Estd. 2000

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 4

Lecture 33

Lecture 33

- Map Interface
- HashMap Class
- LinkedHashMap Class

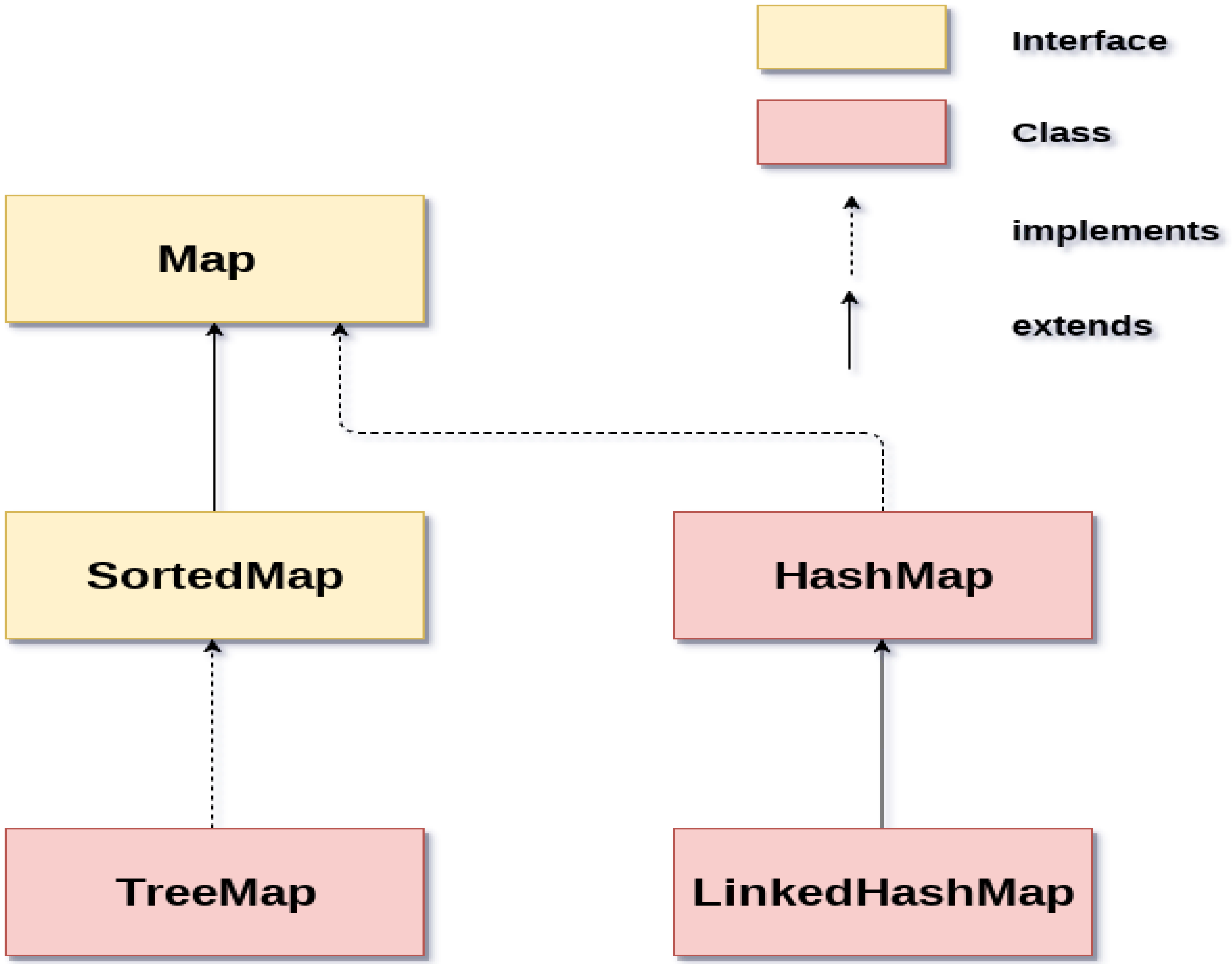
Map Interface

The Map interface in Java is part of the java.util package and represents a collection of key-value pairs. It provides a way to store and retrieve values using keys, similar to a dictionary or an associative array in other programming languages.

Key-Value Pairs: A Map stores data as key-value pairs, where each key is unique and maps to a specific value. Keys are used to access and retrieve the associated values.

Unique Keys: The keys in a Map must be unique. Attempting to insert a duplicate key will replace the value associated with the existing key.

Null Values: A Map can store null values, but the behavior for null keys depends on the specific implementation.


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Map Interface

Implementations: The Map interface has several concrete implementations in Java, including HashMap, TreeMap, LinkedHashMap, and ConcurrentHashMap.

Ordering: Some Map implementations, like TreeMap, maintain an ordering of the keys, while others, like HashMap, do not guarantee any specific order.

Common Methods: The Map interface provides various methods for working with key-value pairs, such as put(), get(), containsKey(), containsValue(), remove(), size(), keySet(), values(), and entrySet().

```
import java.util.HashMap;
 import java.util.Map;
public class MapExample {
 public static void main(String[] args) {
// Creating a HashMap
Map<String, Integer> studentAges = new HashMap<>();
// Adding key-value pairs to the Map
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
 studentAges.put("Charlie", 19);
// Retrieving a value using its key
 int aliceAge = studentAges.get("Alice"); System.out.println("Alice's
age: " + aliceAge); // Output: Alice's age: 20
```

Example

```
// Checking if a key exists
boolean containsBob = studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " + containsBob);
 // Output: Contains 'Bob'? true
// Iterating over the key-value pairs
System.out.println("Student Ages:");
for (Map.Entry<String, Integer> entry : studentAges.entrySet()) {
String name = entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
 // Output: // Student Ages: // Alice: 20 // Bob: 22 // Charlie: 19
 }
 }
```

- In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the Map, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.
- Maps are widely used in scenarios where you need to associate keys with values and efficiently retrieve or manipulate those values based on their keys. Examples include caching systems, dictionaries, symbol tables, and various data structures that require efficient key-based lookups.
- Different Map implementations provide different performance characteristics and features. For example, HashMap offers constant-time performance for most operations but does not maintain any order of the key-value pairs. In contrast, TreeMap maintains the key-value pairs in sorted order based on the keys, but with a slightly higher overhead for certain operations.

Example

```
import java.util.HashMap;

 import java.util.Map;

public class MapExample {

 public static void main(String[] args) {

// Creating a HashMap

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

// Adding key-value pairs to the Map
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
studentAges.put("Charlie", 19);

// Retrieving a value using its key

 int aliceAge = studentAges.get("Alice");
System.out.println("Alice's age: " +
aliceAge); // Output: Alice's age: 20
```

```
// Checking if a key exists
boolean containsBob =
studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " + containsBob);
 // Output: Contains 'Bob'? true
// Iterating over the key-value pairs
System.out.println("Student Ages:");
for (Map.Entry<String, Integer> entry :
studentAges.entrySet()) { String name =
entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
 // Output: // Student Ages: // Alice: 20 // Bob: 22 //
Charlie: 19 }
 }
```

- In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the Map, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.
- Maps are widely used in scenarios where you need to associate keys with values and efficiently retrieve or manipulate those values based on their keys. Examples include caching systems, dictionaries, symbol tables, and various data structures that require efficient key-based lookups.
- Different Map implementations provide different performance characteristics and features. For example, HashMap offers constant-time performance for most operations but does not maintain any order of the key-value pairs. In contrast, TreeMap maintains the key-value pairs in sorted order based on the keys, but with a slightly higher overhead for certain operations.

HashMap Class

HashMap is one of the most commonly used implementations of the Map interface in Java. It stores key-value pairs in a hash table data structure, providing constant-time performance for basic operations like get(), put(), and remove() on average.

Hash Table Implementation: HashMap internally uses a hash table to store its key-value pairs. The hash table is an array of buckets, where each bucket can hold one or more key-value pairs.

Hashing: To store and retrieve values efficiently, HashMap uses a hashing function to map keys to specific indices (buckets) in the underlying array. This hashing function is implemented by the hashCode() method of the key objects.

Collision Handling: Since hashing can cause collisions (two or more keys mapping to the same bucket), HashMap handles collisions using separate chaining. Each bucket contains a linked list of key-value pairs that hash to that bucket.

HashMap Class

Null Keys and Values: HashMap allows one null key and any number of null values.

No Ordering: HashMap does not maintain any specific order of the key-value pairs. The order in which elements are returned by the iterator is not guaranteed to be the same as the insertion order.

Performance: Basic operations like get(), put(), and remove() have constant-time performance on average, assuming a good hash function and a properly sized hash table.

However, in the worst case (e.g., when all keys hash to the same bucket), the performance can degrade to linear time.Iterators: HashMap provides iterators to traverse the keys, values, or key-value pairs through the keySet(), values(), and entrySet() methods, respectively. The order of iteration is not guaranteed.

ABES Engineering College
College Code 032

Example

```
import java.util.HashMap;
 import java.util.Map;
public class HashMapExample {
 public static void main(String[] args) {
 // Creating a HashMap
Map<String, Integer> studentAges = new
HashMap<>();
// Adding key-value pairs to the HashMap
studentAges.put("Alice", 20);
studentAges.put("Bob", 22);
studentAges.put("Charlie", 19);
 // Retrieving a value using its key
int aliceAge = studentAges.get("Alice");
System.out.println("Alice's age: " + aliceAge);
// Output: Alice's age: 20
```

```
// Checking if a key exists
boolean containsBob =
studentAges.containsKey("Bob");
System.out.println("Contains 'Bob'? " +
containsBob);
 // Output: Contains 'Bob'? true // Iterating over
the key-value pairs
System.out.println("Student Ages:");
 for (Map.Entry<String, Integer> entry :
studentAges.entrySet()) {
String name = entry.getKey();
int age = entry.getValue();
 System.out.println(name + ": " + age); }
// Output (order may vary): // Student Ages: //
Alice: 20 // Bob: 22 // Charlie: 19
}
```

In this example, we create a HashMap to store student names and their ages. We demonstrate adding key-value pairs to the HashMap, retrieving a value using its key, checking if a key exists, and iterating over the key-value pairs using the entrySet() method.

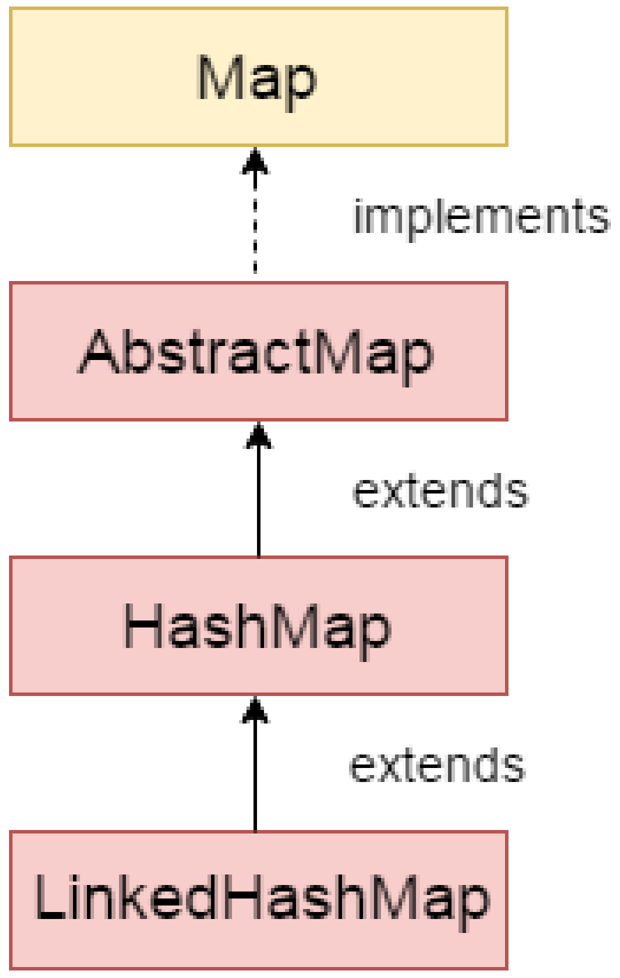
Note that the order of iteration is not guaranteed.

HashMap is widely used when you need to store and retrieve key-value pairs efficiently, and you don't need to maintain any specific order of the elements. It provides constant-time performance for basic operations on average, making it suitable for large datasets.

However, it's important to note that the performance of HashMap depends on the quality of the hash function and the load factor (the ratio of elements to the capacity of the hash table). If the hash function is not well-distributed or the load factor becomes too high, the performance can degrade due to an increase in collisions and the need for resizing the hash table.

LinkedHashMap Class

The LinkedHashMap class in Java is a subclass of the HashMap class and provides the additional functionality of maintaining the insertion order of the elements. It inherits the properties of the HashMap class and additionally maintains a doubly-linked list of the entries, allowing for insertion-order iteration.



- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

LinkedHashMap Class Features

Underlying Data Structure: LinkedHashMap uses a combination of a HashMap and a doubly-linked list to store the key-value pairs. The HashMap provides efficient key-value storage and retrieval, while the doubly-linked list maintains the insertion order of the elements.

Insertion Order: When elements are inserted into a LinkedHashMap, they are added to both the HashMap and the doubly-linked list. The order in which the elements are added to the LinkedHashMap is preserved in the doubly-linked list.

LinkedHashMap Class Features

Access Order: By default, LinkedHashMap maintains the insertion order of the elements. However, it can be configured to maintain the access order (the order in which the elements are accessed) by setting the accessOrder flag to true during the construction of the LinkedHashMap.

Performance: LinkedHashMap has the same time complexity as HashMap for basic operations like get(), put(), and remove(), which is O(1) on average. However, iterating over the elements in the LinkedHashMap takes O(n) time, where n is the number of elements.

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){
  LinkedHashMap<Integer,String> hm=new
LinkedHashMap<Integer,String>();
   hm.put(100,"Amit");
   hm.put(101,"Vijay");
   hm.put(102,"Rahul");
   for(Map.Entry m:hm.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
  }
}
```

Output:

```
100 Amit
101 Vijay
102 Rahul
```

ABES Engineering College
College Code 032
Estd.2000

LinkedHashMap Example: Key-Value pair

```
import java.util.*;
class LinkedHashMap2{
 public static void main(String args[]){
   LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>()
    map.put(100,"Amit");
     map.put(101,"Vijay");
     map.put(102,"Rahul");
       //Fetching key
       System.out.println("Keys: "+map.keySet());
       //Fetching value
       System.out.println("Values: "+map.values());
       //Fetching key-value pair
       System.out.println("Key-Value pairs: "+map.entrySet());
 }
}
```

```
Keys: [100, 101, 102]
Values: [Amit, Vijay, Rahul]
Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]
```

LinkedHashMap Example:remove()

```
import java.util.*;
public class LinkedHashMap3 {
  public static void main(String args[]) {
   Map<Integer,String> map=new LinkedHashMap<Integer,String>();
    map.put(101,"Amit");
    map.put(102,"Vijay");
    map.put(103,"Rahul");
    System.out.println("Before invoking remove() method: "+map);
   map.remove(102);
   System.out.println("After invoking remove() method: "+map);
  }
}
```

```
Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}
After invoking remove() method: {101=Amit, 103=Rahul}
```