# Structuring Data with Java

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

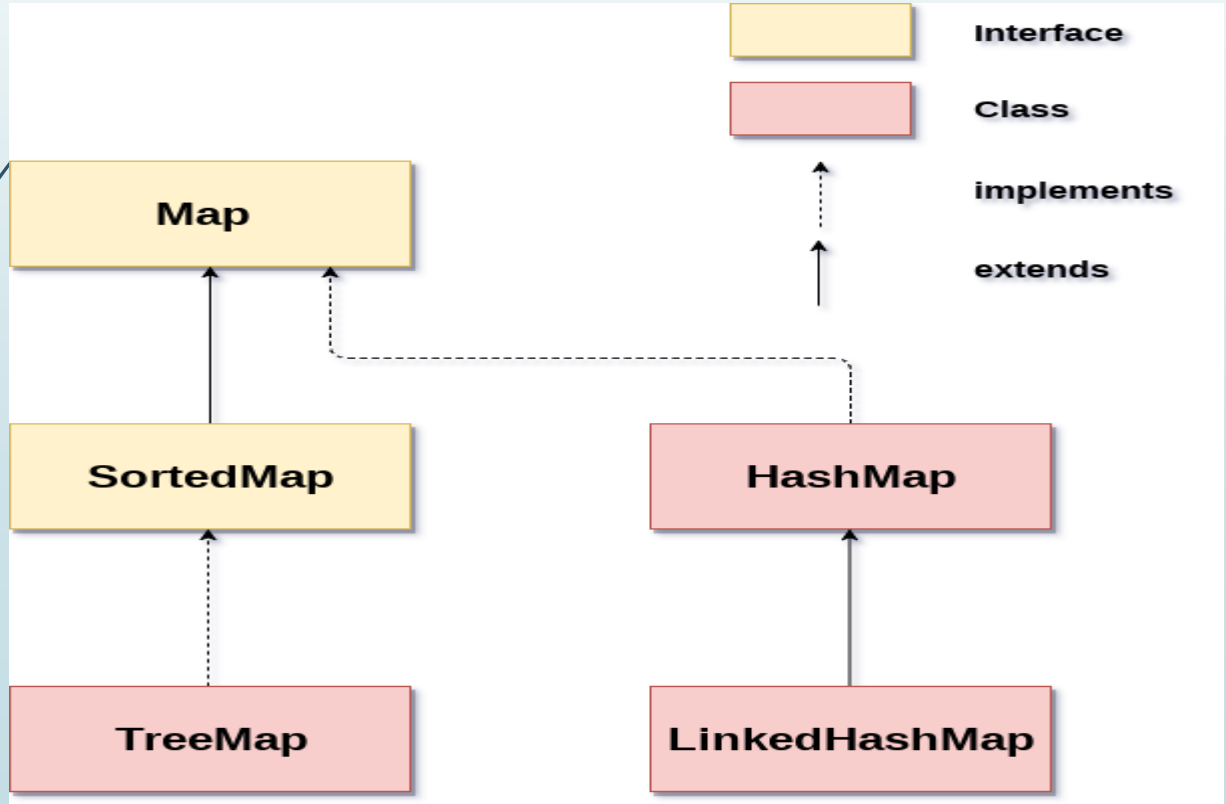
# Structuring Data with Java

## (Mapping with Hashtable and HashMap)

***Java 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. A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.

**Java Map Hierarchy**




- There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.
- A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow **null keys** and values, but TreeMap **doesn't allow any null key** or value.
- A Map can't be traversed, so you need to **convert it into Set** using *keySet()* or *entrySet()* method.

# Structuring Data with Java
## (Mapping with Hashtable and HashMap)

***Java HashMap class***

Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

- Points to remember

- Java HashMap class contains values based on the key.
- Java HashMap class contains only unique keys.
- Java HashMap class may have one null key and multiple null values.
- Java HashMap class is non synchronized.
- Java HashMap class maintains no order.

**Constructor**

- `HashMap()`

It is used to construct a default HashMap.

# Structuring Data with Java
## (Mapping with Hashtable and HashMap)

**Methods**

**`put()`**

- `Syntax: V put(Object key, Object value)`

It is used to insert an entry in the map.

**`containsValue()`**

- `Syntax: boolean containsValue(Object value)`

This method returns true if some value equal to the value exists within the map, else return false.

**`containsKey()`**

- `Syntax: boolean containsKey(Object key)`

This method returns true if some key equal to the key exists within the map, else return false.

**`isEmpty()`**

- `Syntax: boolean isEmpty()`

This method returns true if the map is empty; returns false if it contains at least one key.

**`replace()`**

- `Syntax: V replace(K key, V value)`

It replaces the specified value for a specified key.

- `Syntax:boolean replace(K key, V oldValue, V newValue)`

It replaces the old value with the new value for a specified key.

# Structuring Data with Java
## (Mapping with Hashtable and HashMap)

Example:

```
public class HashMapTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

    Map<Integer, String> map=new HashMap<Integer, String>();

    map.put(10, "subrat");

    map.put(20, "kumar");

    map.put(30, "nayak");

    System.out.println(map);

    if(map.containsKey(30))

    {

        String v=map.get(30);//get() method returns the object that contains the value
        associated with the key.

        System.out.println("Value of key 30 is: "+v);

    }

    map.clear();

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

# Structuring Data with Java
## (Mapping with Hashtable and HashMap)

***Java Hashtable class***

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

- Points to remember
  - A Hashtable contains values based on the key.
  - Java Hashtable class contains unique elements.
  - Java Hashtable class doesn't allow null key or value.
  - Java Hashtable class is synchronized.

**Constructor**

- `Hashtable()`

It creates an empty hashtable having the initial default capacity and load factor.

# Structuring Data with Java

## (Mapping with Hashtable and HashMap)

**HashMap vs HashTable**

| HashMap | HashTable |
| --- | --- |
| • HashMap is non synchronized. | • Hashtable is synchronized. |
| • HashMap allows one null key and multiple null values. | • Hashtable doesn't allow any null key or value. |
| • HashMap is fast. | • Hashtable is slow. |
| • We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | • Hashtable is internally synchronized and can't be unsynchronized. |
| • HashMap is traversed by Iterator. | • Hashtable is traversed by Enumerator and Iterator. |
| • HashMap inherits AbstractMap class. | • Hashtable inherits Dictionary class. |

# Structuring Data with Java
## (Sorting a Collection)

**Problem**

- You put your data into a collection in random order that doesn't preserve the order, and now you want it sorted.

**Solution**

- Use the static method Arrays.sort() or Collections.sort(), optionally providing a Comparator.

***Arrays.sort()***

```
public class ArraySortTest {
public static void main(String[] args)
{
Double[] arr= {5.5,1.5,3.5,2.5};
Arrays.sort(arr);
for(Double x:arr)
System.out.println(x);
}}
```

***Collection.sort()***

```
import java.util.Collections;
public class CollectionsSortTest {
public static void main(String[] args) {
List<String> ts=new ArrayList<String>();
ts.add("Subrat");
ts.add("Kumar");
ts.add("Nayak");
Collections.sort(ts);
System.out.println(ts);
}}
```

# Structuring Data with Java
## (Sorting a Collection)

**Problem**

- Your data needs to be sorted, but you don't want to stop and sort it periodically.

**Solution**

- Not everything that requires order requires an explicit *sort* operation. Just keep the data sorted at all times.

**Discussion**

- You can avoid the overhead and elapsed time of an explicit sorting operation by ensuring that the data is in the correct order at all times, though this may or may not be faster overall, depending on your data and how you choose to keep it sorted. You can keep it sorted either manually or by using a **TreeSet or a TreeMap**.
- One point to note is that if you have a Hashtable or HashMap, you can convert it to a TreeMap, and therefore get it sorted, just by passing it to the TreeMap constructor:

```
TreeMap sorted = new TreeMap(unsortedHashMap);
```

# Structuring Data with Java

## (Mapping with Hashtable and HashMap)

```
TreeSet<String> theSet = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
theSet.add("Gosling");
theSet.add("da Vinci");
theSet.add("van Gogh");
theSet.add("Java To Go");
theSet.add("Vanguard");
theSet.add("Darwin");
theSet.add("Darwin"); // TreeSet is Set, ignores duplicates.
System.out.printf("Our set contains %d elements", theSet.size());
// Since it is sorted we can easily get various subsets
System.out.println("Lowest (alphabetically) is " + theSet.first());
// Print how many elements are greater than "k"
// Should be 2 - "van Gogh" and "Vanguard"
System.out.println(theSet.tailSet("k").toArray().length +
" elements higher than \"k\"");
// Print the whole list in sorted order
System.out.println("Sorted list:");
theSet.forEach(name -> System.out.println(name));
```

# Structuring Data with Java
## (Finding an Object in a Collection)

***Problem***

- You need to see whether a given collection contains a particular value.

***Solution***

- Ask the collection if it contains an object of the given value.

***Discussion***

- if the collection is prepared by another part of a large application, or even if you've just been putting objects into it and now need to find out if a given value was found, this recipe's for you. There is quite a variety of methods, depending on which collection class you have.

*Table 7-4. Finding objects in a collection*

| Method(s) | Meaning | Implementing classes |
|---|---|---|
| `binarySearch()` | Fairly fast search | `Arrays, Collections` |
| `contains( )` | Search | `ArrayList, HashSet, Hashtable, LinkList, Properties, Vector` |
| `containsKey( ), containsValue( )` | Checks if the collection contains the object as a `Key` or as a `Value` | `HashMap, Hashtable, Properties, TreeMap` |
| `indexOf( )` | Returns location where object is found | `ArrayList, LinkedList, List, Stack, Vector` |
| `search( )` | Search | `Stack` |

# Structuring Data with Java
## (Converting a Collection to an Array)

**Problem**

- You have a Collection but you need a Java language array.

**Solution**

- Use the Collection method toArray().

Example:

```
public static void main(String[] args) {
// TODO Auto-generated method stub
List<String> list = new ArrayList<>();
list.add("Blobbo");
list.add("Cracked");
list.add("Dumbo");
// Convert a collection to Object[], which can store objects // of any type.
Object[] ol = list.toArray();
System.out.println("Array of Object has length " + ol.length);
String[] sl = (String[]) list.toArray(new String[0]);
System.out.println("Array of String has length " + sl.length);
}
```

# Structuring Data with Java
## (Stack)

**Problem**

- You need to process data in "last-in, first-out" (LIFO) or "most recently added" order.

**Solution**

- Write your own code for creating a stack; it's easy. Or, use a java.util.Stack.

**Discussion**

- This is a common data structuring operation and is often used to reverse the order of objects. The basic operations of any stack are push() (add to stack), pop() (remove from stack), and peek() (examine top element without removing).

```
public class ArrayListTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

Stack<Integer> st=new Stack<Integer>();

st.push(4);

st.push(5);

st.push(6);

System.out.println(st.peek());
System.out.println(st);
while(!st.isEmpty())
System.out.println(st.pop());
System.out.println(st);
}}
```

# Structuring Data with Java
## (Multidimensional Structures)

Assignment

# End of Session