FAQ :

1) **How HashMap Works Internally In Java?**

Ans :

HashMap Internal Structure :

HashMap stores the data in the form of key-value pairs. Each key-value pair is stored in an object of *Entry<K, V>* class. *Entry<K, V>* class is the static inner class of *HashMap* which is defined like below.

```
1  static class Entry<K,V> implements Map.Entry<K,V>
2  {
3      final K key;
4      V value;
5      Entry<K,V> next;
6      int hash;
7
8      //Some methods are defined here
9  }
```

As you see, this inner class has four fields. *key*, *value*, *next* and *hash*.

key : It stores the key of an element and its final.

value : It holds the value of an element.

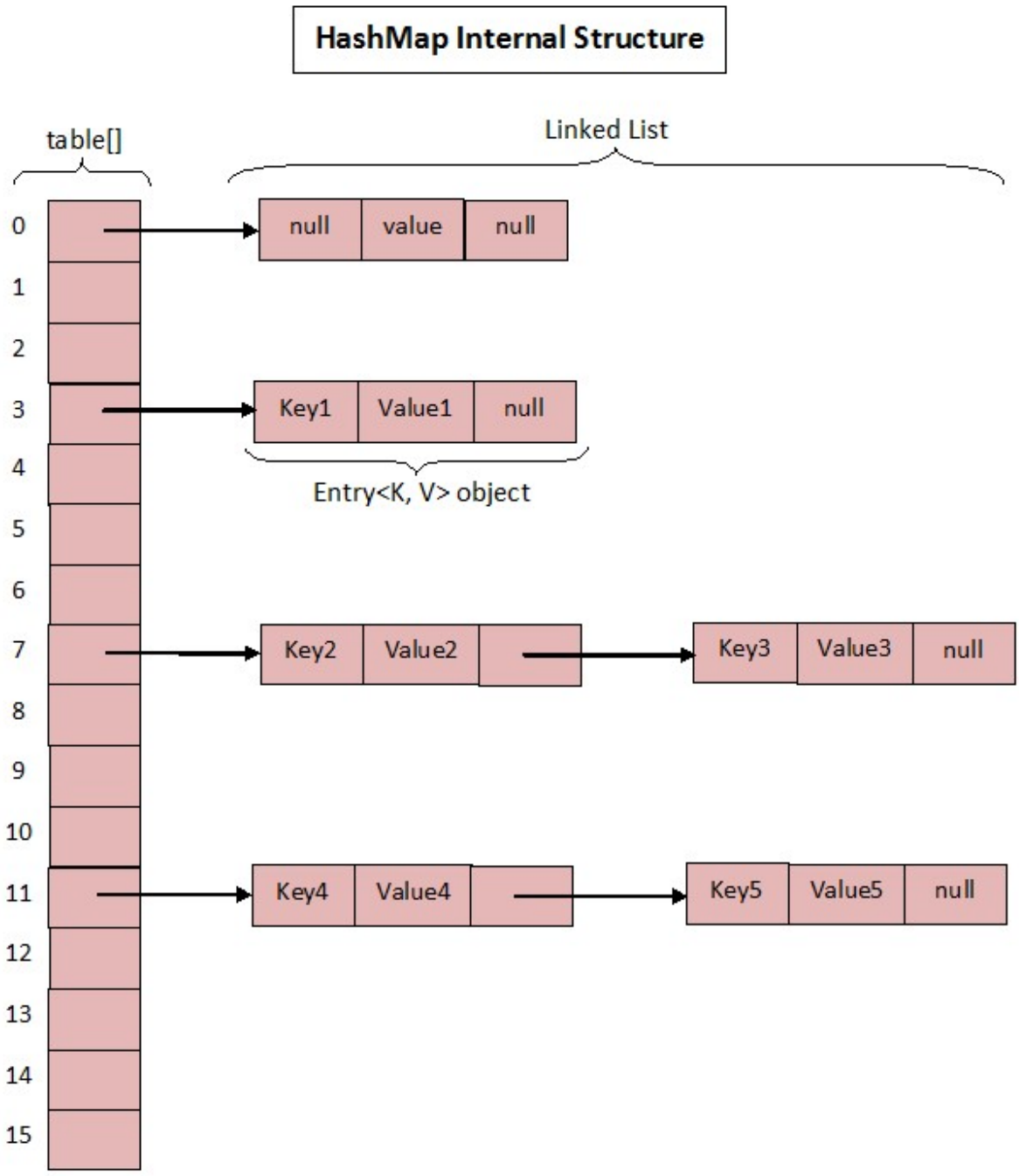
next : It holds the pointer to next key-value pair. ***This attribute makes the key-value pairs stored as a linked list.***

hash : It holds the hashcode of the key.

These *Entry* objects are stored in an array called *table[]*. This array is initially of size 16. It is defined like below.

```
/**
    * The table, resized as necessary. Length MUST Always be a power of two.
    */
   transient Entry<K,V>[] table;
```

To summarize the whole *HashMap* structure, each key-value pair is stored in an object of *Entry<K, V>* class. This class has an attribute called *next* which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these *Entry<K, V>* objects are stored in an array called *table[]*. The below image best describes the *HashMap* structure.



The above image roughly shows how the *HashMap* stores its elements. Internally it uses an array of *Entry<K, V>* class called *table[]* to store the key-value pairs. **But how *HashMap* allocates slot in *table[]* array to each of its key-value pair is very interesting. It doesn't inserts the objects as you put them into *HashMap* i.e first**

element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called *Hashing*.

2) What Is Hashing?

The whole *HashMap* data structure is based on the principle of **Hashing**. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called ***hash code***. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever you insert new key-value pair using *put()* method, *HashMap* blindly doesn't allocate slot in the *table[]* array. Instead it calls hash function on the key. *HashMap* has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented *hashCode()* methods. Below is implementation code of *hash()*.

```
1   /**
2      * Retrieve object hash code and applies a supplemental hash function to
3   the
4      * result hash, which defends against poor quality hash functions.  This is
5      * critical because HashMap uses power-of-two length hash tables, that
6      * otherwise encounter collisions for hashCodes that do not differ
7      * in lower bits. Note: Null keys always map to hash 0, thus index 0.
8      */
9     final int hash(Object k) {
10        int h = 0;
11        if (useAltHashing) {
```

```
12          if (k instanceof String) {
13              return sun.misc.Hashing.stringHash32((String) k);
14          }
15          h = hashSeed;
16      }
17
18      h ^= k.hashCode();
19
20      // This function ensures that hashCodes that differ only by
21      // constant multiples at each bit position have a bounded
22      // number of collisions (approximately 8 at default load factor).
23      h ^= (h >>> 20) ^ (h >>> 12);
24      return h ^ (h >>> 7) ^ (h >>> 4);
    }
```

After calculating the hash code of the key, it calls *indexFor()* method by passing the hash code of the key and length of the *table[]* array. This method returns the index in the *table[]* array for that particular key-value pair.

```
1       /**
2        * Returns index for hash code h.
3        */
4       static int indexFor(int h, int length) {
5           return h & (length-1);
6       }
```

Now, let's see how *put()* method works in detail.

3) **How put() method works?**

Below is the code implementation of *put()* method in the *HashMap* class.

```
1   /**
2       * Associates the specified value with the specified key in this map.
3       * If the map previously contained a mapping for the key, the old
4       * value is replaced.
5       *
6       * @param key key with which the specified value is to be associated
7       * @param value value to be associated with the specified key
8       * @return the previous value associated with <tt>key</tt>, or
9       *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
10      *         (A <tt>null</tt> return can also indicate that the map
11      *         previously associated <tt>null</tt> with <tt>key</tt>.)
12      */
13     public V put(K key, V value) {
14         if (key == null)
15             return putForNullKey(value);
16         int hash = hash(key);
17         int i = indexFor(hash, table.length);
18         for (Entry<K,V> e = table[i]; e != null; e = e.next) {
19             Object k;
20             if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
21                 V oldValue = e.value;
22                 e.value = value;
23                 e.recordAccess(this);
24                 return oldValue;
25             }
26         }
```

```
27
28      modCount++;
29      addEntry(hash, key, value, i);
30      return null;
31    }
```

Let’s see how this code works step by step.

Step 1 : First checks whether the key is null or not. If the key is null, it calls *putForNullKey()* method. *table[0]* is always reserved for null key. Because, hash code of null is 0.

Step 2 : If the key is not null, then it calculates the hash code of the key by calling *hash()* method.

Step 3 : Calls *indexFor()* method by passing the hash code calculated in step 2 and length of the *table[]* array. This method returns index in *table[]* array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index (or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.

How put() method works?

```
public V put(K key, V value)
{
    if (key == null)
      return putForNullKey(value);

    int hash = hash(key);

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

    for (Entry<K,V> e = table[i]; e != null; e = e.next)
    {
      Object k;
      if (e.hash == hash && ((k = e.key) == key ||
key.equals(k)))
      {
        V oldValue = e.value;
        e.value = value;
        e.recordAccess(this);
        return oldValue;
      }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
  }
```

Checks whether key is null or not. If key is null, it is placed at table[0].

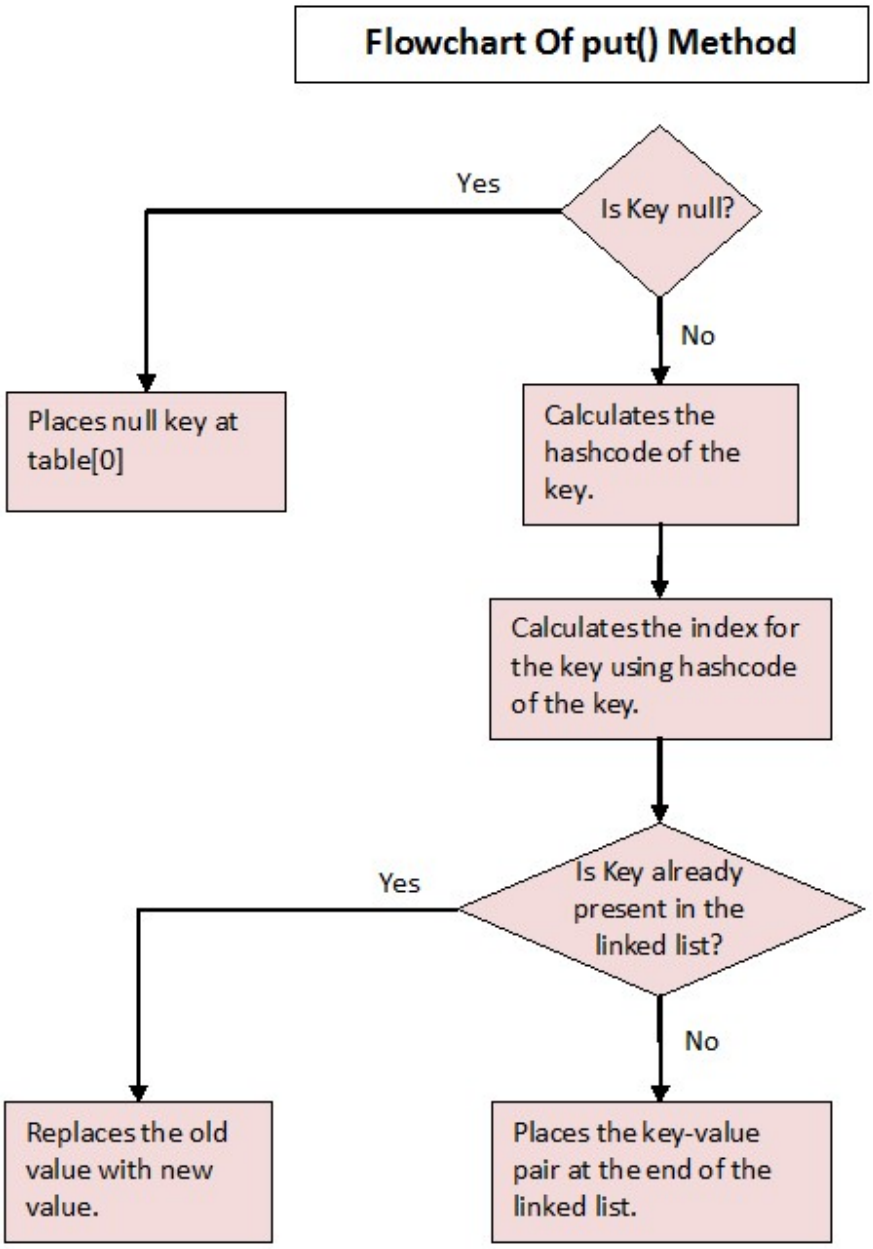
Calculates the hash code for the specified key.

Returns the index in table[] array for specified key-value pair.

This code checks whether the specified key is already present in the linked list or not.

If present, old value is replaced with new value.

If not present, key-value pair is placed at end of the linked list.


Flowchart Of put() Method
Is Key null?
Yes
No
Places null key at table[0]
Calculates the hashcode of the key.
Calculates the index for the key using hashcode of the key.
Is Key already present in the linked list?
Yes
No
Replaces the old value with new value.
Places the key-value pair at the end of the linked list.

4) How get() method Works?

Let’s see how get() method has implemented.

```
1   /**
2   * Returns the value to which the specified key is mapped, or {@code null}
3   * if this map contains no mapping for the key.
4   *
5   *
6
7   * More formally, if this map contains a mapping from a key {@code k} to a
8   * value {@code v} such that {@code (key==null ? k==null :
9   * key.equals(k))}, then this method returns {@code v}; otherwise it returns
10  * {@code null}. (There can be at most one such mapping.)
11  *
12  *
13
14  * A return value of {@code null} does not <i>necessarily</i> indicate that
15  * the map contains no mapping for the key; it's also possible that the map
16  * explicitly maps the key to {@code null}. The {@link #containsKey
17  * containsKey} operation may be used to distinguish these two cases.
18  *
19  * @see #put(Object, Object)
20  */
21  public V get(Object key) {
22      if (key == null)
23      return getForNullKey();
24      int hash = hash(key.hashCode());
25      for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e =
26  e.next) {
```

```
27          Object k;
28          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
29              return e.value;
30      }
31      return null;
    }
```

Step 1 : First checks whether specified key is null or not. If the key is null, it calls *getForNullKey()* method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : *indexFor()* method is used to find out the index of the specified key in the *table[]* array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

5) How to Implement Java's hashCode Correctly?

Thoughts on Hashing

If hashCode is used as a shortcut to determine equality, then there is really only one thing we should care about: **Equal objects should have the same hash code.**

This is also why, if we override equals, we must create a matching hashCode implementation! Otherwise things that are equal according to our implementation would likely not have the same hash code because they use Object's implementation.

The hashCode Contract

The general contract of hashCode **is:**

- **Whenever it is invoked on the same object more than once during an execution of a Java application, the** hashCode **method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.**
- **If two objects are equal according to the** equals(Object) **method, then calling the** hashCode **method on each of the two objects must produce the same integer result.**
- **It is not required that if two objects are unequal according to the** equals(Object) **method, then calling the** hashCode **method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.**

Implementing hashCode

A very easy implementation of Person.hashCode is the following:

```
@Override
public int hashCode() {
   return Objects.hash(firstName, lastName);
}
```

The person's hash code is computed by computing the hash codes for the relevant fields and combining them. Both is left to Objects' utility function hash.

A)Selecting Fields

But which fields are relevant? The requirements help answer this: If equal objects must have the same hash code, then hash code computation should not include any field that is not used for equality checks. (Otherwise two objects that only differ in those fields would be equal but have different hash codes.)

So the set of fields used for hashing should be a subset of the fields used for equality. By default both will use the same fields but there are a couple of details to consider.

B)Consistency

For one, there is the consistency requirement. It should be interpreted rather strictly. While it allows the hash code to change if some fields change (which is often unavoidable with mutable classes), hashing data structures are not prepared for this scenario.

As we have seen above the hash code is used to determine an element's bucket. But if the hash-relevant fields change, the hash is not recomputed and the internal array is not updated.

This means that a later query with an equal object or even with the very same instance fails! The data structure computes the current hash code, different from the one used to store the instance, and goes looking in the wrong bucket.

Conclusion: Better not use mutable fields for hash code computation!

C)Performance

Hash codes might end up being computed about as often as `equals` is called. This can very well happen in performance critical parts of the code so it makes sense to think about performance. And unlike `equals` there is a little more wiggle room to optimize it.

Unless sophisticated algorithms are used or many, many fields are involved, the arithmetic cost of combining their hash codes is as negligible as it is unavoidable. But it should be considered whether all fields need to be included in the computation! Particularly collections should be viewed with suspicion. Lists and sets, for example, will compute the hash for each of their elements. Whether calling them is necessary should be considered on a case-by-case basis.

If performance is critical, using `Objects.hash` might not be the best choice either because it requires the creation of an array for its varargs.

But the general rule about optimization holds: Don't do it prematurely! Use a common hash code algorithm, maybe forego including the collections, and only optimize after profiling showed potential for improvement.

D)Collisions

Going all-in on performance, what about this implementation?

```
@Override
public int hashCode() {
   return 0;
}
```

It's fast, that's for sure. And equal objects will have the same hash code so we're good on that, too. As a bonus, no mutable fields are involved!

But remember what we said about buckets? This way all instances will end up in the same! This will typically result in a linked list holding all the elements, which is terrible for performance. Each `contains`, for example, triggers a linear scan of the list.

So what we want is as few items in the same bucket as possible! An algorithm that returns wildly varying hash codes, even for very similar objects, is a good start.

How to get there partly depends on the selected fields. The more details we include in the computation, the more likely it is for the hash codes to differ. Note how this is completely opposite to our thoughts about performance. So, interestingly enough, using too many *or* too few fields can result in bad performance.

The other part to preventing collisions is the algorithm that is used to actually compute the hash.

E) Computing The Hash

The easiest way to compute a field's hash code is to just call `hashCode` on it. Combining them could be done manually. A common algorithm is to start with some arbitrary number and to repeatedly multiply it with another (often a small prime) before adding a field's hash:

```
int prime = 31;
int result = 1;
result = prime * result + ((firstName == null) ? 0 : firstName.hashCode());
result = prime * result + ((lastName == null) ? 0 : lastName.hashCode());
return result;
```

This might result in overflows, which is not particularly problematic because they cause no exceptions in Java.

Note that even great hashing algorithms might result in uncharacteristically frequent collisions if the input data has specific patterns. As a simple example assume we would compute the hash of points by adding their x and y-coordinates. May not sound too bad until we realize that we often deal with points on the line f(x) = -x, which means x + y == 0 for all of them. Collisions, galore!

But again: Use a common algorithm and don't worry until profiling shows that something isn't right.

Summary
We have seen that computing hash codes is something like compressing equality to an integer value: Equal objects must have the same hash code and for performance reasons it is best if as few non-equal objects as possible share the same hash.
This means that `hashCode` must always be overridden if `equals` is.
When implementing `hashCode`:
• **Use a the same fields that are used in `equals` (or a subset thereof).**
• **Better not include mutable fields.**
• **Consider not calling `hashCode` on collections.**
• **Use a common algorithm unless patterns in input data counteract them.**
Remember that `hashCode` is about performance, so don't waste too much energy unless profiling indicates necessity.

6) Mutable Object As Collection Key

It is a general advice that you should use immutable object as a key in a Collection. HashCode work best when calculated from immutable data. If you use Mutable object as key and change the state of the object so that the hashCode changes, then the store object will be in the wrong bucket in the Collection

The most important thing you should consider while implementing hashCode() is that regardless of when this method is called, it should produce the same value for a particular object every time when it is called. If you have a scenario like an object produces one hashCode() value when it is **put()** into a HaspMap and produces another value during a **get()**, in that case, you would not be able to retrieve that object. Therefore, if you hashCode() depends on mutable data in the object, then made changing those data will surely produce a different key by generating a different hashCode().
Look at the example below

```
import java.util.HashMap;
import java.util.Map;

public class Employee {

    private String name;
    private int age;

    public Employee() {
    }

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    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;
    }
```

```
    @Override
    public boolean equals(Object obj) {
        //Remember: Some Java gurus recommend you avoid using instanceof
        if (obj instanceof Employee) {
            Employee emp = (Employee)obj;
            return (emp.name == name && emp.age == age);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return name.length() + age;
    }

    public static void main(String[] args) {
        Employee e = new Employee("muhammad", 24);
        Map<Object,Object> m = new HashMap<Object,Object>();
        m.put(e, "Muhammad Ali Khojaye");

        // getting output
        System.out.println(m.get(e));

        e.name = "abid";
        // it fails to get
       System.out.println(m.get(e));

        // it fails again
        System.out.println(m.get(new Employee("muhammad", 24)));
    }
}
```

After changing mutableField, the computed `hashCode` value is no longer pointing to the old bucket and the `contains()` returns false.
We can tackle such situation using either of these methods

- `Hashcode` is best when calculated from `immutable data`; therefore ensure that only `immutable` object would be used as key with Collections.
- If you need `mutable` fields included in the `hashCode` method then you need to ensure that object state is not changing after they've been used as `Key` in a hash-based collection. If for any reason it changed, you can calculate and store the `hash value`when the object updates `mutable` field. To do this, you must first remove it from the `collection(set/map)` and then add it back to the `collection` after updating it.

7) Guidelines for generating a hashCode() value ?

Joshua Bloch in Effective Java provides good guidelines for generating a hashCode() value

1. Store some constant nonzero value; say **17**, in an int variable called **result**.

2. For each significant field f in your object (each field taken into account by the equals()), do the following

a. Compute an int hashCode c for the field:

i. If the field is a boolean, compute **c = (f ? 1 : 0).**

ii. If the field is a byte, char, short, or int, compute **c = (int) f**.

iii. If the field is a long, compute **c = (int) (f ^ (f >>> 32))**.

iv. If the field is a float, compute **c = Float.floatToIntBits(f)**.

v. If the field is a double,compute **long l = Double.doubleToLongBits(f)**,
c = (int)(l ^ (l >>> 32))

vi. If the field is an object reference then equals( ) calls equals( ) for this field. compute
c = f.hashCode()

vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hashCode for each significant element by applying above rules to each
element

b. Combine the hashCode c computed in step 2.a into result as follows:**result = 37 * result + c;**

3. Return result.

4. Look at the resulting hashCode() and make sure that equal instances have equal hash codes.

Here is an example of a class that follows the above guidelines

```
1                public class HashTest {
2                  private String field1;
3                  private short  field2;
4                            ----
5
6                    @Override
7                  public int hashCode() {
8                     int result = 17;
9       result = 37*result + field1.hashCode();
10      result = 37*result + (int)field2;
11                   return result;
12                        }
13                        }
```

You can see that a constant **37** is chosen. The purpose of choosing a *prime number* is that it is a *prime number*. We can choose any other *prime number*. Using *prime number* the objects will be distributed better over the buckets

8)*Apache HashCodeBuilder ?*

Writing a good hashCode() method is not always easy. Since it can be difficult to implement hashCode() correctly, it would be helpful if we have some reusable implementations of these.

The ***Jakarta-Commons*** org.apache.commons.lang.builder package is providing a class named HashCodeBuilder which is designed to help implement a hashCode() method. Usually, developers struggle hard with implementing a hashCode() method and this class aims to simplify the process.
Here is how you would implement hashCode algorithm for our above class

```
1   public class HashTest {
2     private String field1;
3     private short  field2;
4           ----
5
6       @Override
7    public int hashCode() {
8  return new HashCodeBuilder(83, 7)
9         .append(field1)
10        .append(field2)
11        .toHashCode();
12          }
13          }
```

Note that the two numbers for the constructor are simply two different, non-zero, odd numbers - these numbers help to avoid collisions in the hashCode value across objects.

If required, the superclass hashCode() can be added using **appendSuper(int)**.
You can see how easy it is to override HashCode() using Apache HashCodeBuilder.

9) Which statements are true about comparing two instances of the same class, given that the equals() and hashCode() methods have been properly overridden? (Choose all that apply.)

A. If the equals() method returns true, the hashCode() comparison == might return false.

B. If the equals() method returns false, the hashCode() comparison == might return true.

C. If the hashCode() comparison == returns true, the equals() method must return true.

D. If the hashCode() comparison == returns true, the equals() method might return true.

E. If the hashCode() comparison != returns true, the equals() method might return true.

Answer: B and D. B is true because often two dissimilar objects can return the same hashcode value. D is true because if the hashCode() comparison returns ==, the two objects might or might not be equal.

A, C, and E are incorrect. C is incorrect because the hashCode() method is very flexible in its return values, and often two dissimilar objects can return the same hash code value. A and E are a negation of the hashCode() and equals() contract.

10) Given:

```
import java.util.*;
class MapEQ {
 public static void main(String[] args) {
   Map<ToDos, String> m = new HashMap<ToDos, String>();
   ToDos t1 = new ToDos("Monday");
   ToDos t2 = new ToDos("Monday");
 ToDos t3 = new ToDos("Tuesday");
  m.put(t1, "doLaundry");
```

```
   m.put(t2, "payBills");
   m.put(t3, "cleanAttic");
  System.out.println(m.size());
 }
 }
 class ToDos{
  String day;
ToDos(String d) {
day = d;
}
  public boolean equals(Object o) {
       return ((ToDos)o).day == this.day;
 }
 // public int hashCode() { return 9; } }
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile.

B. As the code stands the output will be 2.

C. As the code stands the output will be 3.

D. If the hashCode() method is uncommented the output will be 2.

E. If the hashCode() method is uncommented the output will be 3.

F. If the hashCode() method is uncommented the code will not compile.

Answer:

C and D are correct. If hashCode() is not overridden then every entry will go into its own bucket, and the overridden equals() method will have no effect on

determining equivalency. If hashCode() is overridden, then the overridden equals() method will view t1 and t2 as duplicates.

A, B, E, and F are incorrect based on the above.

11) Is Arraylist synchronized? What if I would like to synchronize it?

No, Collections.synchronized method

```
1. //create an ArrayList object
2.    ArrayList arrayList = new ArrayList();
3.
4.    /*
5.     Java ArrayList is NOT synchronized. To get synchronized list from
6.     ArrayList use
7.     static void synchronizedList(List list) method of Collections class.
8.    */
9.
10.   List list = Collections.synchronizedList(arrayList);
11.
12.   /*
13.     Use this list object to prevent any unsynchronized access to original
14.     ArrayList object.
15.
```

12) How Arraylist works?

*ArrayList **internally uses array object** to add(or store) the elements. In other words, ArrayList is backed by Array data -structure.The array of ArrayList is **resizable (or dynamic).***

13) How HashSet works internally in java ?

HashSet internally uses HashMap to maintain the uniqueness of elements

When we have a look at the HashSet.java in java API, we can see the following code:

```
public class HashSet
    extends AbstractSet
    implements Set, Cloneable, java.io.Serializable
{

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }

    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }

    /**
    * Some code
    */
}
```

`Set` achieves the uniqueness in its elements through `HashMap`. In `HashMap`, each key is unique. So, when an object of `HashSet` is created, it will create an object of `HashMap`. When an element is passed to `Set`, it is added as a key in the `HashMap` in the `add(Element e)` method. Now, a value needs to be associated to the key. Java uses a Dummy value (`new Object`) which is called `PRESENT` in `HashSet`.

14) Flow chart for common used collection classes :