HASH TABLE

Hashtable class, introduced as part of the Java Collections framework, implements a hash table that maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method. The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface.

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

```
import java.util.Hashtable;

public class GFG

{

   public static void main(String args[])

   {


       // Create a Hashtable of String

       // keys and Integer values

      Hashtable<String, Integer> ht = new Hashtable<>();


      // Adding elements to the Hashtable

      ht.put("One ", 1);

      ht.put("Two ", 2);
```

```
        ht.put("Three ", 3);


        // Displaying the Hashtable elements
        System.out.println("Hashtable Elements: " + ht);
    }
}
```

HashTable Constructors

In order to create a Hashtable, we need to import it from java.util.Hashtable. There are various ways in which we can create a Hashtable.

1. Hashtable()

This creates an empty hashtable with the default load factor of 0.75 and an initial capacity is 11.

```
Hashtable<K, V> ht = new Hashtable<K, V>();
```

Implementation: Using Hashtable() Constructor

```
import java.io.*;
import java.util.*;


class AddElementsToHashtable
{
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
        Hashtable<Integer, String> ht1 = new Hashtable<>();
```

```
        // Initialization of a Hashtable
        // using Generics
        Hashtable<Integer, String> ht2
            = new Hashtable<Integer, String>();

        // Inserting the Elements
        // using put() method
        ht1.put(1, "one");
        ht1.put(2, "two");
        ht1.put(3, "three");

        ht2.put(4, "four");
        ht2.put(5, "five");
        ht2.put(6, "six");

        // Print mappings to the console
        System.out.println("Mappings of ht1 : " + ht1);
        System.out.println("Mappings of ht2 : " + ht2);
    }
}
```

Output

```
Mappings of ht1 : {3=three, 2=two, 1=one}
Mappings of ht2 : {6=six, 5=five, 4=four}
```

2. Hashtable(int initialCapacity)

This creates a hash table that has an initial size specified by initialCapacity and the default load factor is 0.75.

Hashtable<K, V> ht = new Hashtable<K, V>(int initialCapacity);

Implementation: Using Hashtable(int initialCapacity)

```
import java.io.*;
import java.util.*;

class GFG
{
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
        Hashtable<Integer, String> ht1
          = new Hashtable<>(4);

        // Initialization of a Hashtable
        // using Generics
        Hashtable<Integer, String> ht2
          = new Hashtable<Integer, String>(2);

        // Inserting the Elements
        // using put() method
```

```
        ht1.put(1, "one");
        ht1.put(2, "two");
        ht1.put(3, "three");

        ht2.put(4, "four");
        ht2.put(5, "five");
        ht2.put(6, "six");

        // Print mappings to the console
        System.out.println("Mappings of ht1 : " + ht1);
        System.out.println("Mappings of ht2 : " + ht2);
    }
}
```

Output

```
Mappings of ht1 : {3=three, 2=two, 1=one}
Mappings of ht2 : {4=four, 6=six, 5=five}
```

3. Hashtable(int size, float fillRatio)

This version creates a hash table that has an initial size specified by size and fill ratio specified by fillRatio. fill ratio: Basically, it determines how full a hash table can be before it is resized upward and its Value lies between 0.0 to 1.0.

```
Hashtable<K, V> ht = new Hashtable<K, V>(int size, float fillRatio);
```

Implemention: Using Hashtable(int size, float fillRatio)

```
import java.io.*;
import java.util.*;

class GFG
{
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
        Hashtable<Integer, String> ht1
            = new Hashtable<>(4, 0.75f);

        // Initialization of a Hashtable
        // using Generics
        Hashtable<Integer, String> ht2
            = new Hashtable<Integer, String>(3, 0.5f);

        // Inserting the Elements
        // using put() method
        ht1.put(1, "one");
        ht1.put(2, "two");
        ht1.put(3, "three");

        ht2.put(4, "four");
        ht2.put(5, "five");
```

```
        ht2.put(6, "six");

        // Print mappings to the console
        System.out.println("Mappings of ht1 : " + ht1);
        System.out.println("Mappings of ht2 : " + ht2);
    }
}
```

Output

```
Mappings of ht1 : {3=three, 2=two, 1=one}
Mappings of ht2 : {6=six, 5=five, 4=four}
```

4. Hashtable(Map<? extends K,? extends V> m)

This creates a hash table that is initialized with the elements in m.

```
Hashtable<K, V> ht = new Hashtable<K, V>(Map m);
```

Implementation: Using Hashtable(Map<? extends K,? extends V> m)

```
import java.io.*;
import java.util.*;

class GFG
{
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
```

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

        // Inserting the Elements
        hm.put(1, "one");
        hm.put(2, "two");
        hm.put(3, "three");

        // Initialization of a Hashtable
        // using Generics
        Hashtable<Integer, String> ht2
            = new Hashtable<Integer, String>(hm);

        // Print mappings to the console

        System.out.println("Mappings of ht2 : " + ht2);
    }
}
```

Output

Mappings of ht2 : {3=three, 2=two, 1=one}

Example:

```
// To illustrate Java.util.Hashtable
import java.util.*;

public class GFG {
    public static void main(String[] args)
    {
        // Create an empty Hashtable
        Hashtable<String, Integer> ht = new Hashtable<>();

        // Add elements to the hashtable
        ht.put("vishal", 10);
        ht.put("sachin", 30);
        ht.put("vaibhav", 20);

        // Print size and content
        System.out.println("Size of map is: " + ht.size());
        System.out.println(ht);

        // Check if a key is present and if
        // present, print value
        if (ht.containsKey("vishal")) {
            Integer a = ht.get("vishal");
            System.out.println("value for key"
                               + " \"vishal\" is: " + a);
        }
```

```
    }
}
```

Output

```
Size of map is: 3
{vaibhav=20, vishal=10, sachin=30}
value for key "vishal" is: 10
```