List of Concepts Involved:

- Introduction to Map in Java
- Map Hierarchy
- HashMap
- Other In-Built classes and Inbuilt methods under Map Hierarchy
- Need of Generics and Basics of Generics
- More on Generics in Java
- Collections class and it's inbuilt methods in Java
- Comparator vs Comparable Interface

I) Introduction to Map :-

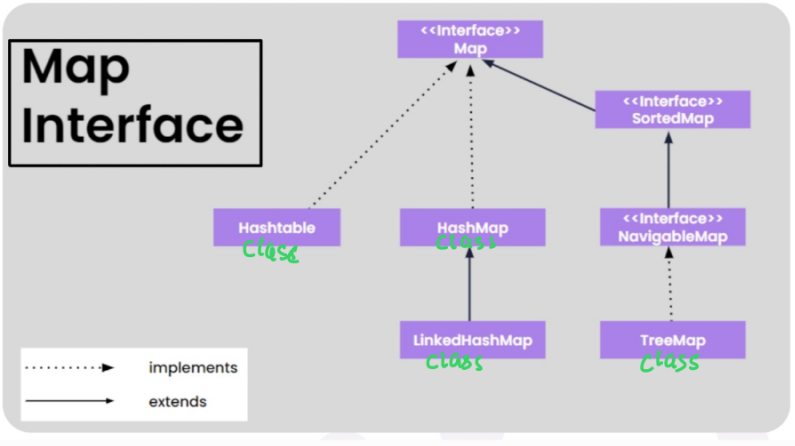
: maps in java are used to store group of individual objects as key-value pair.

- Key must be unique (no duplicates)
- Value can be both unique or non-unique
- Every key-value pair in a map is called as "Entry"

A map is useful if you want to search, update or delete elements on the basis of key.

II) Map Hierarchy

map does not extends or implements any classes or interfaces of Collection Hierarchy.



III. Hash Map and Linked Hash Map

3.1 Hash Map :-

- Order of insertion not preserved (use Linked Hash Map for it)
- Key and Value can also be "null".
- Creating Hash Map :-

```
HashMap hm = new HashMap(); //create HashM object
hm.put(01,"asit"); //use put method to insert "key value pair
hm.put(02,"Amit");
hm.put(03,"Rohan");
```

- It is thread safe.

3.2 Linked Hash Map :-

Order of insertion is preserved

IV. Accessing The Objects in Map

- printing is different from accessing the data
- no iterator() method in Map

Creating HM :- `Map map = new HashMap();`

- `map.get(4)` :- to access value of key = 4.
- `map.keySet()`; returns a set of keys which can be further iterated upon

```
Ex:- Set keyset = map.keySet();
     Iterator itr = keyset.iterator();
     while (itr.hasNext())
     {
       sop(itr.next());
     }
```

- `map.values()` :- returns a collection cuz Set only allow unique values

Example `Collection values = map.values`

```
Iterator itr = values.iterator();

while (itr.hasNext())
{
    sop (itr.next())
}
```

- entrySet();

```java
import java.util.HashMap;

class Student{
    private String name;
    private int age;
    private String city;

    public Student(String name, int age, String city)
    {
        this.name = name;
        this.age = age;
        this.city = city;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
public class Launch4_Hm {
    public static void main(String[] args) {
        /*Creating Student objects */
        Student st1 = new Student("Asit", 24, "Delhi");
        Student st2 = new Student("Chris", 42, "Delhi");
        Student st3 = new Student("Robert", 49, "Delhi");

        /*Stroring above objects in a map */
        HashMap map = new HashMap();
        map.put(1,st1);
        map.put(2,st2);
        map.put(3,st3);
        System.out.println("map: "+map);
    }
}
```

```
map: {1=Student@2f92e0f4, 2=Student@28a418fc, 3=Student@5305068a}
PS C:\Users\Euphor\Documents\JAVA\16_Maps_and_Generics>
```

Observing we notice that the map is storing the object reference rather than the instances

Why is HashMap storing references rather than the instances?

Reason:-

In Object Class of java there is a method called toString method :-

```
Class Object
{
    public String toString()
    {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
}
```

This toString() method is called by System.out.println();
So if this to string method is not overridden in the above Student class then Sop will call the Object class's toString method and return reference "Student@<hexCode>".

above program with toString() method Overridden.

HashTable:

- no duplicate keys
- order of insertion not preserved
- load factor 0.75 i.e dynamic memory, it doubles when 75% memory is filled
- Not Thread Safe (HashMap is thread safe)
- No "null" value allowed

TreeMap:-

- order of insertion not preserved
- key in sorted order

HashMap vs WeakHashMap :-

Generics:

- There is no need of typecasting and down casting in array cuz we know Array provide "typesaftey" cuz only one type of data can be stored in an array.

Ex:-

```
// typesaftey
String ar[] = new String[10];
ar[0] = "asit";
ar[1] = "Shashni";
String name1 = ar[0]
String name2 = ar[1]
```

- In ArrayList there is no typesaftey cuz ArrayList can hold heterogenous type of data as objects So we have to do downcasting of object data

Ex:-

```
// no typesaftey
ArrayList al = new ArrayList();
al.add("Asit"); al.add("Shashni");
al.add(10);
String n1 = (String) al.get(0); // type casting
String n2 = (String) al.get(1);
String n3 = (String) al.get(2);
```

Now there will be no exception in the compilation but an exception will be given at runtime c/a "ClassCastException". Runtime exception is a very big problem for a programmer. To solve this problem "Generics" concept can be used to give error at compile time and not at runtime

5☆→ there is no need of type casting after using generics

Example:-

```
ArrayList<String> al = new ArrayList<String>();
al.add("Asit"); al.add("shashni"); al.add("abcd");
```

error

```
String name1 = al.get(0);   // no typecasting = (String)al.get(0)
String name2 = al.get(1);
```

List of Concepts Involved:

- Introduction to Map in Java
- Map Hierarchy
- HashMap
- Other In-Built classes and Inbuilt methods under Map Hierarchy
- Need of Generics and Basics of Generics
- More on Generics in Java
- Collections class and it's inbuilt methods in Java
- Comparator vs Comparable Interface

Introduction to Map in Java

Map

To represent a group of individual objects as a key value pair then we need to opt for Map(I).

- It is not a child interface of Collection.
- If we want to represent a group of Objects as a key-value pair then we need to go for Map.
- Both keys and values are Objects only
- Duplicate keys are not allowed but values are allowed.
- Key-value pair is called "Entry".

Map interface

It contains 12 methods which is common for all the implementation Map Objects
a. Object put(Object key,Object value)
b. void putAll(Map m)
c. Object get(Object key)
d. Object remove(Object key)
e. boolean containsKey(Object key)
f. boolean containsValue(Object value)
g. boolean isEmpty()
h. int size()
i. void clear()

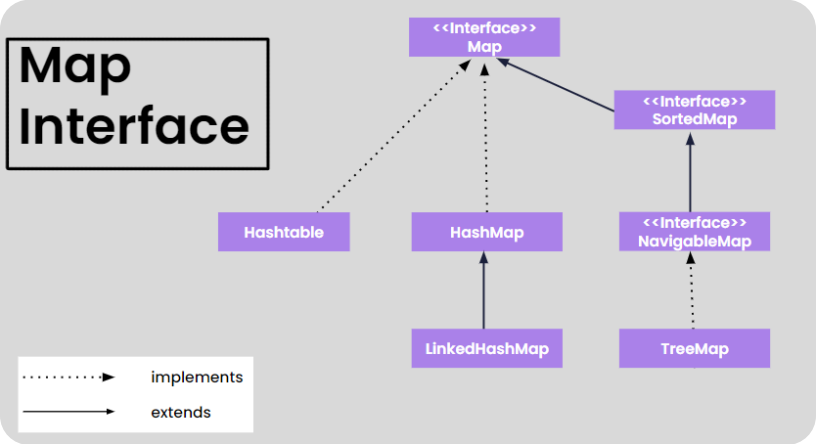
views of a Map

j.Set keySet()
k.Collection values()
l.Set entrySet()

Entry(I)

1. Each key-value pair is called Entry.
2. Without the existence of a Map,there can't be the existence of an Entry Object.
3. Interface entry is defined inside the Map interface.

```
interface Map{
 interface Entry{
  Object getKey();
  Object getValue();
  Object setValue(Object newValue);
 }
}
```



HashMap

- Underlying DataStructure : Hashtable
- insertion order : not preserved
- duplicate keys : not allowed
- duplicate values : allowed
- Heterogenous objects : allowed
- null insertion : for keys allowed only once,but for values can be any no.
- implementation interface : Serializable,Cloneable.

Difference b/w HashMap and Hashtable

HashMap => All the methods are not synchronized.
Hashtable => All the methods are synchronised.

HashMap => At a time multiple threads can operate on an Object, so it is ThreadSafe.
Hashtable => At a time only one Thread can operate on an Object, so it is not ThreadSafe.

HashMap => Performance is high.
Hashtable => performance is low.

HashMap => null is allowed for both keys and values.
Hashtable => null is not allowed for both keys and values,it would result in NullPointerException.

HashMap => Introduced in 1.2v
Hashtable => Introduced in 1.0v

Note: By default HashMap is nonSynchronized,to get the synchronized version of HashMap we need to use synchronizedMap() of Collection class.

Constructors

1. HashMap hm=new HashMap()
   //default capacity => 16, loadfactor => 0.75
2. Hashmap hm=new HashMap(int capacity);
3. HashMap hm=new HashMap(int capacity,flot filtration);
4. HashMap hm=new HashMap(Map m);

LinkedHashMap

=> It is the child class of HashMap.
=> It is same as HashMap,but with the following difference

HashMap => underlying data structure is hashtable.
LinkedHashMap => underlying data structure is LinkedList + hashtable.

HashMap => insertion order not preserved.
LinkedHashMap => insertion order preserved.

HashMap => introduced in 1.2v
LinkedHashMap => introduced in 1.4v

SortedMap

1. It is the child interface of Map
2. If we want an Entry object to be sorted and stored inside the map,we need to use "SortedMap".

SortedMap defines few specific method like
a. Object firstKey()
b. Object lastKey()
c. SortedMap headMap(Object key)
d. SortedMap tailMap(Object key)
e. SortedMap subMap(Object obj1,Object obj2)
f. Comparator comparator()

NavigableMap (I):

=> It is the Child Interface of SortedMap.
=> It Defines Several Methods for Navigation Purposes.

TreeMap

- Underlying data structure is "redblacktree".
- Duplicate keys are not allowed,whereas values are allowed.
- Insertion order is not preserved and it is based on some sorting order.

- If we are depending on natural sorting order,then those keys should be homogenous and it should be Comparable otherwise ClassCastException.
- If we are working on customisation through Comparator,then those keys can be heterogeneous and it can be NonComparable.
- No restrictions on values, it can be heterogeneous or NonComparable also.
- If we try to add null Entry into TreeMap, it would result in "NullPointerException".

Hashtable:

- The Underlying Data Structure for Hashtable is Hashtable Only.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
- Heterogeneous Objects are Allowed for Both Keys and Values.
- null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.
- It implements Serializable and Cloneable,but not RandomAccess.
- Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe,so best suited when we work with Search Operation.

Need of Generics and Basics of Generics Generics

The purpose of Generics is

1. To provide TypeSafety.
2. To resolve TypeCasting problems.

Case1:

TypeSafety

- A guarantee can be provided based on the type of elements.
- If our programming requirement is to hold only String type of Objects,we can choose String Array.
- By mistake if we are trying to add any another type of Objects, we will get "CompileTimeError".

eg#1.

```
String[] s=new String[10000];
     s[0] = "dhoni";
     s[1] = "sachin";
     s[2] = new Integer(10); //CE:
incompatible types found :
java.lang.Integer

                  required:
java.lang.String
```

W.r.t Arrays we can guarantee that what type of elements is present inside Array, hence Arrays are safe to use w.r.t type,so Arrays as TypeSafety.

eg#2.

```
ArrayList l=new ArrayList();
     l.add("dhoni");
     l.add("sachin");
     l.add(new Integer(10));
        ...
        ....
     String s1=(String)l.get(0);
     String s2=(String)l.get(1);
     String s3=(String)l.get(2); //
RE: ClassCastException
```

For Collections, we can't guarantee the type of elements present inside Collection.
If our program requirement is to hold only String type of Objects then if we choose ArrayList by mistake if we are trying to add any other type of Object,we won't get any CompileTimeError,but the program may fail at runtime.

Note: Arrays are TypeSafe,whereas Collections are not TypeSafe.
Arrays provide guarantee for the type of elements we hold, whereas Collections won't provide guarantee for the type of elements.

Need of Generics

1. Arrays to use we need to know the size from the beginning,but if we don't the size and still If we want to provide type casting we need to use "Collections along with Generics".

Case2:
Type casting

eg#1

```
String s[]=new String[3];
   s[0] = "sachin";
 String name=s[0];//
```

Typecasting not required as we it holds only String elements only.

eg#2

```
ArrayList l=new ArrayList();
   l.add("sachin");
 String name=l.get(0);//
```

CE: found : java.lang.Object required: java.lang.String

```
String name=(String)l.get(0);
        |=>TypeCasting is compulsory when we work with Collections.
```

To Overcome the above mentioned problems of Collections we need to go for Generics in 1.5V, which provides TypeSafety and to Resolve TypeCasting problems**.**

How TypeSafety is provided in Generics and how it resolves the problems of TypeCasting?

eg#1
ArrayList

```
al=new ArrayList();//
```

Non-Generic ArrayList which holds any type of elements.

```
al.add("sachin");
 al.add("dhoni");
 al.add(new Integer(10));
            al.add("yuvi");
```

eg#2.

ArrayList

```
<String> al=new ArrayList<String>();//
```

Generic ArrayList which holds only String.

```
al.add("sachin");
 al.add("dhoni");
 al.add(new Integer(10));//CE
            al.add("yuvi");
```

Note: Through Generics TypeSafety is provided.

ArrayList

```
<String> al=new ArrayList<String>();//
```

Generic ArrayList which holds only String.

```
al.add("sachin");
 al.add("dhoni");
            al.add("yuvi");
 String name=al.get(0); //
```

TypeCasting is not required

At the time of retrieval, we are not required to perform TypeCasting.
Note: Through Generics TypeCasting problem is solved.

Difference b/w
ArrayList l=new ArrayList();

- It is a non generic version of ArrayList Object.
- It won't provide TypeSafety as we can add any elements into the ArrayList.
- TypeCasting is required when we retrieve elements.

ArrayList<String> l=new ArrayList<String>();

- It is a generic version of ArrayList Object.
- It provides TypeSafety as when we can add only String type of Objects.
- TypeCasting is not required when we retrieve elements.

Conclusion - 1

```
⟹ parameter type
      ArrayList<String> al =new ArrayList<String>();
          ⟹ BaseType

      List<String> al =new ArrayList<String>();
      Collection<String> al =new ArrayList<String>();

      ArrayList<Object> al=new ArrayList<String>();//CE:incompatible type:
        found ArrayList<String>
        required ArrayList<Object>
```

Polymorphism is applicable only for the BaseType,but not for the Parameter type.
Polymorphism=> usage of parent reference to hold Child object is the concept of "Polymorphism".

Conclusion - 2

Collection concept is applicable only for Object,it is not applicable for primitive types.
So parameter type should be always be class/interface/enum,if we take primitive it would raise in "CompileTimeError".

eg#1
ArrayList

```
<int> al=new ArrayList<int>();//
```

CE: unexpected type: found int required reference

Generic classes

until 1.4 version, nongeneric version of ArrayList class is declared as follows

```
class ArrayList{
 boolean add(Object o); //Argument is Object so no typesafety.
 Object get(int index); //return type is Object so type casting is required.
}
```

In 1.5 Version Generic version class is defined as follows

```
class ArrayList{
 boolean add(Object o); //Argument is Object so no typesafety.
 Object get(int index); //return type is Object so type casting is required.
}
```

In 1.5 Version Generic version class is defined as follows

```
                ⟹ TypeParameter
class ArrayList<T>{
 boolean add(T t);
 T get(int index);
}
```

T => Based on our runtime requirement, T will be replaced with our provided type.

class ArrayList

```
<String>{
 boolean add(String t); //We can add only String type of Object it provides TypeSafety
 String get(int index); //Retrieval Object is always of type String, so TypeCasting not  required.
   }
```

To hold only String type of Object

ArrayList

```
<String> al =new ArrayList<String>
 al.add("sachin");
 al.add(new Integer(10)); //CE: can't find symbol method: add(java.lang.Integer)
```

location: class ArrayList<String>
String name=al.get(0);
System.out.println(name);

Note:
In Generics we are associating a type-parameter to the class, such type of parameterised classes are nothing but Generic classes.
Generic class : class with type-parameter.

Collection(I)

Inside this interface, the commonly used method required for all the collection classes is present
a. boolean add(object o)=> Only one object
b. boolean addAll(Collection c)=>To add group of Object
c. boolean remove(Object o) => to remove particular object
d. boolean removeAll(Collection c)=> to remove particular group of collection
e. void clear() => to remove all the object
f. int size() => to check the size of the array
g. boolean retainAll(Collection c) => except this group of objects remaining all objects should be removed.
h. boolean contains(Object o) => to check whether a particular object exists or not
i. boolean containsAll(Collection c) => To check whether a particular Collection exists or not
j. boolean isEmpty() => To check whether the Collection is empty or not
k. Object[] toArray()=> Convert the object into Array.
l. Iterator iterator() => cursor need to iterate the collection object

Note :There is no concrete class which implements Collection interface directly.

Comparator vs Comparable Interface

Comparator

1. It is an interface present in java.util package
2. It contains 2 abstract method
   public abstract int compare(Object obj1,Object obj2)
   public abstract boolean equals(Object o)
3. int compare(Object obj1,Object obj2)
   |=> return -ve iff obj1 has to come before obj2
   |=> return +ve iff obj1 has to come after obj2
   |=> return 0 both are equal
4. Whenever we are implementing an Comparator interface compulsorily we should give body for compare().
5. Whereas for equals(),we get the body from Object class through inheritance.

eg:

```
class MyComparator implements Comparator{
public int compare(Object obj1,Object obj2){
 ....
 ....
    }
}
```

Comparable(I)

=> It is a part of java.lang package
=> It contains only one method compareTo.
public int compareTo(Object o)

=> obj1.compareTo(obj2)
returns -ve iff obj1 has to come before obj2
returns +ve iff obj1 has to come after obj2
returns 0 if both are equal

eg#1.

```
System.out.println("A".compareTo("Z"));//A should  come before Z so -ve
System.out.println("Z".compareTo("K"));//Z should  come after  K so +ve
System.out.println("A".compareTo("A"));//Both are equal zero
System.out.println("A".compareTo(null));//NullPointerException
```

Comparable
=> compareTo()
It is meant for the default natural sorting order.

Comparator
=> compare()
It is meant for customized sorting order.

Scenario

When to go for Comparable and Comparator?

1st category

Predefined Comparable classes like String and Wrapper class
=> Default natural sorting order is already available
=> If not satisfied, then we need to go for Comparator

2nd Category

Predefined NonComparable classes like StringBuffer
=> Default natural sorting order not available so go for Comparator only aways

3rd Category

Our Own classes like Employee,Student,Customer
=>Person who is writing this classes are responsible for implementing comparable interface to promote Natural sorting order.
=>Person who is using this class,can define his own natural sorting order by implementing Comparator interface.

Comparable and Comparator

Comparable => Meant for default natural sorting order
Comparator => Meant for customized sorting order

Comparable => part of java.lang package
Comparator => part of java.util package

Comparable => only one method compareTo()
Comparator => 2 methods compare(),equals()

Comparable => It is implemented by Wrapper class and String class
Comparator => It is implemented by Collator and RuleBaseCollator(GUI based API)