**Java interview preparation**

**Core Java vs Advanced Java**

According my knowledge there is no core and advanced java in java world... we have only j2se, j2ee and j2me.

J2SE covers - OOPS concept & implementation, Data types, Expressions (including Lambda), Control Flow (if, else, switch, for, while etc.), Classes, Objects, Inner Classes, Wrapper Classes, Packages, Enum Types, Generics, Strings, Numbers, Collection API, and JDBC API etc.

The fundamentals concepts of java which deals basically with OOPS concepts and their implementation in language can be called core java and the remaining can be called advanced java.

The **Core Java** comprises the Single Tier Architecture. It comprises util, lang, awt, io and net packages. It covers OOPS Concepts, Wrapper Classes, Special Operators, Data types, exception Handling, Collection API, and JDBC API. It means "stand -alone" java application.

**Advanced Java** is the next advanced level concept of Java programming. It basically uses two Tier Architecture i.e Client and Server. It covers the Swings, Socket Programming, AWT, Thread Concepts etc. It relates to specialization in domains such as web, networking, and data base handling. It is used for developing the web based application and enterprise application.

**J2SE vs J2EE vs J2ME**

**J2SE(Java Platform, Standard Edition)**

Also known as **Core Java**, this is the most basic and standard version of Java. It’s the purest form of Java, a basic foundation for all other editions.

It consists of a wide variety of general purpose API’s (like java.lang, java.util etc). J2SE is mainly used to create applications **for Desktop environment**. It consist all the basics of Java the language, variables, primitive data types, Arrays, Streams, Strings Java Database Connectivity(JDBC) and much more. This is the standard, from which all other editions came out. JVM is developed using J2SE.

**J2ME (Java Platform, Micro Edition)**

Mainly concentrated for the applications running on embedded systems, mobiles and small devices. Targeted to address Constraints - limited processing power, battery limitation, small display etc. They use web compression technologies to reduce network usage and hence avail cheap internet accessibility.

Old Nokia phones, which used Symbian OS, used this technology. Most of the apps, developed for the phones (prior to smartphones era), were built on J2ME platform only(the .jar apps on Nokia app store).

**J2EE(Java Platform, Enterprise Edition)**

The Enterprise version of Java has a much larger usage of Java, like development of web services, networking, server side scripting and other various web based applications. J2EE is a **community driven edition**, i.e. there is a lot of continuous contributions from industry experts, Java developers and other open source organizations.

J2EE uses many components of J2SE, as well as, has many new features of its own like Servlets, JavaBeans, Java Message Services, adding a whole new functionalities to the language. J2EE uses HTML, CSS, JavaScript etc., so as to create web pages and web services. It’s also one of the most widely accepted web development standard.

What distinguishes it from other languages (like .net, php etc.) is the versatility, compatibility and security features.

**Java Card:**

This edition was targeted, to run applets smoothly and securely on smart cards and similar technology. **Portability** and **security** was its main features.

**JavaFX:**

Another edition of Java technology which is now merged with J2SE 8. It is mainly used, to create rich GUI (Graphical User Interface) in Java apps. It replaces Swings (in J2SE), with itself as the standard GUI library. It is supported by both Desktop environment as well as web browsers.

**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).

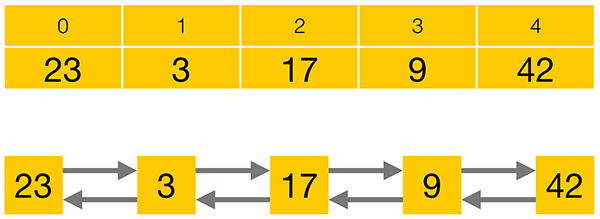
List list = Collections.synchronizedList(new LinkedList(...));

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**:

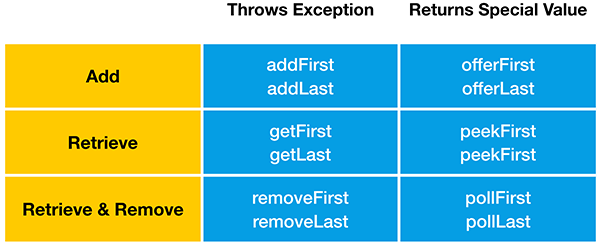


“**add**” will throw an Exception when the Queue is full, while “**offer**” will return false in this case. LinkedList, like most Queue implementations, has an unlimited capacity, so it will never be full. **ArrayBlockingQueue**, on the other hand, is a Queue implementation that has a limited capacity.

**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.

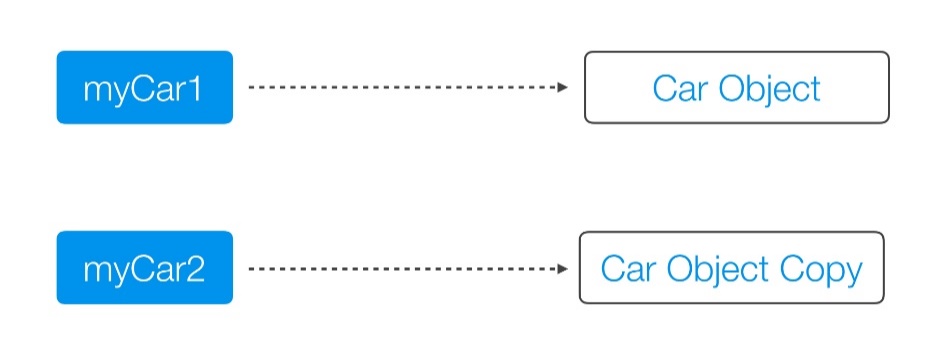
**Difference between shallow cloning and deep cloning of objects?**

**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;

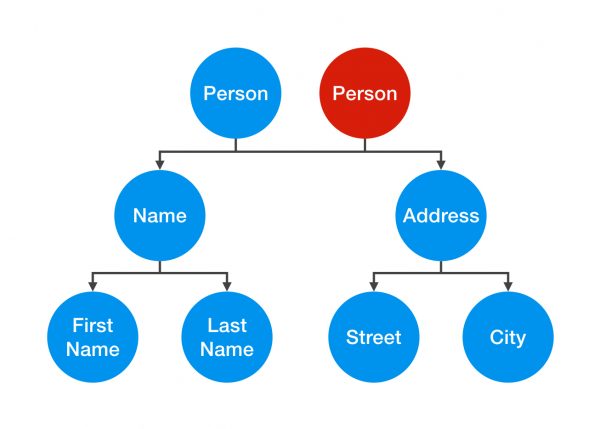
this.address = originalPerson.address;

}

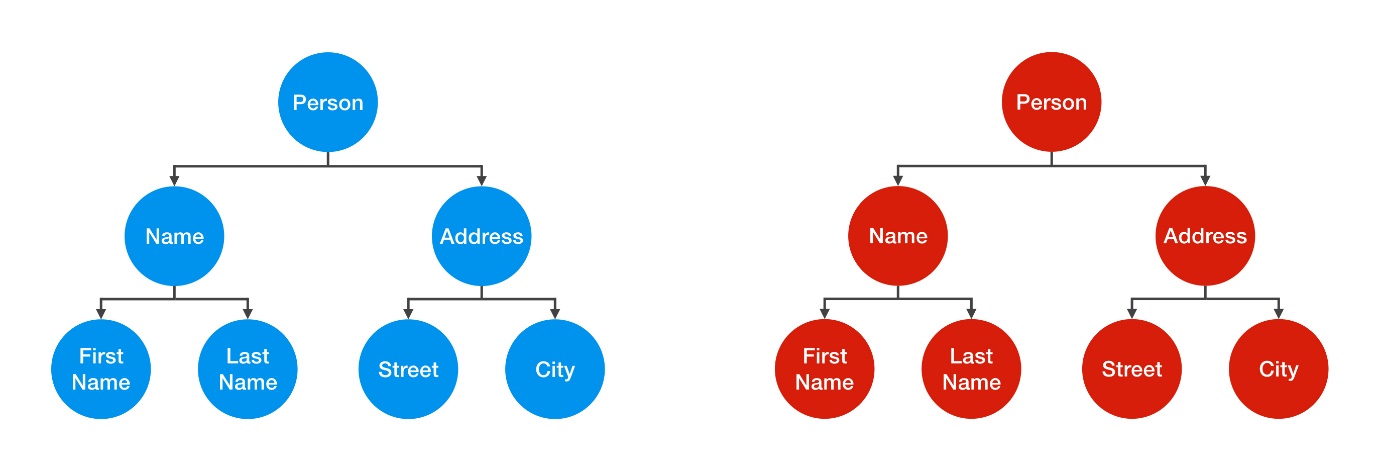
[…]

}

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 they 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;

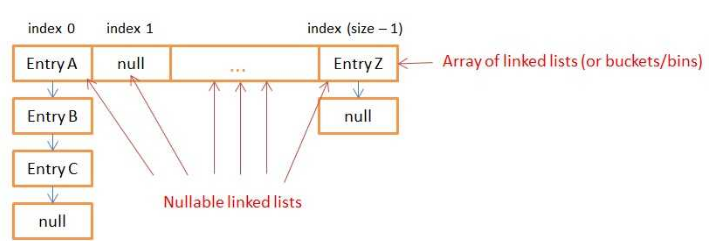
Entry<K,V> next;

final int hash;

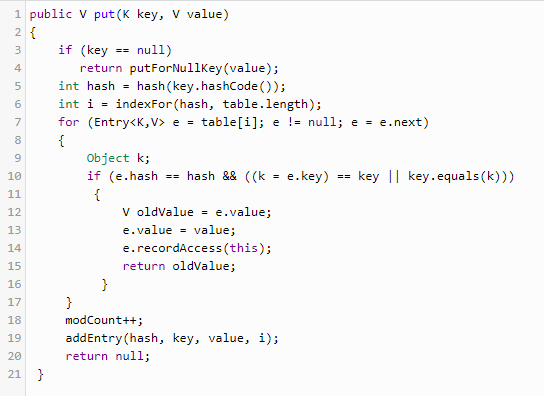
........

}

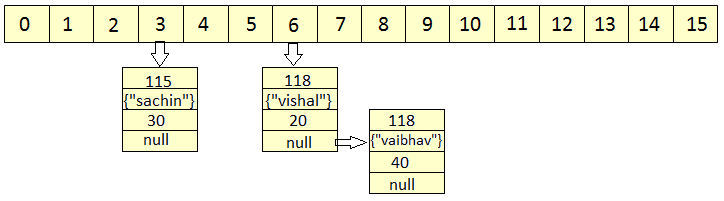
A HashMap stores data into multiple singly linked lists of entries (also called **buckets** or **bins**). All the lists are registered in an array of Entry (Entry<K,V>[] array) and the default capacity of this inner array is 16.



**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/>

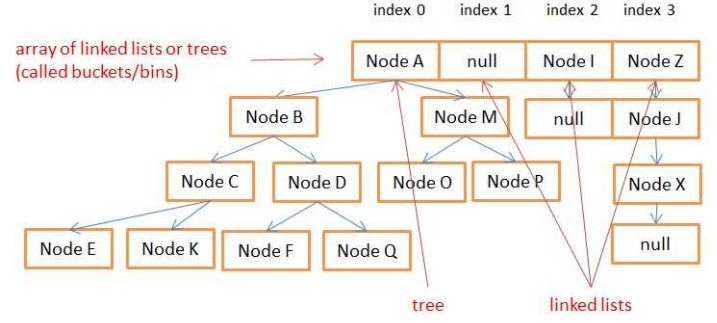
**Java 8 changes to HashMap:**

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.



**Use of Volatile Keyword in java:**

Using volatile is yet another **way of making class thread safe** ((like synchronized, atomic wrapper)). Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

The **values of volatile variable will never be cached** and all writes and reads will be done to and from the main memory.

Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in its own memory. The volatile keyword is used to say to the jvm "Warning, this variable may be modified in an other Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword **could be used on every kind of variable, either primitive or objects**.

**Atomic classes:**

Atomic classes **support lock-free thread-safe programming** on single variables.

Primitive variables can't be part of Synchronized block. Ex. Int, Float, Long, boolean etc. Use java.util.concurrent.atomic.AtomicInteger, AtomicLong etc. instead.

**Atomic operations** are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

**count++** is not an atomic operation. So by the time one threads read its value and increment it by one, other thread has read the older value leading to wrong result. To solve this issue, we will have to make sure that increment operation on count is atomic. We can do that using Synchronization, but Java 5 **java.util.concurrent.atomic** provides wrapper classes for int and long that can be used to achieve this atomic operation **without usage of Synchronization**.

Ex. AtomicInteger method **incrementAndGet()** atomically increments the current value by one.

AtomicInteger count = new AtomicInteger();

count.incrementAndGet();

Atomic classes are **not general purpose replacements** for java.lang.Integer and related classes. They **do not define** methods such as hashCode and compareTo. (Because atomic variables are expected to be **mutated**, they are **poor choices for hash table keys**).

Also atomic operation concurrency classes are assumed to be more efficient than synchronization which involves locking resources.

**Vs. Wrapper class:**

Concept of atomicity comes when something is mutable. We want the operation of modifying a field/variable (could be many steps, read -> update -> write) as atomic operation. So that in multithreaded scenario, there should not be any data corruption. java.util.concurrent.atomic package does it for us. Wrapper classes are immutable, we can't modify it, need to create a new instance.

**Functional interfaces in Java EE:**

Ex. java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator, java.util.concurrent.Callable - they have only one method declared in their interface definition.

These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**. And a popular way in which these are used is by creating Anonymous Inner classes using these interfaces.

**With Java 8** the same concept of **SAM interfaces is recreated** and are called Functional interfaces. These can be **represented using Lambda expressions**, Method reference and constructor references. **@FunctionalInterface** can be used for compiler level errors when the interface you have annotated is not a valid Functional Interface. It can also declare the abstract methods from the java.lang.Object class.

Interface can extend another interface and in case the Interface it is extending is functional and it doesn’t declare any new abstract methods then the new interface is also functional. But an interface can have one abstract method and any number of default methods and the interface would **still be called a functional interface**.

Ex.

@FunctionalInterface

**public** **interface** SimpleFuncInterface {

**public** **void** doWork();

**public** String toString();

**public** **boolean** equals(Object o);

}

Another example:

// Use Lambda operator or Arrow Token

// Method followed by Arrow Token and then the code that perform actions

Runnable runnable = () -> System.***out***.println("Welcome to Java 8 : Thread 1");

Runnable runnable2 = () -> {

//Multiliner statements - return is mandatory else can use single liner conditional if

System.***out***.println("Welcome to Java 8 : Thread 2");

}; //Ending with semicolon

**What is a serialVersionUID and why should I use it?**

SerialVersionUID is a unique identifier for each class, JVM uses it to compare the versions of the class ensuring that the same class was used during Serialization is loaded during Deserialization.

Specifying one gives more control, though JVM does generate one if you don't specify.

As per Java docs, "the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization".

**You must declare serialVersionUID because it give us more control**.

When a Serializable class object is serialized Java Runtime associates a serial version no.(called as serialVersionUID) with this serialized object. At the time when you deserialize this serialized object Java Runtime matches the serialVersionUID of serialized object with the serialVersionUID of the class. If both are equal then only it proceeds with the further process of deserialization else throws **InvalidClassException**.

So we conclude that to make Serialization/Deserialization process successful the serialVersionUID of serialized object must be equivalent to the serialVersionUID of the class. In case if programmer specifies the serialVersionUID value explicitly in the program then the same value will be associated with the serialized object and the class.

It is also possible that the environment where the object is serialized is using one JRE (ex: SUN JVM) and the environment where deserialzation happens is using Linux Jvm(zing). In such cases serialVersionUID associated with serialized object will be different than the serialVersionUID of class calculated at deserialzation environment. In turn deserialization will not be successful. So to avoid such situations/issues programmer must always specify serialVersionUID of Serializable class.

**Impact of not defining serialVersionUID in class:**

If we don’t define serialVersionUID in the class, and any modification is made in class, then we won’t be able to deSerialize our class because serialVersionUID generated by java compiler for modified class will be different from old serialized object. And deserialization process will end up throwing **java.io.InvalidClassException** (because of serialVersionUID mismatch).

If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing InvalidClassException - Simply we need to define **serialVersionUID** in class.

When we Deserialize class (class which has been **modified after Serialization** and also class **doesn’t declare SerialVersionUID**) **InvalidClassException** is thrown.

When we Deserialize class (class which has been modified after Serialization and also class declare SerialVersionUID) its gets DeSerialized successfully.

**Why to use Generics?**

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

The difference is that the inputs to **formal parameters** (to method declarations) are **values**, while the inputs to **type parameters** are **types**.

Generics are nothing but **parameterized types**. Generics helps us to create a single class, which can be useful to operate on multiple data types. A class, interface or a method that operates on a parameterized type is called generics class, interface or method. Generics adds type safety. Remember that generics only works on objects, not primitive types.

**Benefits over non-generic code:**

* **Stronger type checks at compile time.**  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* **Elimination of casts.**  
  The following code snippet without generics requires casting:

List list = new ArrayList();

list.add("hello");

String s = **(String)** list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* **Enabling programmers to implement generic algorithms:**

By using generics you can define an algorithm once, and you can apply it on any kind of datatype without any additional effort.

Consider the following method, printList:

public static void printList(List<Object> list) {

for (Object elem : list)

System.out.println(elem + " ");

System.out.println();

}

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList method, use List<?>:

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList to print a list of any type:

List<Integer> li = Arrays.asList(1, 2, 3);

List<String> ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);

The **unbounded wildcard type** is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. Unbounded wildcard is a useful for below scenarios:

* If you are writing a method that can be implemented using functionality provided in the **Object** class.
* When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, **Class<?>** is so often used because most of the methods in Class<T> do not depend on T.

**What are Generic Types & Generic Methods?**

A generic type is a generic class or interface that is parameterized over types.

**Non-generic** Box class

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

**Generic** Version of the Box Class

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

**Generic methods:**

Generic methods are methods that introduce their own type parameters. The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

public class Util {

**public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

boolean same = Util.compare(p1, p2);

**What are Bounded Type Parameters?**

Used when you want to restrict the types that can be used as type arguments in a parameterized type.

For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Ex.

public <U **extends Number**> void inspect(U u){

System.out.println("T: " + t.getClass().getName());

System.out.println("U: " + u.getClass().getName());

}

.

.

.

inspect("some text"); // **error: this is still String!**

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String.

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return **n.intValue()** % 2 == 0;

}

// ...

}

The isEven method invokes the intValue method defined in the Integer class through n.

A type parameter can also have multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }