**Java interview preparation**

**Q. How does LinkedList work?**

LinkedList => Doubly-linked list implementation of the List, Queue and Deque interfaces, as Deque extends the Queue interface. Implements all optional list operations, and permits all elements (including null).

**This implementation is not synchronized.** It *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the list (best done at creation time).

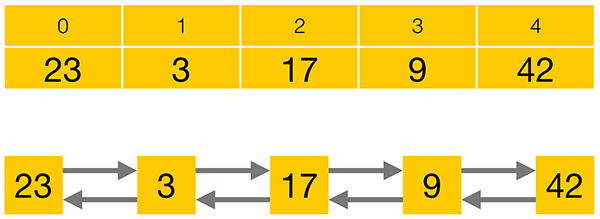
List list = Collections.synchronizedList(new LinkedList(...));

The iterators returned by this class's iterator and listIterator methods are ***fail-fast*:** if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html).

It's a series of instances of a private nested class Entry, which has next, previous and element references. LinkedList is a chain of entities in which every entity knows about next-one, so get(index) operation requires iterating over this chain with counter. But this list optimized for adding and deleting by position.

**Vs** **Linked List data structure:** it is an abstract concept, independent of any specific programming language. The LinkedList Java class is a concrete implementation of this abstract concept.

**Vs ArrayList:** it is based on an Array data structure, while LinkedList is based on a Doubly Linked List data structure:



Compared to a LinkedList, storing elements in an ArrayList consumes less memory and generally gives faster access times. Adding or removing elements is usually faster for a LinkedList, but the performance loss for iterating to the corrrect position often prevails over the performance gain in adding or removing an element.

LinkedList also implements the Queue and the Deque interfaces which give it some additional functionality over ArrayList.

Queue interface consists of three simple operations:

* **add an element** to the end of the Queue
* **retrieve an element** from the front of the Queue, without removing it
* **retrieve and remove an element** from the front of the Queue.

Queue interface offers each of its operations in two flavours –**one method that will throw an Exception, and one that will return a special value in certain cases**:

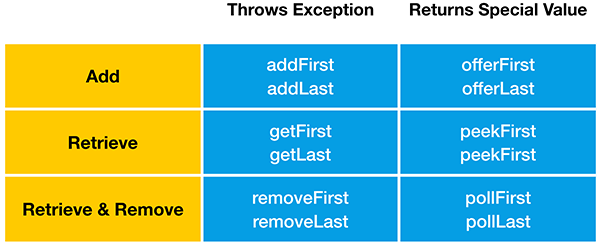


“**add**” will throw an Exception when the Queue is full, while “**offer**” will return false in this case. LinkedList, like most Queue implementations, has an unlimited capacity, so it will never be full. **ArrayBlockingQueue**, on the other hand, is a Queue implementation that has a limited capacity.

**retrieve** an element from the front of the Queue, without removing it.

Finally, you can retrieve and remove an element from the front of the Queue. If the Queue is empty, remove will throw an Exception, while poll will return false.

**Deque** is the short form of “Double Ended Queue”, so it is a Queue that can be accessed from either end.



LinkedList has an unlimited capacity, so it will never be full. **LinkedBlockingDeque**, on the other hand, is a Deque implementation that may have a limited capacity.

**Stack data structure:** Deque interface also supports the methods of the Stack data structure, “**push**” “**peek**” and “**pop**”. Therefore java.util.LinkedList can also be used as Stack.

“**push**” adds an element to the top of the Stack. Equivalent to “addFirst” method

“**peek**” retrieves but does not remove. Equivalent to “peekFirst” method

“**pop**” retrieves and removes an element from the top of the Stack. Equivalent to the “removeFirst” method.

**Collections intro:**

A **collection** - sometimes called a **container** - is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

**Collections Framework?**

A collections framework is a **unified architecture for representing and manipulating collections**. All collections frameworks contain the following:

* **Interfaces:** => **abstract data types** that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations:** => **concrete implementations** of the collection interfaces. In essence, they are **reusable** **data structures**.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. In essence, algorithms are reusable functionality.

**Benefits of the Java Collections Framework**

* **Reduces programming effort:** By providing useful data structures & algorithms and by facilitating interoperability among unrelated APIs.
* **Increases program speed and quality:** provides high-performance, high-quality implementations of useful data structures and algorithms.
* **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth.
* **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output.
* **Reduces effort to design new APIs:**  Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
* **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable.

**Collection Interfaces:**

The *core collection interfaces* encapsulate different types of collections:



* A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth.
* Hierarchy consists of two distinct trees - a Map is not a true Collection.
* All core collection interfaces are generic. Ex. public interface Collection<E>... Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

**Collection** - the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

**Set** - a collection that **cannot contain duplicate** elements. This interface models the mathematical set abstraction and is used to represent sets, such as the courses making up a student's schedule, or the processes running on a machine.

**List** - an **ordered collection** (sometimes called **a sequence**). Lists **can contain duplicate** elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

**Queue** - a collection used to **hold multiple elements prior to processing**.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

**Deque** - a collection used to hold multiple elements prior to processing.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.

**Map** - **an object that maps keys to values**. A Map **cannot contain duplicate** **keys**; each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

**SortedSet** - a Set that maintains its **elements in ascending order**. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

**SortedMap** - a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

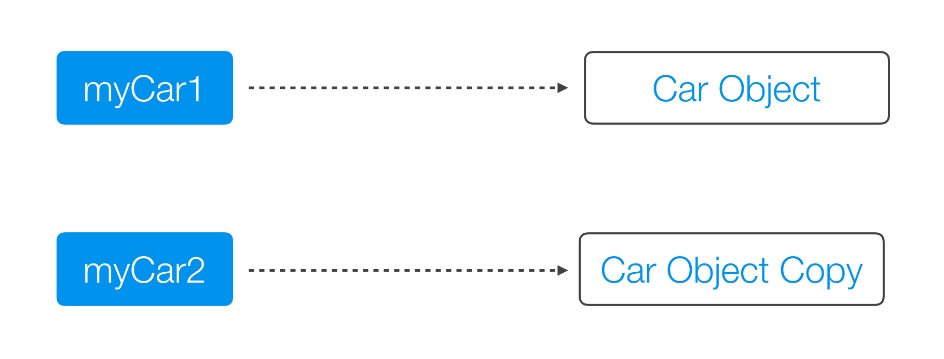
**Difference between shallow cloning and deep cloning of objects?**

**Copy:**

* A **reference copy**, as the name implies, creates a copy of a reference variable pointing to an object.



* An **object copy** creates a copy of the object itself.



Both a **Deep Copy** and a **Shallow Copy** are **types of object copies**. An object copy, usually called a **clone**, is created if we want to modify or move an object, while still preserving the original object.

**A shallow copy** of an object copies the ‘main’ object, but doesn’t copy the inner objects. The **‘inner objects’ are shared** between the original object and its copy. For example, in our Person object, we would create a second Person, but both objects would share the same Name and Address objects.

public class Person {

private Name name;

private Address address;

public Person(Person originalPerson) {

this.name = originalPerson.name;

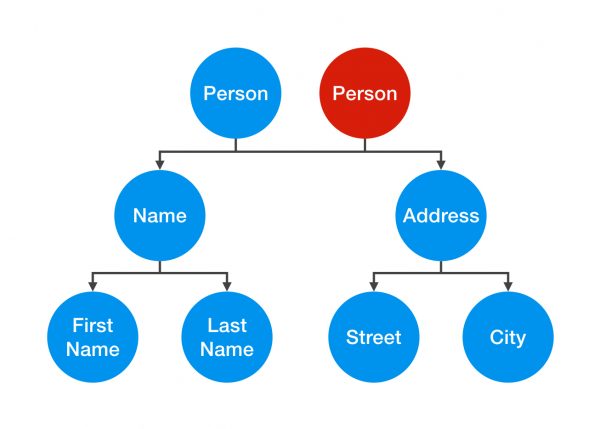
this.address = originalPerson.address;

}

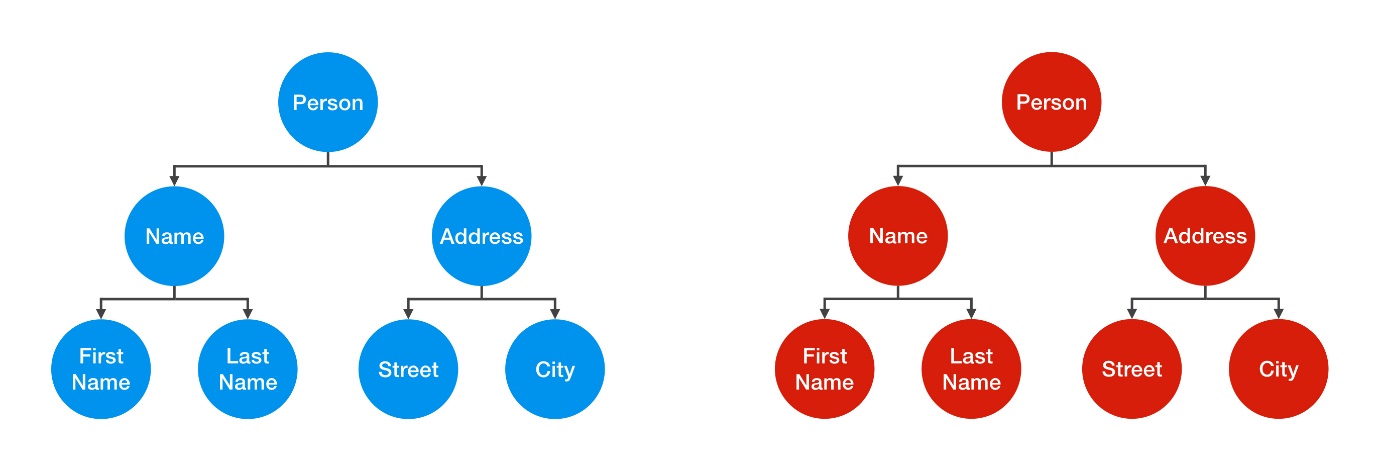
[…]

}

The problem with the shallow copy is that the two objects are not independent. If you modify the Name object of one Person, the change will be reflected in the other Person object.



A **deep copy** is a**fully independent copy of an object**. If we copied our Person object, we would copy the entire object structure.



A change in the Address object of one Person wouldn’t be reflected in the other object.

public class Person {

private Name name;

private Address address;

public Person(Person otherPerson) {

this.name = new Name(otherPerson.name);

this.address = new Address(otherPerson.address);

}

[…]

}

However, that’s not the end of the story. To create a true deep copy, we need to keep copying all of the Person object’s nested elements, until there are only primitive types and “**Immutables**” left.

public class Street {

private String name;

private int number;

public Street(Street otherStreet){

this.name = otherStreet.name;

this.number = otherStreet.number;

}

[…]

}

In Street object, int number is a primitive value and not an object. It’s just a simple value that can’t be shared, so by creating a second instance variable, we are automatically creating an independent copy.

String is an Immutable. Therefore, you can share it without having to create a deep copy of it.

The default behavior of an **object’s clone()** method **automatically yields a shallow copy**. So to achieve a deep copy the classes must be edited or adjusted. **clone()** method returns ‘Object’ as type and you need to explicitly cast back to your original object.

The class which you want to be cloned should implement clone method and overwrite it. It should provide its own meaning for copy or to the least it should invoke the **super.clone().** Also you have to implement **Cloneable marker interface** or else you will get CloneNotSupportedException. When you invoke the super.clone() then you are dependent on the Object class’s implementation and what you get is a shallow copy.

When you need a **deep copy** then you need to **implement it yourself**. When you implement deep copy be careful as you **might fall for cyclic dependencies**. If you don’t want to implement deep copy yourselves then you can **go for serialization**. It does **implements deep copy implicitly and gracefully handling cyclic dependencies**.

**Can we have interfaces with no defined methods?**

The interfaces with no defined methods act like markers. They just **tell the compiler** that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. **Marker interfaces** are also known as **“tag” interfaces**.

**What is the difference between “==” and equals() method?**

The **== (double equals)** returns true, **if the variable reference points to the same object** in memory. This is called “**shallow comparison**”.

The **equals() method** calls the user implemented equals() method, which compares the object attribute values. The equals() method provides “**deep comparison**” by checking if two objects are **logically equal** as opposed to the shallow comparison provided by the operator ==.

If equals() method does not exist in a user supplied class then the inherited **Object class's equals() method** will be called which evaluates if the references point to the same object in memory. In this case, the **object.equals() works just like the "==" operator**.

**Can I use a Class Object as Key in HashMap?**

Yes. In order to be used as a HashMap key, the class has to implement hashCode() and equals() methods to reflect "equality" of two objects.

If you create two different instances with

Key a = new Key("xyz");

Key b = new Key("xyz");

and expect them to be equal and work in a HashMap, you have to override hashCode() so that it returns the same value in both instances, and equals() returns true when comparing them.

If the object identity is based on the string value, then

@Override

public int hashCode()

{

return theStringValue.hashCode();

}

and

@Override

public boolean equals(Object o)

{

return this.theStringValue.equals(o);

}

should work.

In eclipse, just right click on class and go to Source>Generate hashCode() and equals()… choose the attributes you want to include in these methods and the code will be auto-generated.

After overriding hashcode and equals method, you need to use your object while getting data from hashMap. Ex.

HM.get(new Account(2193,"Uri"));

A great care must be exercised if mutable objects are used as map keys. You should make the keys immutable so they do not change.

Joshua Bloch says on Effective Java:

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

If two objects are equal, their hashcodes must be equal as well.

Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be stored in the collection, and the hashcode is used again to help locate the object in the collection.

Hashing retrieval is a two-step process.

1. Find the right bucket (using hashCode())
2. Search the bucket for the right element (using equals() )

**How does HashMap work?**

It works on the **hashing principle**.

* **Hashing** is the mechanism of **assigning unique code to a variable** or attribute using an algorithm **to enable easy retrieval**. A true hashing mechanism should **always return the same hashCode()** when it is applied to the same object.
* Hashing is a process of converting an object into integer form by using the method hashCode(). Its necessary to write hashCode() method properly for better performance of HashMap.
* **hashCode() method** of object class returns the memory reference of object in integer form. In HashMap, it is used to calculate the bucket and therefore calculate the index.
* **equals method** is used to check that 2 objects are equal or not. HashMap uses equals() to **compare the key** whether the are equal or not.
* A **bucket** is one **element of HashMap array**. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes. A single bucket can have more than one nodes, it depends on hashCode() method. The better your hashCode() method is, the better your buckets will be utilized.
* **Index Calculation in Hashmap:**

Hash code of key may be large to create an array => may cause outOfMemoryException. Hence, we generate index to minimize the size of array.

index = hashCode(key) & (n-1); where n is number of buckets or the size of array. 16 is the default size. & is bitwise AND operator.

The JAVA HashMap class implements the interface Map<K,V>. The main methods of this interface are:

* V put(K key, V value)
* V get(Object key)
* V remove(Object key)
* Boolean containsKey(Object key)

HashMaps use an **inner class** to store data: the **Entry<K, V>**. This entry is a simple key-value pair with two extra data:

* a reference to another Entry so that a HashMap can store entries like singly linked lists
* a hash value that represents the hash value of the key. This hash value is stored to avoid the computation of the hash every time the HashMap needs it.

Here is a part of the Entry implementation in JAVA 7:

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

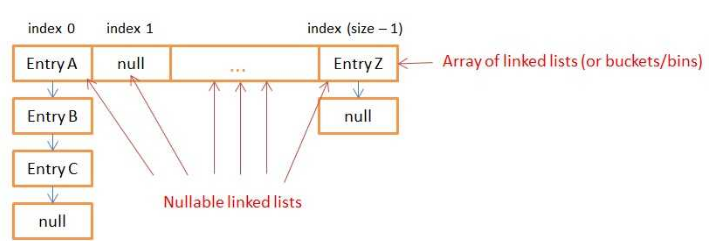
Entry<K,V> next;

final int hash;

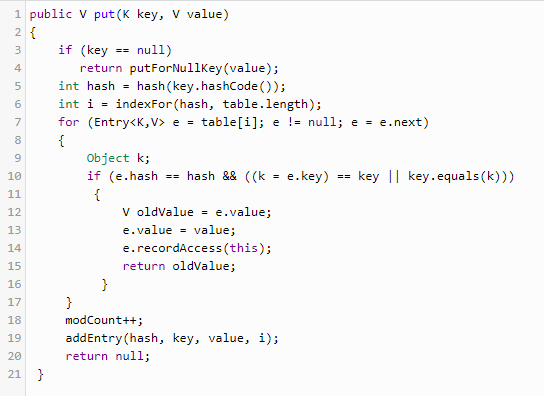
........

}

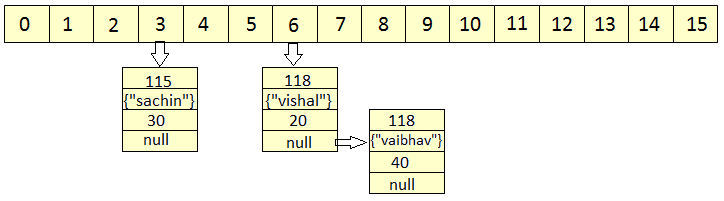
A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**). All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.



**How Does Put() Method Work Internally?**



* First - If the **given key is null**, it will be stored in the **zero position**. As HashMap also allows null key, and hash code of null will always be 0.
* Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode. Say 118
* The indexFor() method is used to get the exact location to store the **Entry** object. For 118 hashcode, it is 6.
* Then comes the most important part what happens if two different object has the same hashcode( eg : Aa, BB will have the same hashcode) and will it be stored in the same bucket. Like the next attribute of LinkedList in data structure; the **next attribute in the Entry** class points to the next object. Using this, different objects with the same hashcode will be placed next to each other.
* In the case of the **Collision**, the HashMap checks for the value of the **next attribute** if it is null it inserts the Entry object in that location, if next attribute is not null then it keeps the loop running till next attribute is null then stores the Entry object there.



* HashMap **doesn't allow duplicate keys**, even though when we insert the same key with different values, only the latest value is returned.

HashMap uses equals() method to check for the equality of key. If **key.equals(k)**is true, then it will replace the value object inside the Entry class. Otherwise connect this node object to the previous node object via linked list and both are stored in the same bucket.

* **How Does Get() Method Work Internally?**
* First, it gets the hash code of the key object, which is passed, and finds the bucket location.
* If the correct bucket is found, it returns the value (e.value)
* If no match is found, it returns null.
* **What Happens If Two Keys Have the Same Hashcode?**

The same **collision** resolution mechanism will be used here. key.equals(k) will check until it is true, and if it is true, it returns the value of it.

* HashMap **has the ability to increase its inner array** in order to keep very short linked lists. When you create a HashMap, you can specify an initial size and a loadFactor with the following constructor:

public HashMap(int initialCapacity, float loadFactor)

If you don’t specify arguments, the default initialCapacity is 16 and the default loadFactor is 0.75. The initialCapacity represents to the size of the inner array of linked lists.

Each time you add a new key/value in your Map with put(…), the function checks if it needs to increase the capacity of the inner array.

**Threshold** is **capacity multiplied by load factor** and whenever we try to add an entry, if map size is greater than threshold, HashMap **rehashes** the contents of map into a new array with a larger capacity.

HashMap only increases the size of the inner array, it **doesn’t provide a way to decrease** it.

* **HashTable** implementation is a **thread safe implementation**. But, since all the CRUD methods are synchronized this implementation is very slow.

A smarter implementation of a thread safe HashMap exists since JAVA 5: the **ConcurrentHashMap**. Only the buckets are synchronized so multiples threads can get(), remove() or put() data at the same time.

* Why Strings and Integers are a good implementation of keys for HashMap? Mostly because they are **immutable**! If you choose to create your own Key class and don’t make it immutable, you might lose data inside the HashMap.

<http://coding-geek.com/how-does-a-hashmap-work-in-java/>

* When using a HashMap, you need to find a hash function for your keys that **spreads the keys into the most possible buckets**. To do so, you need to **avoid hash collisions**. The String Object is a good key because of it has good hash function. Integers are also good because their hashcode is their own value.

## Skewed HashMap vs well balanced HashMap

If the hash function of your key is ill-designed, you’ll have a **skew** repartition (no matter how big the capacity of the inner array is). All the put() and get() that use the biggest linked lists of entry will be slow because they’ll need to iterate the entire lists.

Well balanced HashMap has a hash (of the key) function that gives better repartition for entries in the buckets.

**Notes:**

1. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
2. If key given already exist in HashMap, the value is replaced with new value.
3. hash code of null key is 0.
4. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

<http://www.geeksforgeeks.org/internal-working-of-hashmap-java/>

**Java 8 changes to HashMap:**

The performance has been improved by using **balanced trees** instead of linked lists under specific circumstances. It has only been implemented in the classes java.util.HashMap, java.util.LinkedHashMap and java.util.concurrent.ConcurrentHashMap.

In JAVA8, you still have an array but it now stores **Nodes** that contains the exact same information as **Entries**. Nodes can be **extended to TreeNodes**. A TreeNode is a red-black tree structure that stores really more information.

By inheritance, the **inner table can contain** both **Node** **(linked list ) and** **TreeNode** (red-black **tree**). Oracle decided to use both data structures with the following rules:

* If for a given index (bucket) in the inner table there are more than 8 nodes, the linked list is transformed into a red black tree
* If for a given index (bucket) in the inner table there are less than 6 nodes, the tree is transformed into a linked list.

