) {
     List<Integer> list1= Arrays.asList(1,2,3);
     List<Number> list2=Arrays.asList(1.1,2.2,3.3);
     List<Double> list3=Arrays.asList(1.1,2.2,3.3);
      List<String> list4=Arrays.asList("s","j","r");

        printlist(list1);
       printlist(list2);
      printlist(list3);
    printlist(list4);
    }
     private static void printlist(List<? extends Number> list)      {
         System.out.println(list);
    }
}
```

Output:
list4 – compilation error
Type not applicable for the arguements

Lower Bounded Wildcard

```
class test{
    public static void main(String[] args) {
     List<Integer> list1= Arrays.asList(1,2,3);
     List<Number> list2=Arrays.asList(1.1,2.2,3.3);
     List<Double> list3=Arrays.asList(1.1,2.2,3.3);
        printlist(list1);
       printlist(list2);
      printlist(list3);
}
    private static void printlist(List<? super Integer> list)     {
        System.out.println(list);
   }
}
```

Output:
list3 – compilation error
Type not applicable for the arguements

Multiple Bounds in Generics

innovate achieve lead

```
class Bound<T extends A & B> {
    private T objRef;
    public Bound(T obj){
        this.objRef = obj;     }

    public void doRunTest(){
        this.objRef.displayClass();     }
}

interface B {
    public void displayClass(); }

class A implements B{
    public void displayClass()     {
        System.out.println("Inside class A");     }
}
```

Multiple Bounds in Generics

```
class C implements B {
    public void displayClass()    {
        System.out.println("Inside class C ");    }
}

public class test {
    public static void main(String a[])    {
        //Creating object of sub class A and
        //passing it to Bound as a type parameter.
        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();
    }
}
```

Note:
The type passed to the class should be of sub type class A and should have implemented interface B

innovate achieve lead

BITS Pilani

Pilani Campus

Coming back to Collections

ArrayList

- Growable Array implementation of List interface.
- Insertion order is preserved.
- Duplicate elements are allowed.
- Multiple null elements of insertion are allowed.
- Default initial capacity of an ArrayList is 10.
- The capacity grows with the below formula, once ArrayList reaches its max capacity.
- newCapacity= (oldCapacity * 3)/2 + 1
- **When to use?**
  - **If elements are to be retrieved frequently. Because ArrayList implements RandomAccess Interface**
- **When not to use?**
  - **If elements are added/removed at specific positions frequently**

ArrayList vs Vector

ArrayList	Vector
There are no synchronized methods.	All methods are synchronized.
No Thread safe: Multiple threads can access the array list at the same time.	Thread safe: Only one thread is allowed to operate on vector object at a time.
Threads are not required to wait and hence performance is high	It increases the waiting time of threads (since all the methods are synchronized) and hence performance is low

LinkedList

- Linked list is implementation class of List interface.
- Underlying data structure is Double linked list.
- Insertion order is preserved.
- Duplicate elements are allowed.
- Multiple null elements of insertion are allowed.

LinkedList - Methods

innovate achieve lead

Constructor/Method	Description
List list = new LinkedList();	It creates an empty linked list.
public boolean add(E e);	It adds the specified element at the end of the list.
public void addFirst(E e);	It adds the specified element in the beginni ng of the list.
public void addLast(E e);	It adds the specified element to the end of the list
public E removeFirst();	It removes and returns the first element fr om the list.
public E removeLast();	It removes and returns the last element fro m the list.
public E getFirst();	It returns the first element from the list.
public E getLast();	It returns the last element from the list.

Stack

- Stack is child class of Vector
- Stack class in java represents LIFO (Last in First Out) stack of objects.

Method	Description
public E push(E item);	Pushes the item on top of the stack
public synchronized E pop();	Removes the item at the top of the stack and returns that item
public synchronized E peek();	Returns the item at the top of the stack
public boolean empty();	Checks whether stack is empty or not
public synchronized int search (Object o);	Returns the position of an object in the stack.

innovate achieve lead

BITS Pilani

Pilani Campus

Set Interface

Set Interface

- The set interface is an unordered collection of objects in which duplicate values cannot be stored.
- The Java Set does not provide control over the position of insertion or deletion of elements.
- Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).

TreeSet

- TreeSet implements the SortedSet interface so duplicate values are not allowed.
- Objects in a TreeSet are stored in a sorted and ascending order.
- TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
- TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add hetrogeneous objects.
- TreeSet serves as an excellent choice for storing large amounts of sorted information because of its faster access and retrieval time.
- TreeSet is basically implementation of a self-balancing binary search tree. Operations like add, remove and search take O(Log n) time. And operations like printing n elements in sorted order takes O(n) time.

TreeSet - Example

innovate achieve lead

```
TreeSet<Integer> set = new TreeSet<Integer>();
    set.add(12);
    set.add(63);
    set.add(34);
    set.add(45);

    Iterator<Integer> iterator = set.iterator();
    System.out.print("Set data: ");
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
```

Output:
Set data: 12 34 45 63

HashSet

- Implements Set Interface.
- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- Execution time of add(), contains(), remove(), size() is constant even for large sets.

HashSet - Example

innovate achieve lead

```
HashSet<Integer> set = new HashSet<Integer>();
    set.add(12);
    set.add(63);
    set.add(34);
    set.add(45);

    Iterator<Integer> iterator = set.iterator();
    System.out.print("Set data: ");
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
```

Output:
Set data: 34 12 45 63

LinkedHashSet

- LinkedHash set extends HashSet and has no member of its own.
- It maintains a linked list of entries in the set in the order of insertion.
- As it implements the Set Interface, duplicate values are not allowed.
- NULL elements are allowed.

LinkedHashSet - Example

innovate achieve lead

```
LinkedHashSet<Integer> set = new LinkedHashSet<Integer>();
    set.add(12);
    set.add(63);
    set.add(34);
    set.add(45);

    Iterator<Integer> iterator = set.iterator();
    System.out.print("Set data: ");
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
```

Output:
Set data: 12 63 34 45

Priority Queue

- PriorityQueue doesn't permit NULL pointers.
- PriorityQueue doesn't have restriction on duplicate elements
- PriorityQueue cannot be created for Objects that are non-comparable
- PriorityQueue are unbound queues.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

Priority Queue - Methods

- The peek() method retrieves the value of the first element of the queue **without removing** it from the queue. For each invocation of the method we always get the same value and its execution does not affect the size of the queue. **If the queue is empty the peek() method returns null.**
- The element() method behaves like peek(), so it again retrieves the value of the first element **without removing it.** Unlike peek), however, **if the list is empty element() throws a NoSuchElementException.**
- The poll() method retrieves the value of the first element of the queue **by removing it from the queue.** . At each invocation it removes the first element of the list and if the list is already empty **it returns null but does not throw any exception.**
- The remove() method behaves as the poll() method, so it **removes the first element** of the list and **if the list is empty it throws a NoSuchElementException.**

Priority Queue - Example

```
class Account{
int acc;
String name;
float cibil;
Account(int acc,String name,float cibil){
this.acc = acc;
this.name = name;
this.cibil = cibil; }
}
class AccCmp implements Comparator<Account>{
public int compare(Account a1,Account a2){
if(a1.cibil==a2.cibil)
return 0;
else if(a1.cibil<a2.cibil)
return 1;
else
return -1; }
}
```

Priority Queue - Example

```
class test {
public static void main(String[] args) {
PriorityQueue<Account> al = new
        PriorityQueue<Account>(5, new AccCmp());

al.add(new Account(123,"Ankit",8.1f));
al.add(new Account(112,"Ashok",4.5f));
al.add(new Account(111,"Ryan",6.5f));

System.out.println("Acc. No. processed on their priority order");
while (!al.isEmpty()) {
System.out.println(al.poll().acc);  }
}
}
```

Output:
Acc. No. processed on their priority order
123
111
112

innovate achieve lead

BITS Pilani

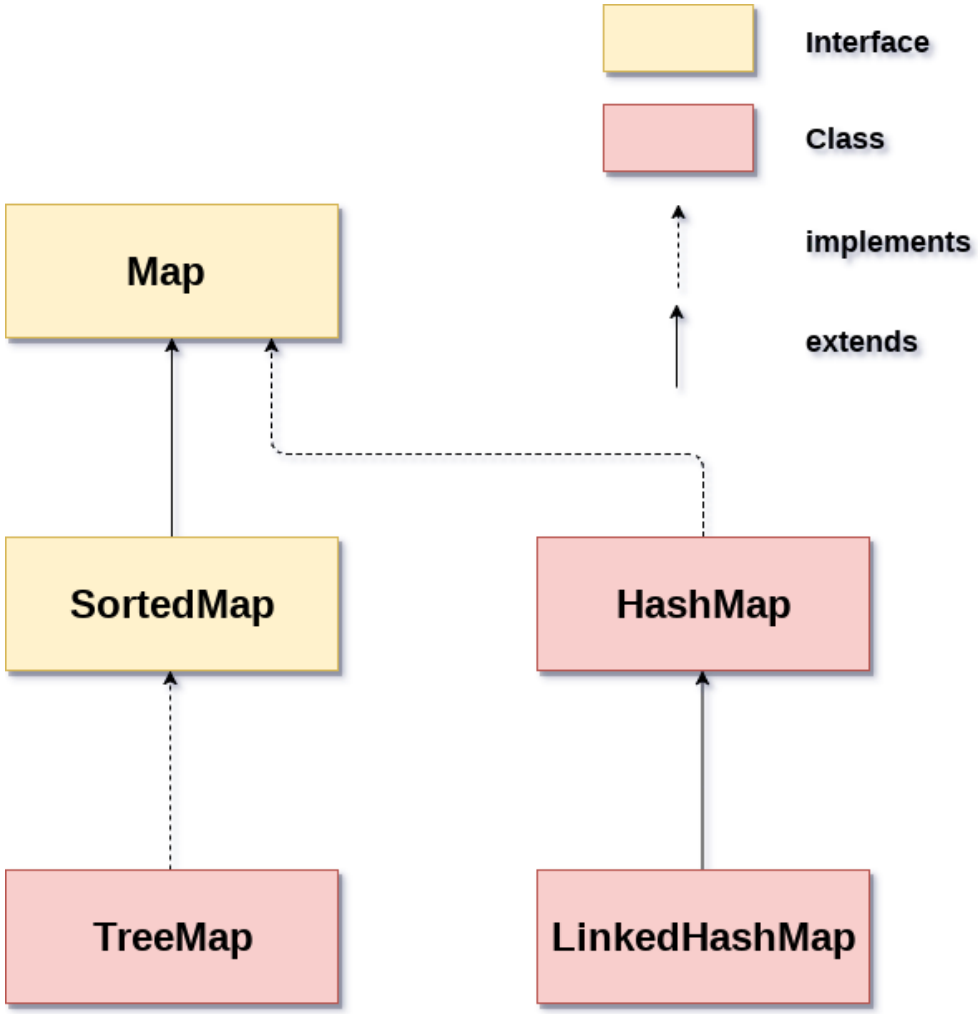
Pilani Campus

Map Interface

Map Interface

- A map contains values on the basis of key i.e. key and value pair.
- Each key and value pair is known as an entry.
- Map contains only unique keys.
- Map is useful if you have to search, update or delete elements on the basis of key.

Map Hierarchy



HashMap – no specific order

LinkedHashMap – maintains insertion order

TreeMap – maintains ascending order

Map Interface

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.
void putAll(Map map)	It is used to insert the specified map in this map.
Object remove(Object key)	It is used to delete an entry for the specified key.
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.

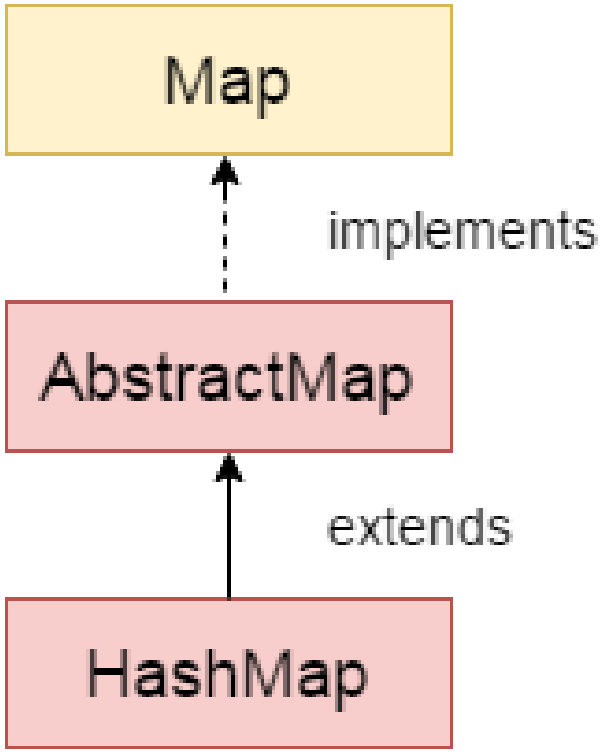
Map.Entry Interface

- Entry is the sub interface of Map.
- It provides methods to get key and value.

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.

HashMap

- A HashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.



Hash Map

```
class test {
public static void main(String[] args) {
  HashMap<Integer,String> hm=new HashMap<Integer,String>();
  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");
  hm.put(102,"RaKul");
  hm.put(null,null);
  hm.put(103,null);
  for(Map.Entry<Integer,String> m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
}
}
```

Output:

null null
100 Amit
101 Vijay
102 RaKul
103 null

Self Study

- LinkedHash Map
- Tree Map
- Hashtable