Importance of implementing hashcode() method for Java

I thought of sharing an important concept of implementing the method hashcode(). Some of us might know the concept, but based on some debugging and testing, I feel there is still some ambiguity/confusion in understanding the finer points around it.

I learned this concept from the Effective Java book, which explains the importance of this concept. Joshua mentions that the user needs to run some tests to verify the above fact. The following blog does exactly that.

The example illustrates the primary concept that even if two Java objects are semantically equal because of the "equals" implementation, they can still generate two different keys.

Brief explanation of the example:

Two equal phone number objects are created , and they represent the same phone number.
The first phone object is used as a key and a value is inserted into the map.
During the get operation, the second same phone number object is created and used as the key, the object returned is null.
Why is this?

The answer lies in the internal implementation of the HashMap class and the way it generates indices from keys and puts the value in the internal table. [Basically an internal associative array]
.

I have attached a word document that explains these details with debug screenshots.

The bottom line is that if you are making use of Java objects as keys into a HashMap, please ensure that it implements the hashcode() method.

Now, what is the best hashcode() implementation? For that please read the details from the Effective Java book on Chapter number 3, item 9.

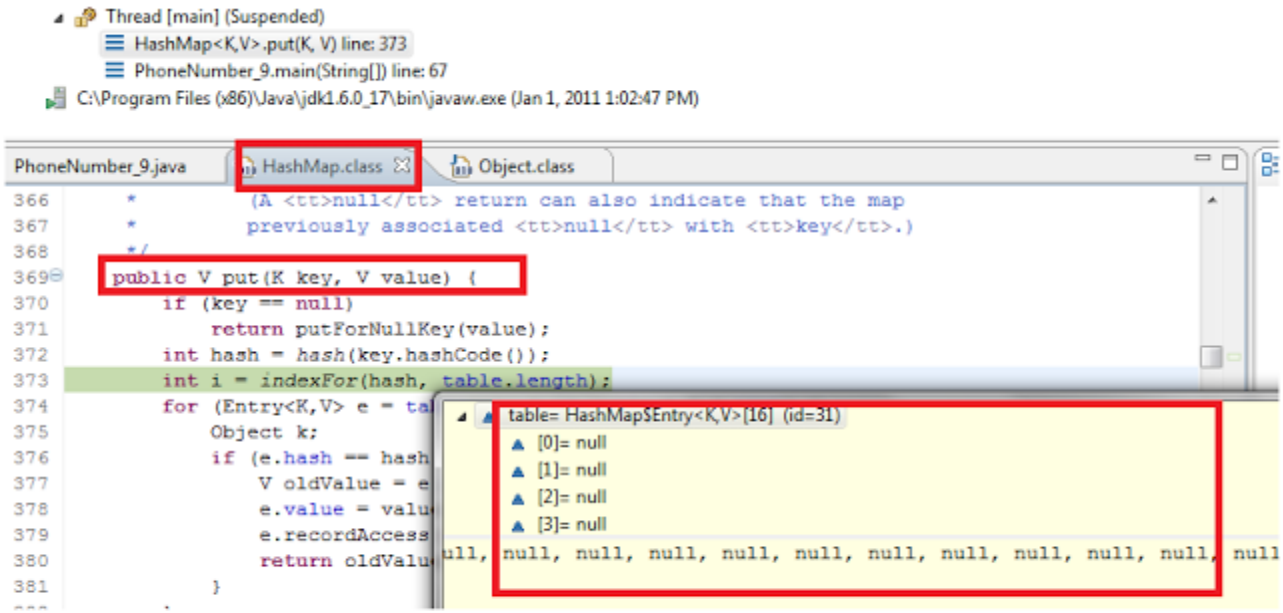
Explanation of the internal working of the HashMap and the concept of keys and buckets

The diagram below shows how a basic HashTable or HashMap is implemented. Basically a HashMap contains an internal Entry table, whose indices are computed by running a hash function on it.

For a more detailed understanding, please read the following article: http://en.wikipedia.org/wiki/Hash_table

WORKFLOW, WHEN THE PHONENUMBER OBJECT {KEY OBJECT} DOES NOT IMPLEMENT HASHCODE().

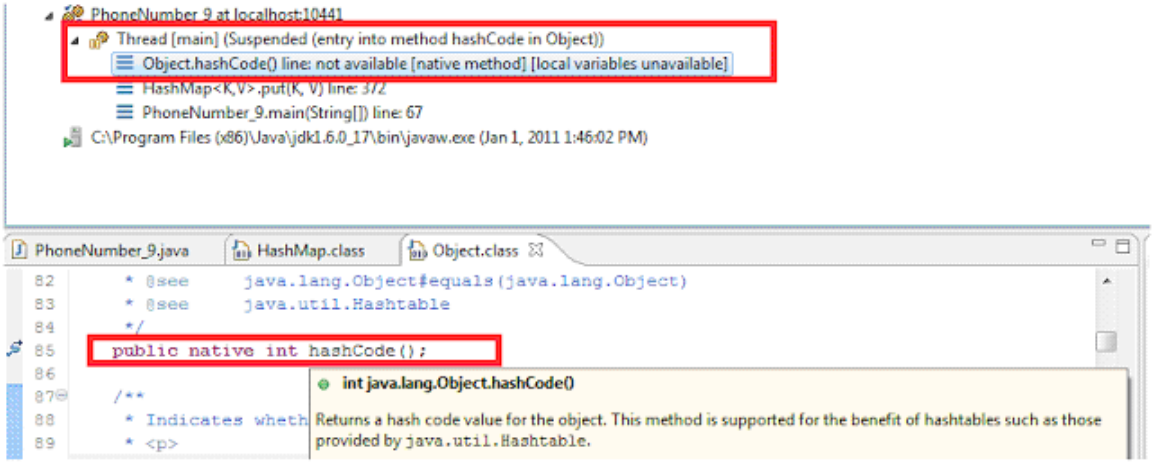
Screenshot1: Internal working of HashMap.put(Key k ,V value) method:



This screenshot displays the algorithm being used by the Hashmap.put() method.
The method basically obtains the hashcode for the key by invoking the hashcode() method on it. If the object provides the implementation, then the hashCode() implementation returns the appropriate value, else the**"Object.hashcode()"** is returned. After that the index value for the bucket or slot is calculated by running a hash function on it.
This is done by running the following two lines:

```
int hash = hash(key.hashCode());
int i = indexFor(hash, table.length);
```

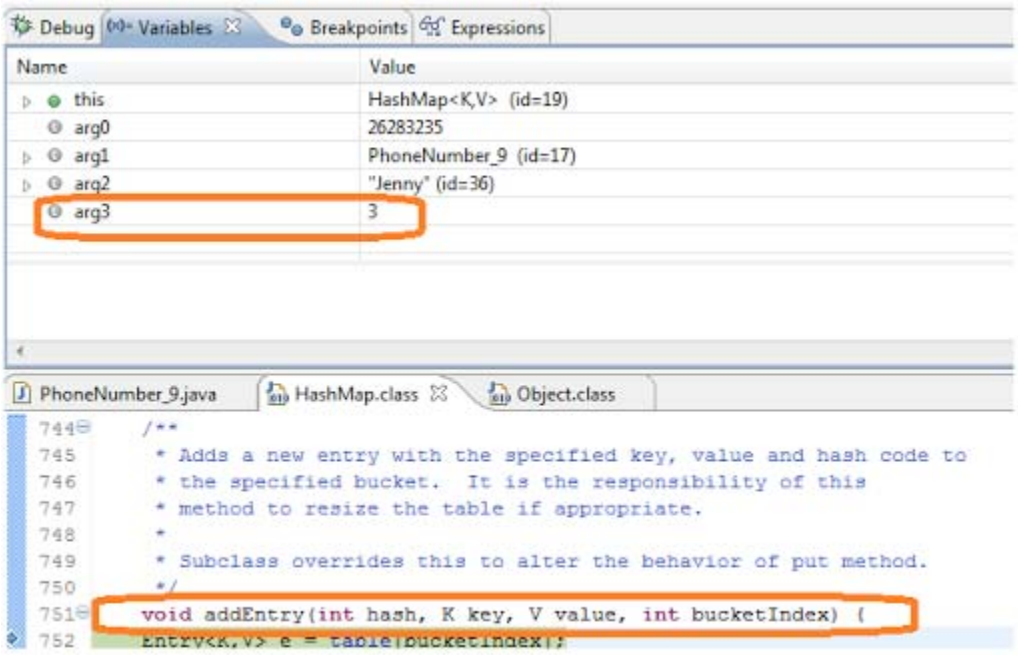
Screenshot2: Who calls hashCode()?



Both HashMap.get() and HashMap.put() invoke the hashCode() method.
This results in a call to ***"Object.hashcode()"*** method, if the Key object does not implement hashCode. This is implemented as native [Non-Java] method. This always generates a unique Id, even if two objects are semantically equal.

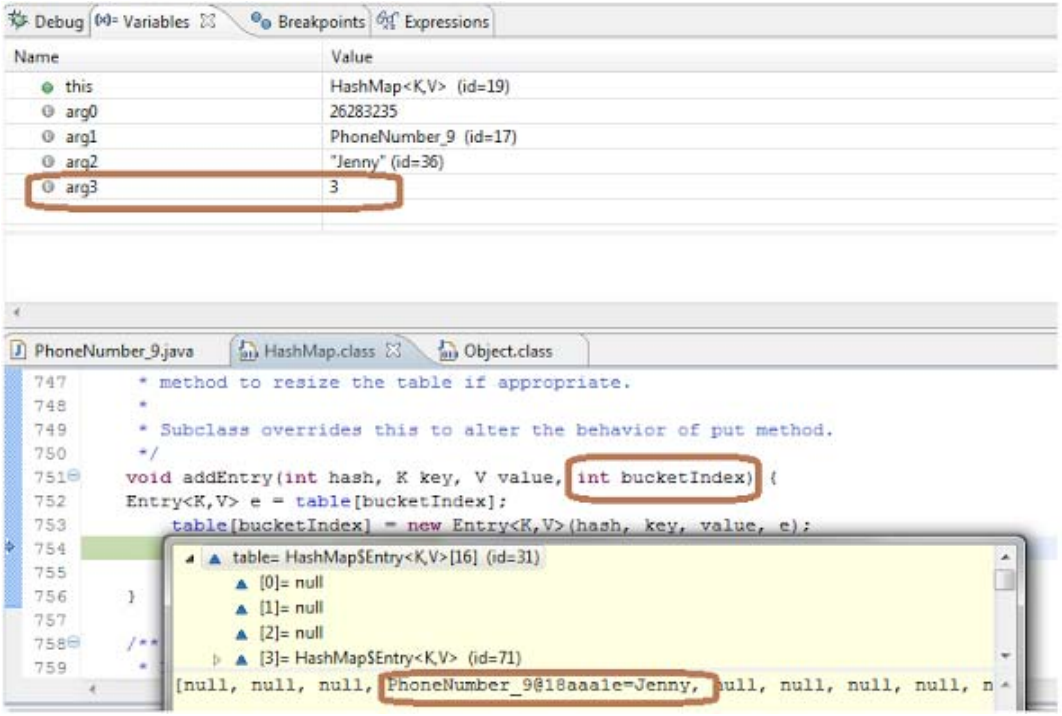
Screenshot3: In which slot/bucket, the new values are put in the HashMap?

The screenshot below highlights the logic used to insert a new value into the HashMap. An index is calculated by the HashMap.indexFor() method. This is then used to insert the entry into the right index position.



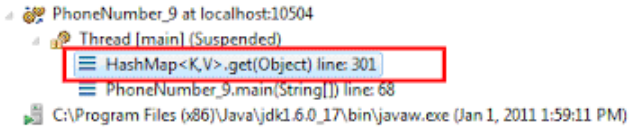
Screenshot 4: Value inserted into the right slot.

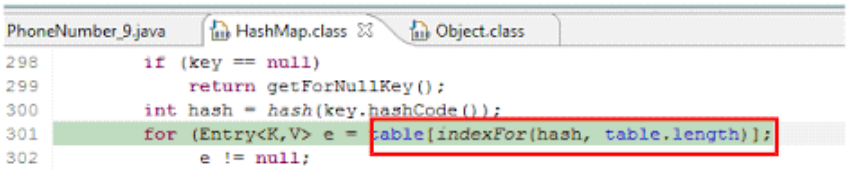
The value [Jenny] for the first instance of the PhoneNumber is inserted into the index position 3.



Screenshot 5: Hash function invoked to calculate the index.

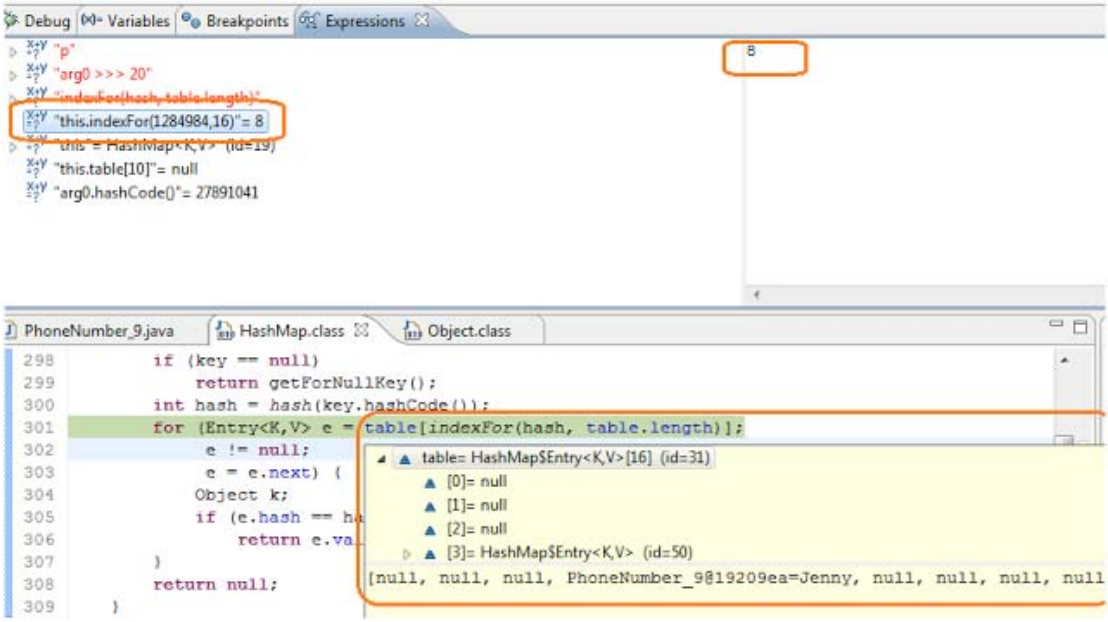
When the second phone number instance is used to retrieve the value from the map, the HashMap.get() method again calls hashcode() method to obtain the hash value, which is then passed as a parameter to the indexFor() method {hash function} used to calculate the index values.





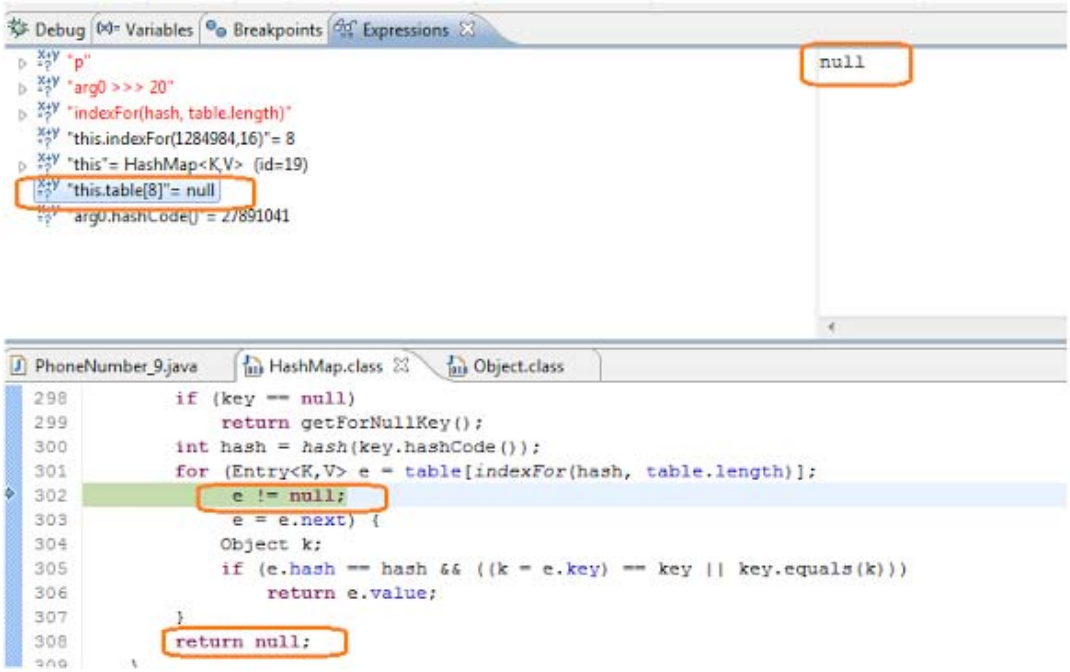
Screenshot 6: How is the appropriate hash index calculated?

The indexFor() method takes the hash value and the length of the internal table to calculate the index. The index calculated for the second PhoneNumber instance is 8.



Screenshot 7: Different phone instance generates a different index values.

As different index values are calculated for the two phone number objects, the second phone number object yields a null value; even though both the phone number objects are equal as implemented by the PhoneNumber.equals() method.

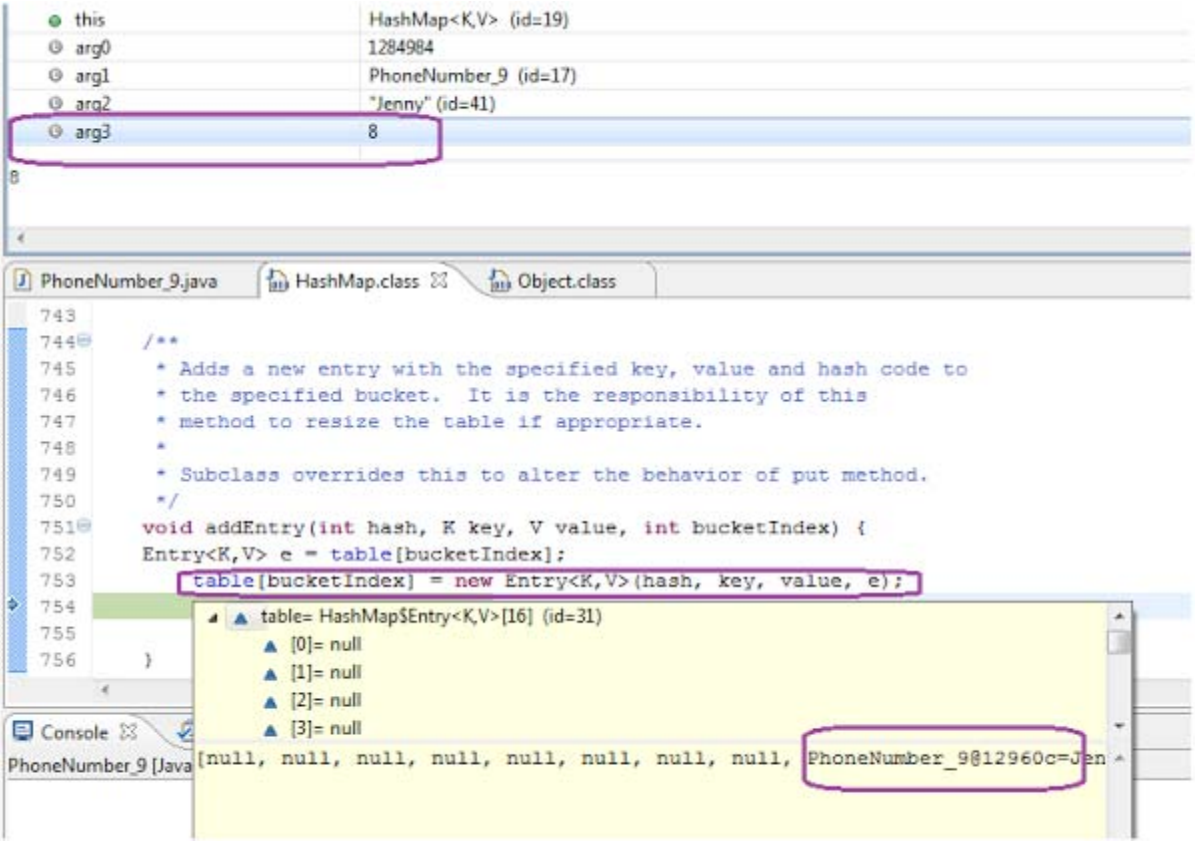


WORKFLOW WHEN THE PHONENUMBER {KEY OBJECT} IMPLEMENTS HASHCODE()

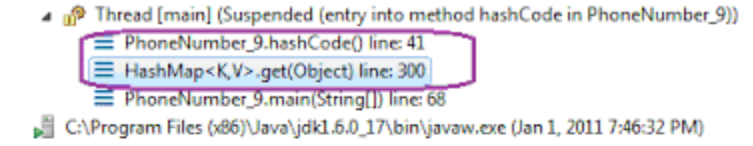
After the Java class PhoneNumber implements HashCode () method, both different instances of the Phone number object generate same index values for the internal table lookup returning the same value from the table. Following screenshots depict the workflow and retrieval of values for two different phone number instances depicting same value.

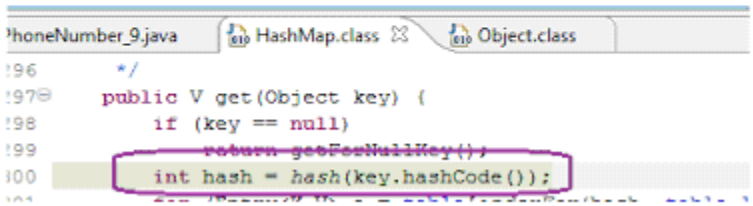
Screenshot1: Index position, when the instance1 of the phone number instance is created.

This displays the index position calculated for the first phone number object as 8.

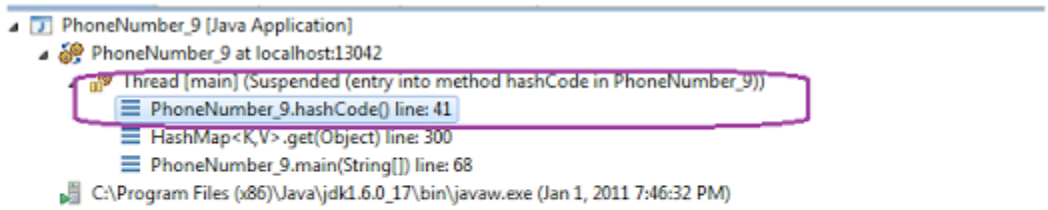


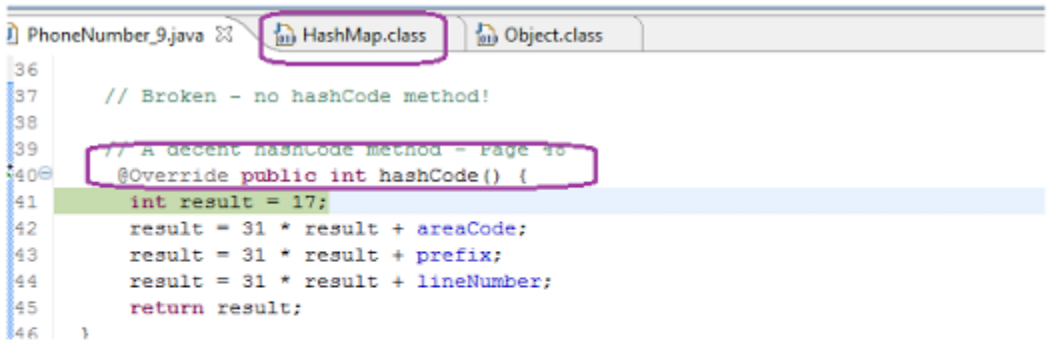
Screenshot2: HashMap.get() internally invokes hashCode() on the second PhoneNumber object.



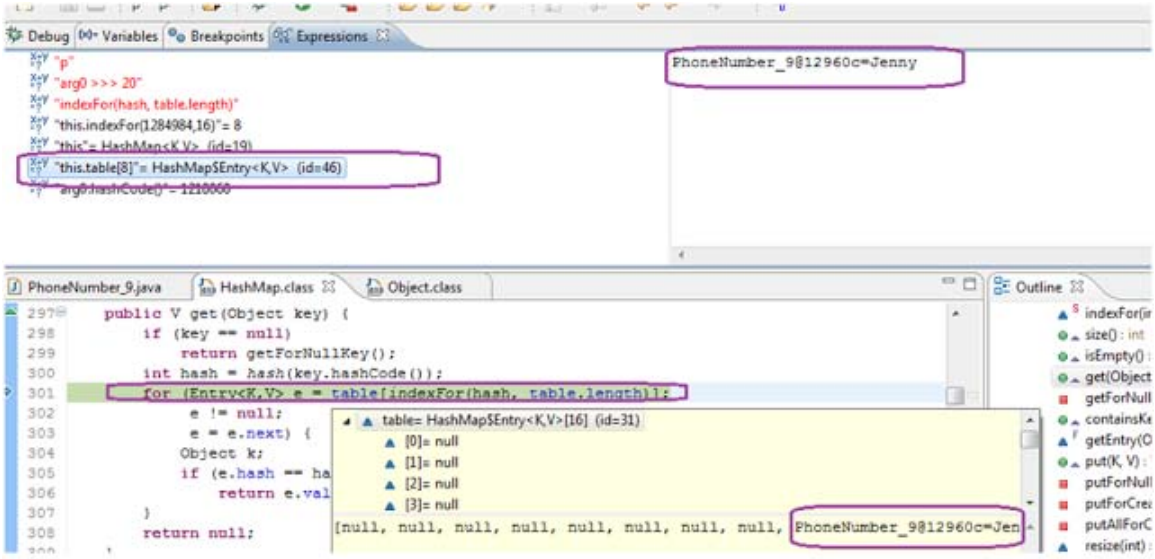


Screenshot 3: The PhoneNumber.HashCode() is called, that returns the same index position.



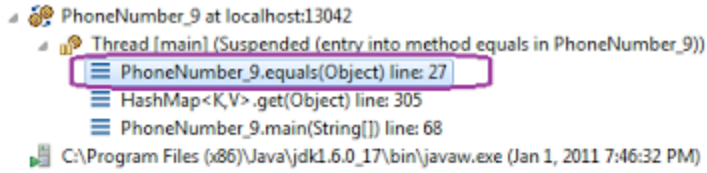


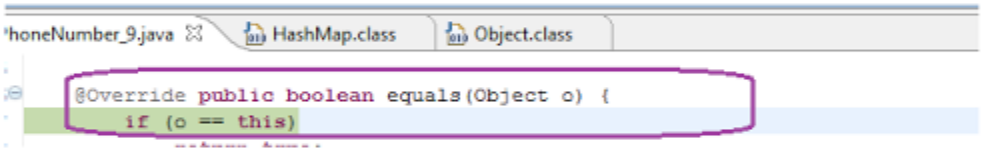
Screenshot 4: Doing a lookup on the same index position returns the same Value object.



Screenshot 5: The significance of equals() method:

The screenshots below, show that the HashMap.get() calls the PhoneNumber.equals method(). The second screenshot highlights the fact that even after a right Map entry is found in the internal table based on the calculated hash index; ***a last check is performed on the equality of both the key objects, before returning the associated value.***





Screenshot6: The right value is returned for the second instance of the phone number object. Null is not returned.