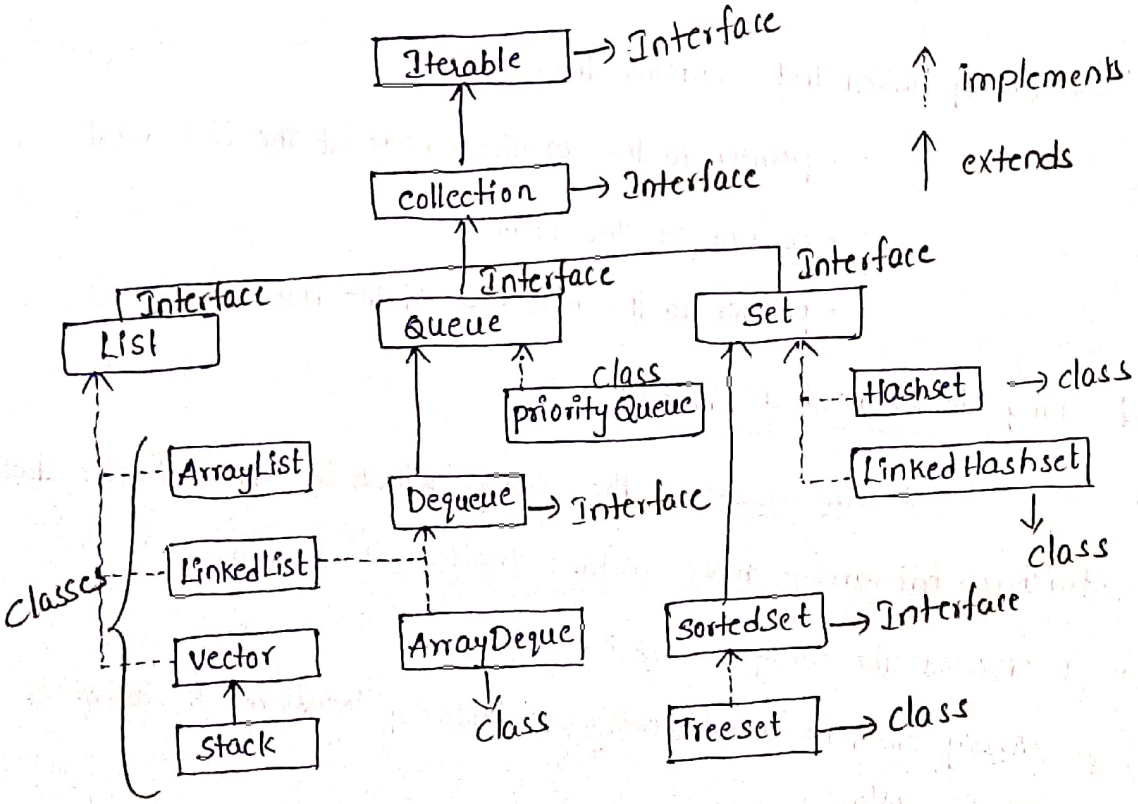
UNIT-I
DATA STRUCTURES IN JAVA

- Linked List
- Stacks
- Queues
- Sets
- Maps
- Generics
  - Generics classes & Type parameters
  - Implementing Generic Types
  - Generic Methods
- Wrapper classes
- Serialization

Collection FrameWork :-

- collection framework provides interfaces and class implementations that enable data handling easy and meaningful.
- Using collections one can store, retrieve and manipulate the data very effectively and easily.

Architecture of Collection Framework



Scanned with CamScanner

Linked List :-

1. A Linked List is a linear datastructure where the elements are stored in contiguous locations and every element is a separate object with a data part and address part.

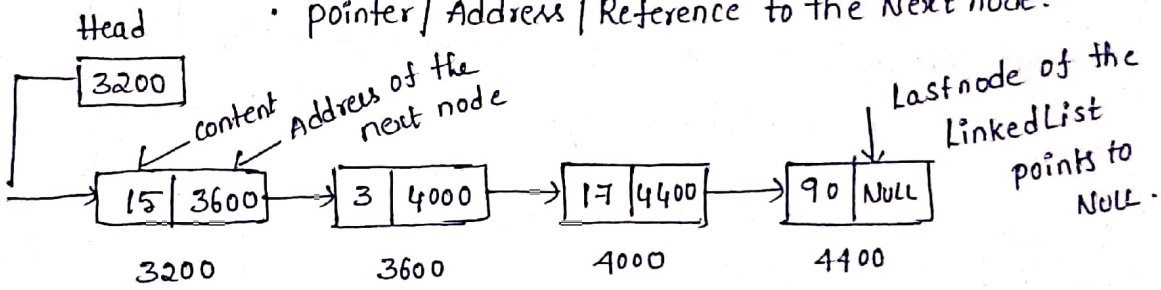
2. It is a datastructure used for collecting the sequential of elements it provids efficient adding and removal of elements in middle of the process.

3. Linked List representation

→ Each Element in the linked list is called the Node.

→ Each node of the linked list contains two items

- Content of the element
- pointer / Address / Reference to the Next node.



→ Doubly linked list contains three parts.

- pointer to the previous node of the linked list.
- content of the element
- pointer to the next node of the linked list.

4. Why do We need a Linked List ?

We are aware of the arrays which is also a linear data structure but arrays have certain limitations such as

1. Size of the array is fixed
2. Array elements need contiguous memory locations to store their values.

Scanned with CamScanner

3) Inserting an element in an array is performance wise expensive. Similarly deleting an element from the array is also performance wise expensive operation.

These limitations are handled in the linked list by providing following features:

1. Linked List allows dynamic memory allocation.
2. Linked List elements don't need contiguous memory locations.
3. Insertion and deletion of elements not performance wise expensive.
4. Linked List class belongs to java.util package.

5. Syntax:

```
LinkedList<String> objectname = new LinkedList<String>();
```

6. LinkedList Methods

1. objectname.addFirst();
2. objectname.addLast();
3. objectname.getFirst();
4. objectname.getLast();
5. objectname.removeFirst();

7. LinkedList class uses iterator class it is also belongs to java.util package.
8. It Encapsulates the references of all elements which are stored in Linked List. It has a hasnext() method to perform this task.

Scanned with CamScanner

9. iterator class is also a generic type. The next() method throws a NoSuchElementException. If it returns true, it means it has a element.

Syntax:

```
ListIterator<String> iterator = new list.ListIterator<String>();
```

10. iterator class has some methods.

```
iterator.add();
iterator.previous();
iterator.Set();
iterator.next();
iterator.remove();
```

Ex:-

```
import java.util.LinkedList;
import java.util.ListIterator;
public class ListDemo
{
    public static void main (String args[])
    {
        LinkedList <String> staff = new LinkedList <string>();
        Staff.add ("Diana");
        Staff.add ("Helina");
        Staff.add ("Romeio");
        Staff.add ("Tiny");
        System.out.println ("The elements in the linked list are:"
                                                    +staff);
        ListIterator <String> iterator = Staff.ListIterator();
         iterator.next();
         iterator.next();
         iterator.add ("Juliet");
         iterator.add ("Nina");
```

DHRT
IDHRT
DIHRT
DHIRT
DHJIRT
DHJNIRT
DHJNRIT
DHJNT

Scanned with CamScanner

```
        iterator.next();
        iterator.remove();
        System.out.println("The elements after modification are:"
                                                    +staff);
    }
}
```

Advantages

1. Linked List is Dynamic Data Structure.
2. A LinkedList can grow and shrink during runtime.
3. Insertion and deletion operations are Easier.
4. Efficient memory utilization, i.e., no need to preallocate memory.
5. Faster Access time, can be expanded in constant time without memory overheal.

Disadvantages

1. Memory Usage : More memory is required to store elements in L.L as compared to array. Because in L.L each node contains a pointer and it requires extra memory for itself.
2. Traversal : Elements or nodes traversal is difficult in L.L. We can not randomly access any element as we do in array by index.
3. Reverse Traversing : In L.L reverse traversing is really difficult. In case of D.L.L its easier but extra memory is required for back pointer hence wastage of memory.

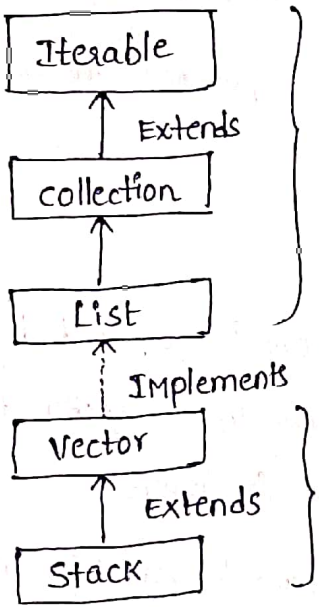
=

Scanned with CamScanner

Stack DataStructure

1. Java collection framework provides a Stack class which implementing Stack data structure.
2. Stack is a Subclass of Vector that implements a Standard last-in, first-out Stack.
3. Stack only defines the default constructor, which creates an empty stack.

   Stack()
4. Stack includes all the methods defined by Vector, and adds Several of it's own.



"Stack class inherits from Vector in Java. It is a thread Safe class we do not need thread Safety."

"Stack implementation is more efficient in a Single threaded environment."

5. The Stack class can perform push for adding/inserting elements into the stack. pop for deleting/removing elements from the stack.
6. Along with these two operations the Stack class provides three more functions of empty(), Search(), peek().
7. While implementing stack class in Java no need to write Stack Empty() and Stack full() conditions like C, C++.

Scanned with CamScanner

Stack methods:

1. boolean empty() : Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2. object peek() : Returns the element on the top of the stack, but does not remove it.
3. object push() : pushes the element onto the stack. Element is also returned.
4. object pop() : Returns the element on the top of the stack, remove it in the process.
5. int Search (object element) : Searches for element in the stack. If found, it's offset from the top of the stack is returned. otherwise -1 is returned.

Example:

```
import java.util.*;
public class StackDemo
{
    static void Showpush ( Stack st, int a)
    {
        st.push (new Integer (a));
        System.out.println ("push(" +a+ ")");
        System.out.println ("Stack:" +st);
    }
    static void Showpop ( Stack st)
    {
        System.out.println ("pop→");
        Integer a = (Integer) st.pop();
        System.out.println (a);
        System.out.println ("Stack: " +st);
    }
```

Scanned with CamScanner

```
public static void main (String args[])
{
    Stack st = new Stack();
    System.out.println("Stack : " +st);
    Showpush(st, 42);
    Showpush(st, 66);
    Showpush(st,99);
    Showpop(st);
    Showpop(st);
    Showpop(st);
    try{
        Showpop(st);
    } catch (EmptyStackException e)
      {
        System.out.println("Empty stack");
      }
  }
}
```

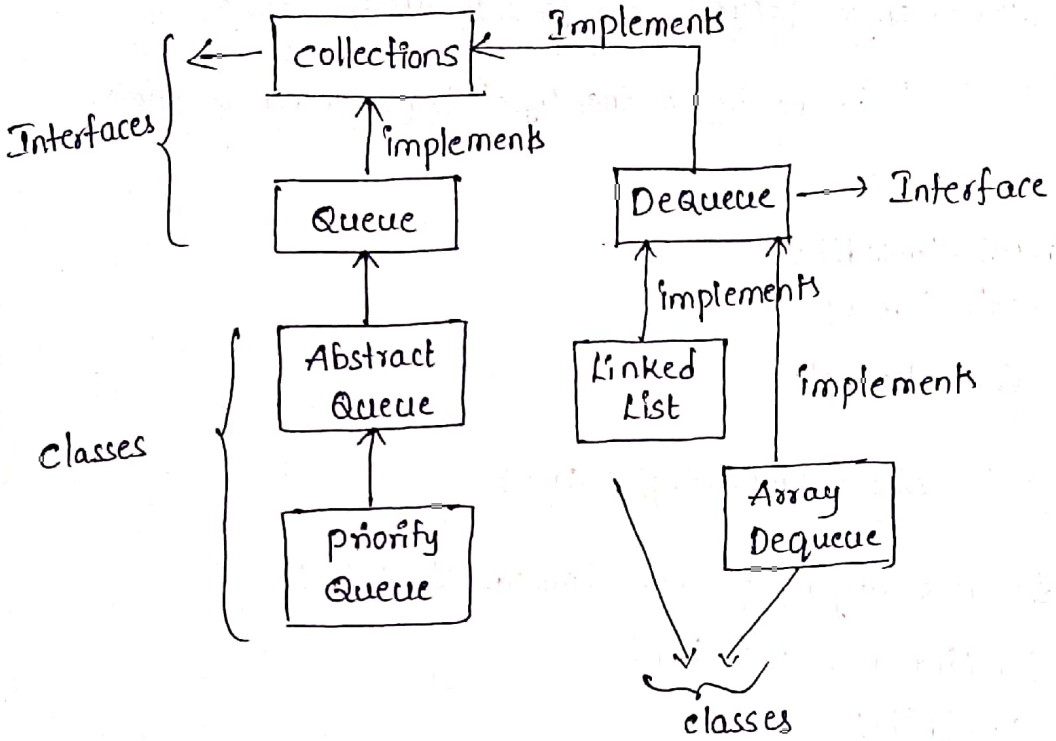
Output

```
Stack: [ ]
push (42)
stack: [42]
push (66)
stack: [42, 66]
push (99)
stack : [42, 66, 99]
pop → 99
stack : [42, 66]
pop → 66
stack: [42]
stack: [ ]
pop → empty stack
```

Scanned with CamScanner

Queue Datastructure:

- The Queue Interface is present in the java.util package and extends the collection interface.
- It is an ordered list of objects with it's use limited to insert elements at the end of the list and deleting elements from the start of the list, i.e., it follows the FIFO or the First-In-First-Out principle.



- Being an interface the queue needs a concrete class for the declaration and the most common classes are the priority Queue and LinkedList in java.
- But both implementations are not thread safe. priorityBlockingQueue is one alternative implementation if thread Safe implementation is needed.

Declaration: The Queue interface is declared as

```
public interface Queue extends Collection
```

[: A thread-safe class is class that guarantees the internal state of the class as well as returned values from methods are correct while invoked concurrently from multiple threads.]

Scanned with CamScanner

Creating Queue object :-

- Since Queue is an interface, objects cannot be created of the type Queue. We always need a class which extends this list in order to create an object. And also, after the introduction of Generics in java 1.5, it is possible to restrict the type of object that can be stored in the Queue.

This type-safe queue can be defined as :

```
Queue<obj> queue = new PriorityQueue<obj>();
```

↓ is the type of the object to be stored in Queue.

Example :

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample
{
    public static void main(String[] args)
    {
        Queue<Integer> q = new LinkedList<>();
        for (int i=0; i<5; i++)
         q.add(i);
        System.out.println("Elements of queue " +q);
        int removedele = q.remove();
        System.out.println("removed element:" + removedele);
        System.out.println(q);
        int head = q.peek();
        System.out.println("head of queue:" +head);
        int size = q.size();
        System.out.println("Size of queue:" +size);
    }
}
```

output :

```
Elements of Queue [0,1,2,3,4]
removed element : 0
[1,2,3,4]
 head of Queue : 1
 Size of Queue : 4
```

Scanned with CamScanner

Operations on Queue Interface :- (Using priority Queue class)

1. Adding Elements : In order to add an element in a queue, we can use the add() method. The insertion order is not retained in the priority Queue. The elements are stored based on the priority order which is ascending by default.

Ex :-

```
import java.util.*;
public class queadd
{
    public static void main(String args[])
    {
        Queue <String> pq = new priorityQueue<>();
        pq.add("BDA");
        pq.add("Cloud computing");
        pq.add("CAO");
        System.out.println(pq);
    }
}
```

Output: [BDA, CAO, Cloud computing]

2. Removing Elements : In order to remove an element from a queue, we can use remove() method. It removes the first occurrence of the object. Apart from that, poll() method is also used to remove the head and return it.

Ex :-

```
import java.util.*;
public class querem
{
    public Static void main(String args[])
    {
        Queue <String> pq = new priorityQueue<>();
        pq.add("BDA");
        pq.add("cc");
        pq.add("CAO");
```

Scanned with CamScanner

```
        pq.remove("cc");
        System.out.println("After remove :" +pq);
        System.out.println("poll method :" +pq.poll());
        System.out.println("Final queue :" +pq);
    }
}
```

Output :

```
Initial Queue [BDA, CC, COA]
After remove : [BDA, COA]
poll method : BDA
Final queue : [COA]
```

Characteristics of a Queue :

1. The Queue is used to insert elements at the end of the queue and removes from the begining of the Queue. It follows the FIFO concept.
2. The Java Queue supports all methods of collection interface including insertion, deletion etc.
3. LinkedList, ArrayBlockingQueue, and priority Queue are the most frequently used implementations.
4. If any null operation is performed on Blocking Queues, NullPointer Exception is thrown.
5. The Queues which are available in java.util package are unbounded Queues.
6. The Queues which are available in java.util.concurrent package are the Bounded Queues.
7. All the Queues except the Dequeue Supports insertion and removal at the tail and head of the queue respectively. The Deques Support element insertion and removal at both ends.

Scanned with CamScanner

priority Blocking Queue :-

- The priority Queue and LinkedList are not thread-Safe.
- priority Blocking Queue is one alternative implementation if thread-safe is needed.
- priority Blocking Queue is an unbounded blocking Queue that uses same ordering rules as class priorityQueue and supplies blocking retrieval operations.
- Since it is unbounded, adding elements may sometimes fail due to resource exhaustion resulting in OutOfMemoryError.

Example:

```
import java.util.concurrent.priorityBlockingQueue;
import java.util.*;

class queue
 {
    public static void main(String args[])
    {
       Queue<Integer> pbq = new priorityBlockingQueue<Integer>();
       pbq.add(10);
       pbq.add(20);
       pbq.add(15);
       System.out.println(pbq.peek());
       System.out.println(pbq.poll());
       System.out.println(Pbq.peek());
    }
 }
```

output :
```
10
10
15
```

Methods of Queue :-

1. add() : is used to add elements at the tail of the Queue.
2. peek() : This method used to view the head of queue without removing it. It returns null if queue is empty.
3. poll() : it removes and returns the head of the queue. It returns null if queue is empty.
4. remove() : removes and not return the object.

Scanned with CamScanner

Set in JAVA

(1) The Set Interface present in the java.util package and extends the collection Interface.

(2) A Set is an unordered collection of objects in which duplicate values can not be stored.

(3) The java platform contains three types of Set implementations.

1. HashSet
2. LinkedHashSet
3. TreeSet

HashSet :

(1) Hashset class is implemented in the collection framework is an inherent implementation of the hashtable datastructure.

(2) It stores its elements in HashTable.

(3) The objects that we insert into the HashSet does not guarantee to be inserted in the same order.

(4) The objects are inserted based on their hashcode.

(5) This class also allows the insertion of NULL elements.

Ex :-

```
import java.util.*;
class hashset
{
  public static void main (String[] args)
  {
    Set<String> h=new HashSet<String>();
    h.add("India");
    h.add("Australia");
    h.add("South Africa");
    h.add("India");
    System.out.println("Set After removing Australia :" +h);
    System.out.println("Iterating over set:");
    Iterator<String> i = h.iterator();
    While (i.hasNext())
      System.out.println(i.next());
  }
}
```

Scanned with CamScanner

O/P: [South Africa, Australia, India]

Set After removing Australia: [South Africa, India]

Iterating over set: South Africa
India

LinkedHashSet:

(1) LinkedHashset class which is implemented in the collections framework is an ordered version of Hashset that maintains a doubly linked list across all the elements.

(2) When the iteration order is needed to be maintained this class is used.

(3) When iterating through a Hashset the order is unpredictable, while a LinkedHashSet iterate through the elements in the order in which they are inserted.

Ex:-
```
import java.util.*;
class linkedhashset
{
    public static void main(String[] args)
    {
        Set<String> lh = new LinkedHashSet<String>();
        lh.add("India");
        lh.add("Australia");
        lh.add("SouthAfrica");
        lh.add("India");
        System.out.println(lh);
        lh.remove("Australia");
        System.out.println("Set After removing Australia :" +lh);

        System.out.println("Iterating over set:");
        Iterator<String> i = lh.iterator();
        While (i.hasNext())
            System.out.println(i.next());
    }
}
```

O/P: [India, Australia, SouthAfrica]

Set After removing Australia : [India, SouthAfrica]

Iterating over set: India
South Africa

Scanned with CamScanner

(3) TreeSet:-

(1) TreeSet class which is implemented in the collections framework an implementation of the SortedSet Interface and SortedSet Interface extends Set Interface.

(2) Treeset uses tree datastructure for storage.

(3) objects are stored in Sorted, ascending order.

(4) But we can iterate in descending order using method TreeSet.descending Iterator(). (Treeset.descendingIterator()).

Ex:-
```
import java.util.*;
class treeset
{
    public static void main(String[] args)
    {
        Set<String> ts = new TreeSet<String>();
        ts.add("India");
        ts.add("Australia");
        ts.add("South Africa");
        ts.add("India");
        System.out.println(ts);
        ts.remove("Australia");
        System.out.println(" Set after removing Australia: " +ts);
        System.out.println("Iterating over set:");
        Iterator<String> i = ts.iterator();
        While(i.hasNext())
            System.out.println(i.next());
    }
}
```

O/P: [Australia, India, South Africa]

Set after removing Australia: [India, South Africa]

Iterating over set:

India

South Africa

Scanned with CamScanner

EnumSet :

(1) Enumset class which is implemented in the collections framework is one of the specialized implementation of Set Interface for use with the enumeration type.

(2) It is a high performance set implementation, much faster than Hashset.

(3) All of the elements in an enum set must come from a single enumeration type that is specified when the set is created either explicitly or implicitly.

Ex:-

```
import java.util.*;
enum Exam
 {
    CODE,
    LEARN,
    CONTRIBUTE,
    QUIZ,
    MCQ
 };
 public class enumset
  {
     public static void main(String[] args)
      {
         Set<Exam> set1;
         set1 = EnumSet.of(Exam.QUIZ,
                           Exam.CONTRIBUTE,
                           Exam.LEARN,
                           Exam.CODE);
         System.out.println("Set 1:" +set1);
      }
  }
```

Op: Set 1 : [CODE, LEARN, CONTRIBUTE, QUIZ]

Scanned with CamScanner

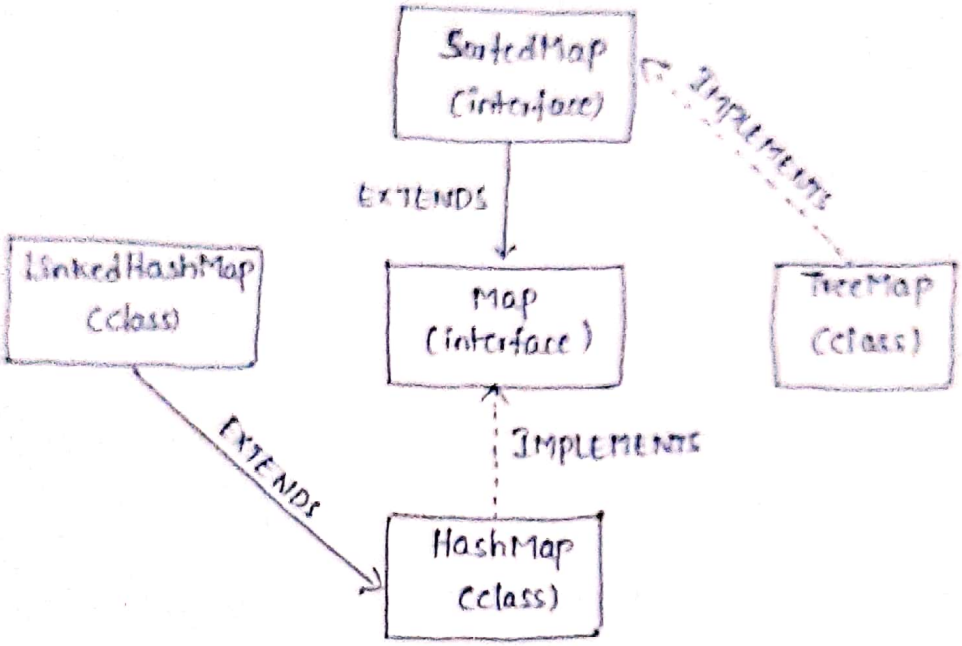
Methods present in the Set Interface

1. add()
   Adds an object to the collection.
2. clear()
   Removes all objects from the collection.
3. contains()
   It returns true if a specified object is an element within the collection.
4. isEmpty()
   Returns true if the collection has no elements.
5. iterator()
   Returns an iterator object for the collection which may be used to retrieve an object.
6. remove()
   Removes the specified object from the collection.
7. Size()
   Returns the number of elements in the collection.
8. hashcode()
   This method is used to get the hashcode value for this instance of the set. It returns an integer value which is the hashcode value for this instance of the set.

Scanned with CamScanner

MAP

1. A Map interface present in java.util package represents a mapping between a key and a value.
2. The Map interface is not a subtype of the collection interface therefore it behaves a bit differently from the rest of collection types.
3. If you need key values to the variable, then we still go through Map.



MAP Hierarchy in JAVA

Characteristics of a Map Interface :

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap but do not like the TreeMap.
2. The order of a map depends on the specific implementations. for example. TreeMap and LinkedHashMap have predictable order, while HashMap does not.
3. There are two interfaces for implementing Map in java. They are. Map and SortedMap, and three classes : HashMap, TreeMap, and LinkedHashMap.

Scanned with CamScanner

Why and When to use Maps?

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform by keys when someone wants to retrieve and update elements by keys.

Some examples are:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manger (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of Students (value).

Creating Map object:

Since Map is an interface, objects cannot be created of the type Map. We always need a class which extends this map in order to create an object. And also, after the introduction of Generics in java 1.5, it is possible to restrict the type of object that can be stored in Map. This type-safe map can be defined as

```
Map map = new HashMap();
```

Operations on MAP

(1) Adding Elements: In order to add an element to the map, we can use the put() method. However, the insertion order is not retained in the hashmap. Internally, for every element, a separate hash is generated and the elements are indexed based on this hash to make it more efficient.

Scanned with CamScanner

Ex:-
```
import java.util.*;
class maps
{
    public static void main(String args[])
    {
        // Default Initialization
        Map<Integer, String> map1 = new HashMap<>();
        // Initialization of a Map using Generics
        Map<Integer,String> map2 = new HashMap<Integer,String>();
        map1.put(1,"Hadoop");
        map1.put(2,"python");
        map1.put(3,"Datastructures");
        map2.put(new Integer(1), "Hadoop");
        map2.put(new Integer(2), "python");
        map2.put(new Integer(3), "Datascience");
        System.out.println(map1);
        System.out.println(map2);
    }
}
```

O/P:- {1 = Hadoop, 2 = python, 3 = Datastructures}
{1 = Hadoop, 2 = python, 3 = Datascience}

2. changing Elements: The elements in the map are indexed using the keys, the value of the key can be changed by simply inserting the updated value for the key for which we wish to change.

Ex:-
```
import java.util.*;
class maps
{
    public static void main(String args[])
    {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(new Integer(1), "Hadoop");
        map.put(new Integer(2), "python");
        map.put(new Integer(3), "DS");
```

Scanned with CamScanner

```
        System.out.println("Initial Map" +map);
        map.put(new Integer(2), "Datastorage");
        System.out.println("Updated Map" +map);
    }
}
```

output:

Initial Map { 1 = Hadoop, 2 = python, 3 = DS }

Updated Map { 1 = Hadoop, 2 = Datastorage, 3 = DS }

3. Removing Element: In order to remove an element from the Map, we can use the remove() method. This method takes the key value and removes the mapping for a key from this map if it is present in the map.

Ex:

```
import java.util.*;
class maps
{
    public static void main(String args[])
    {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(new Integer(1), "DS");
        map.put(new Integer(2), "For");
        map.put(new Integer(3), "storage");
        System.out.println(map);
        map.remove(new Integer(2));
        System.out.println(map);
    }
}
```

output:-

{ 1 = DS, 2 = For, 3 = storage }

{ 1 = DS, 3 = storage }

Scanned with CamScanner

4. Iterating through the Map :- There are multiple ways to iterate through the Map. The most famous way is to use a for-each loop and get the keys. The value of the key is found by using the getValue() method.

Ex:-
```
import java.util.*;
class mapiterate
{
   public static void main (String args[])
   {
      Map<Integer, String> map = new HashMap <Integer, String>();
      map.put (new Integer(1), "Hadoop");
      map.put (new Integer(2), "Bigdata");
      map.put (new Integer(3), "Analytics");
      for(Map.Entry mapElement : map.entrySet())
      {
        int key = (int)mapElement.getKey();
       String value = (String) mapElement.getValue();
       System.out.println ( key + " : " + value);
      }
   }
}
```

output:

1 : [redacted] Hadoop
2 : Bigdata
3 : Analytics

Implementation of MapInterface :-

Implementation of Map interface will done by

(i) HashMap
(ii) LinkedHashMap
(iii) TreeMap

Scanned with CamScanner

HashMap :- HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (key, value) pairs. To access a value one must know its key. This class uses a technique called "Hashing". Hashing is a technique of converting a large String to small String that represents the same string. A shorter value helps in indexing and faster searches.

Ex:-
```
import java.util.*;
public class hashmap
{
    public static void main(String args[])
    {
        Map<String, Integer> map = new HashMap<>();
        map.put ("vishal", 10);
        map.put ("Sachin", 30);
        map.put ("vaibhav", 20);
        for (Map.Entry<String, Integer> e : map.entrySet())
            System.out.println(e.getKey() + " " + e.getValue());
    }
}
```

Output:
```
vaibhav 20
vishal 10
Sachin 30
```

LinkedHashMap :- LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provides the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

Scanned with CamScanner

Ex:-
```
import java.util.*;
public class linkedhashmap
{
    public static void main(String[] args)
    {
        Map<String, Integer> map = new LinkedHashMap<>();
        map.put("Vishal", 10);
        map.put("Sachin", 30);
        map.put("vaibhav", 20);
        for(Map.Entry<String, Integer> e: map.entrySet())
            System.out.println(e.getKey() + " " + e.getValue());
    }
}
```

output:

Vishal 10
Sachin 30
vaibhav 20

TreeMap :- The TreeMap in java is used to implement Map interface and NavigableMap along with the Abstract class. The map is sorted according to the natural ordering of it's keys, or by a comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and ~~Stro~~ storing the key-value pairs.

Ex:-
```
import java.util.*;
public class treemap
{
    public static void main(String[] args)
    {
        Map<String, Integer> map = new TreeMap<>();
        map.put("vishal", 10);
        map.put("Sachin", 30);
        map.put("vaibhav", 20);
```

Scanned with CamScanner

```
for(Map.Entry<String, Integer> e : map.entrySet())
    System.out.println(e.getKey() + "  " + e.getValue());
  }
}
```

output:

```
Sachin 30
vaibhav 20
Vishal 10
```

Methods in MapInterface

#	Method	Description
1	void clear()	Removes all key/value pairs from the invoking map.
2	boolean containsKey()	Returns true if the invoking map contains K as a key. Otherwise returns false.
3	boolean containsValue()	Returns true if the map contains V as a value. otherwise, returns false
4	entrySet()	Returns a set that contains the entries in the map.
5	boolean equals()	Return true if map contains same entries otherwise returns false.
6	Object get()	Returns the value associated with the key K.
7	int hashCode()	Returns the hashcode for the invoking map.
8	boolean isEmpty()	Returns true if the invoking map is empty otherwise returns false.
9	Set keySet()	Returns a set that contains the keys in the invoking map.
10	Object put(K,V)	put an entry in the invoking map, overwriting any previous value associated with the key. Returns null if the key did not already exist. otherwise, the previous value linked to the key is returned
11	void putAll()	put all the entries from m into this map.
12	object remove()	Removes the entry whose key equals K.
13	int size()	Returns the no. of key/value pairs in the map
14	Collection values()	Returns the collection containing the values in the map

Scanned with CamScanner

Generics:

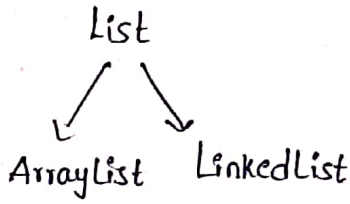
What is Generics: Generics are used for provide "type-safety", it gives the best solution to type casting problems.

Where we need type-safety!

case 1:
```
String[] S = new String[12000];
S[0] = "Ravi";
S[1] = "Madhavi";
S[2] = new Integer(10); ⟹ When we are compile this code
```
we get compile time error.
incompatible types found: java.lang.
Integer ~~required~~ Exception
java.lang.String is required.

case 2: If we are used list from collection framework introduced in JDK 1.4 V

```
List list = new ArrayList();
list.add("Hadoop");
list.add(new Integer(5)); //OK
String str = (String) list.get(0); // type casting leading to "ClassCastException"
String str = (String) list.get(1);          at runtime.
```



Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer.

So, to avoid the ClassCastException at runtime and we don't need to do type-casting by using Generics which are provided in JDK 1.5V.

Scanned with CamScanner

Generic Class: We can define our own classes with generics type.

A generic type is a class or interface that is parameterized over types.

We use angle brackets (< >) to specify the type parameter.

EX:

```
public class GenericsType<T>
{
    private T t;
    public T get()
    {
        return this.t;
    }
    public void set(T t1)
    {
        this.t = t1;
    }
    public static void main(String args[])
    {
        GenericsType<String> type = new GenericsType<String>();
        type.set("Hadoop"); //valid
        GenericsType type1 = new GenericsType(); //raw type
        type1.set("Hadoop"); //valid
        type1.set(10); //valid and autoboxing support
    }
}
```

Java Generic Type:

- Java Generic Type Naming Convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language.
- So generics also come with its own naming conventions.
- Usually type parameter names are single, uppercase letters to make it easily distinguishable from java variables.
- The most commonly used type parameter name are

Scanned with CamScanner

* E – Element (used in ArrayList, set etc.,)
* K – key (used in Map)
* N – Number
* T – Type
* V – Value (used in Map)

Java Generic Method: Sometimes we don't want whole class to be parameterized, in that case we can create java generics method. Since constructor is a special kind of method, we can use generics type in constructors too.

Ex:

```
class GenericsType <T>
{
    private T t;
    public T get()
    {
        return this.t;
    }
    public void set (T t1)
    {
        this.t = t1;
    }
}
public class GenericsMethods
{
    public static <T> boolean isEqual (GenericsType<T> g1, GenericsType<T> g2)
    {
        return g1.get().equals(g2.get());
    }
    public static void main (String args[])
    {
        GenericsType <String> g1 = new GenericsType<>();
        g1.set ("Hadoop");
        GenericsType <String> g2 = new GenericsType<>();
        g2.set ("Hadoop");
        boolean isequal = GenericsMethods.<String>isEqual(g1,g2);
        System.out.println(isequal);
    }
}
```

Scanned with CamScanner

Wrapper classes :

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper classes :

1. They convert primitive data types into objects. objects are needed if we wish to modify the arguments passed into a method. (Because primitive types are passed by value.)
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the collection framework, such as ArrayList and Vector, store only objects (reference types) and non primitive types.
4. An object is needed to support Synchronization in multithreading.

Primitive Datatypes and their Corresponding Wrapper Class

primitive Data type	Wrapper Class
char	Character
byte	Byte
Short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

Scanned with CamScanner

Autoboxing : Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example - conversion of int to Integer, long to Long, double to Double etc.

Ex:-
```
import java.util.ArrayList;
class Autoboxing
{
    public static void main(String[] args)
    {
        char ch = 'a';
        Character a = ch; // primitive to object
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(25); //Auto boxing bcz ArrayList can understand only object.
        System.out.println(arrayList.get(0));
    }
}
```

output :- 25

Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to it's corresponding primitive type is known as unboxing. For example - conversion of Integer to int, Long to long, Double to double etc.

Ex:
```
import java.util.ArrayList;
class Unboxing
{
    public static void main(String[] args)
    {
        Character ch = 'a';
        char a = ch;
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);
        int num = arrayList.get(0);
        System.out.println(num);
    }
}
```

output : 24

Scanned with CamScanner

Wrapping and UnWrapping :-

```
public class WrappingUnWrapping
{
    public static void main(String args[])
    {
      byte grade = 2;
      int marks = 50;
     float price = 8.6f;
    double rate = 50.5;

     Byte g1 = new Byte(grade);
    Integer m1 = new Integer(marks);
     Float f1 = new Float(price);
     Double r1 = new Double(rate);

     System.out.println("values of Wrapper objects (printing as objects)");
    System.out.println("Byte object g1 : " + g1);
    System.out.println("Integer object m1 : " + m1);
    System.out.println("float object f1 : " + f1);
    System.out.println("Double object r1 : " + r1);
     byte bv = g1.byteValue();
     int iv = m1.intValue();
    float fv = f1.floatValue();
    double dv = r1.doubleValue();
    System.out.println("Unwrapped values (printing as data types)");
    System.out.println("byte value, bv: " + bv);
    System.out.println("int value, iv: " + iv);
    System.out.println("float value, fv: " + fv);
    System.out.println("double value, dv: " + dv);
    }
}
```

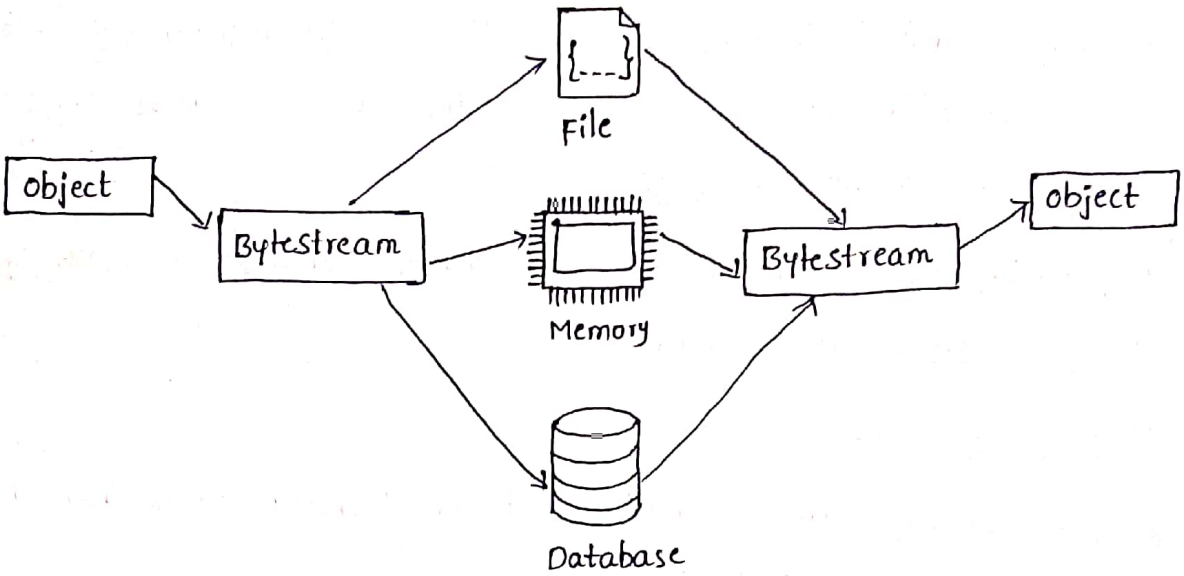
o/p : values of wrapper objects (printing as objects)

Byte object g1: 2
Integer object m1 : 50
float object f1 : 8.6
Double object r1 : 50.5

unwrapped values (printing as data types)
byte value, bv: 2
int value, iv : 50
float value, fv : 8.6
double value, dv : 50.5

Scanned with CamScanner

Serialization: Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



- After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes ~~as well as information~~ that represent the object and it's data can be used to recreate the object in memory.
- Most impressive is that the entire process is JVM independent, i.e., an object can be serialized on one platform and deserialized on an entirely different platform.
- Classes "ObjectInputStream" and "ObjectOutputStream" are high-level streams that contain the methods "readObject()" and "writeObject()" method for serializing an object.
- `public final void WriteObject(Object x) throws IOException`
  - for serializing an object
- `public final Object readObject() throws IOException, ClassNotFoundException`
  - for deserializing an object.

Scanned with CamScanner

Serializable Interface :

A class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All the fields in the class must be serializable. If a field is not Serializable, it must be marked 'transient'.
- 'Serializable' is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability.
- The 'ObjectOutputStream' class is used to Serialize an object by using method 'WriteObject'.

points to remember

1. If a parent class has implemented Serializable interface then child class doesn't need to implement.
2. only non-static data members are saved via Serialization process.
3. Static data members and transient data members are not saved via serialization process.
4. constructor of object is never called when an object is deserialized.
5. Associated objects must be implementing Serializable interface.

Ex:-

```
import java.io.*;
public class SerializeDemo
{
    public static void main(String [] args)
    {
        Employee e = new Employee();
        e.name = "Madhavi Latha";
        e.address = "Ongole";
        e.SSN = 11122333;
        e.number = 101;
        try
        {
```

Scanned with CamScanner

```
        FileOutputStream fileOut = new FileOutputStream("H://employee.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.WriteObject(e);
        out.close();
        fileOut.close();
        System.out.printf("Serialized data is saved in H://employee.ser");
      }
      Catch(IOException i)
      {
         i.printStackTrace();
      }
   }
}
```

→ it is a method of Throwable class it gives perfect trace of an error it is extended from Exception class.

Deserializing an object:

```
import java.io.*;
public class DeserializeDemo
{
   public static void main(String[] args)
   {
      Employee e = null;
      try {
         FileInputStream fileIn = new FileInputStream("H://employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      } catch(IOException i)
        {
           i.printStackTrace();
           return;
        } catch(ClassNotFoundException e)
          {
             System.out.println("Employee class not found");
             e.printStackTrace();
             return;
          }
```

Scanned with CamScanner

```
        System.out.println("Deserialized Employee---");
        System.out.println("Name: " + e.name);
        System.out.println("Address: " + e.address);
        System.out.println("SSN: " + e.SSN);
        System.out.println("Number: " + e.number);
    }
}
```

O/P: Deserialized Employee---

Name : ~~Bharat Kumar~~ MadhaviLatha

Address : Ongole

SSN : 0

Number : 101

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM, can't find a class during the deserialization of an object, it throws a ClassNotFound Exception.
- The value of the SSN field was 11122333 when the object was Serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Scanned with CamScanner