# **ArrayList v/s Vector:**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| **ArrayList** is introduced in the original collection framework in **Java 1.2** version | **Vector** is a **legacy** class including Stack, Dictionary, HashTable & Properties and introduced in **Java 1.0** version |
| **ArrayList** methods are **non-synchronized** | All legacy collection classes are synchronized, thus **Vector**is **synchronized** **(i.e.; all methods of Vector class is synchronized**) |
| As **ArrayList** is **non-synchronized**, hence it **isn’t thread-safe**. So, programmer need to handle thread-safety while working in **multi-threaded** environment | As **Vectror** is **synchronized**, hence it is **thread-safe**. So, no need to worry while working in **multi-threaded**environment, as only one thread get chance to work at any given time |
| This is comparatively **faster** as it is **non-synchronized**, as threads doesn’t require to obtain lock before operating on ArrayList | **Performance-wise** vector is slower comparing with ArrayList due to synchronization, as threads need to wait for their chance to operate on Vector object |
| ArrayList **increases** its size by **50%** of current array, when its capacity exceeds | Vector **increases** its size by **100%** of current array, when its capacity exceeds |
| Only **Iterator** is allowed to iterate item/elements inside **ArrayList** | Both **Iterator & Enumeration** can be used to iterate item/elements inside **Vector** |
| **ArrayList** can be converted into **synchronized ArrayList**using **static utility methods of Collections** class  Collection.synchronizedList(arrayList); | No need to do that, as already **Vector is synchronized by default** |

## **When to use ArrayList ?**

* If performance is the factor while storing element/objects, then ArrayList is apt
* But definitely extra precautions need to be taken while working with multil-threaded environment
* Also, check how much extra space is required when List is full; if 50% of original size if required then ArrayList will fits the case perfectly

## **When to use Vector ?**

* If we aren’t concerned with performance, but element/objects need to be accessed in thread-safe manner, then Vector is good choice
* But performance will be a big hit, as every thread to need to wait to obtain lock before accessing vector element/objects
* Here, size increase in 2 times the original size; so if there are more number of items to be added then Vector will fits the bill perfectly

# ArrayList v/s LinkedList:

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| To store item/elements, ArrayList uses **dynamic array** or **dynamically re-sizing array**i.e.; internal data structure | To store items/elements, LinkedList uses **doubly linked list** i.e.; internal data structure |
| The initial capacity of ArrayList is **10** | LinkedList doesn’t have any initial capacity i.e.; just constructs **empty list of size 0** |
| When ArrayList exceeds its capacity, then its size increases by **50%** | No such thing required in LinkedList |
| When ArrayList exceeds its capacity, then internally new array is created with **50% more of the original size** and  **Old array data copied into new array** | No such overhead, as item/element is added to **end of LinkedList**  Due to this, **insertion is faster** in LinkedList comparing with ArrayList |
| Similarly, while **deleting** from the middle of ArrayList involves lot of **shifting work** | **Deletion** is much simpler in LinkedList, as previous and next links gets deleted and **new link is formed** |
| ArrayList internally uses array to store the items, so **retrieval** becomes faster as array works on **index-based** | LinkedList iterate over list to retrieve/get required item/element |
| Overall, **retrieval** is faster in ArrayList when comparing with LinkedList  In other words, if any application requires lot of **retrieval tasks** then ArrayList is the best suit | Overall, **insertion/removal** is faster in LinkedList when comparing with ArrayList  In other words, LinkedList is the best suit for an application involving lot of**insertion/deletion** tasks |
| There are no **memory overhead** in ArrayList as it holds only actual item/elements (data) | When compared with ArrayList, LinedkList has **more memory overhead** as it need to maintain addresses of previous and next node in addition to actual actual item/elements (data) |
| ArrayList can be **traversed** in only **one direction** while iterating over its item/elements | LinkedList has an API to **traverse** in **both directions** while iterating over its item/elements i.e; using **descendingIterator()**method |
| Elements of ArrayList stored in **consecutive** memory location | Elements of LinkedList stored in **random** memory location |

## **When to use ArrayList ?**

* When there are more number of **retrievals** like accessing employee records against employee code
* **Insertion** and **deletion** is very less (or very minimal) for an application
* **Reason:** when ArrayList capacity exceeds, then internally a new array with **50%** more than original size is created and **older** array data/items/elements are copied into **new** array
* It is better to avoid ArrayList, when there are more number of insertion/removal/deletion of an item/element from ArrayList as it involves lot of shifting work internally

## **When to use LinkedList ?**

* When there are more number of **insertion** like for example, whenever an aeroplane lands then its data need to be captured and stored into the list
* Also when item/element need to be **deleted** from list then LinkedList is the best fit, when comparing with ArrayList
* Don’t use LinkedList, when there are more number of retrieval as every items need to be **traversed** either from **beginning/end** to get the required item from list

# **List v/s Set:**

|  |  |
| --- | --- |
| **List** | **Set** |
| List stores elements according to **insertion order**  So, **insertion order is preserved** | Set stores elements in **random order**, as it uses hashing technique  **Insertion order isn’t preserved** |
| While iterating **List items**, elements will be retrieved as per **insertion order** | While iterating **Set items**, elements will be retrieved in **random order** |
| List allows duplicate elements | Set doesn’t allow duplicate elements i.e.; it stores only unique elements  **Note:** if same element is added again, there won’t be any **compile-time** or **runtime error**, just that **add()** method returns **false**; |
| Any number of **NULL** object is allowed to add to the List | **Maximum** of one **NULL** is allowed |

## **When to use List ?**

* If the business requirement **is to preserve insertion order** and
* adding **duplicate elements** is not a big concern
* then List is the good choice to **store group of elements**
* **Example:** it could be ArrayList or LinkedList or Vector, etc

## **When to use Set ?**

* If the business requirement is **to avoid storing duplicate elements**
* And storing only **unique elements**
* Where **insertion order isn’t** big factor while iterating items
* then Set is the good choice **to store group of elements**
* **Example:** it could be HashSet, etc

# **HashSet v/s LinkedHashSet:**

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| Uses **hashtable** to store element/objects  (actually HashMap instance) | Uses combination of **hashtable** + **LinkedList** to store element/objects |
| Doesn’t maintain insertion order  i.e.; while iterating through HashSet, we will get items in **random-order** | Since, it uses doubly-linked list to store elements, maintains **insertion-order** |
| This is introduced in the original collection framework in **Java 1.2** version | This is introduced in **Java 1.4** version |

## **When to use HashSet ?**

* HashSet stores **unique elements** using **hashing** technique
* So, **search operation** is faster
* So, if business requirement is to store **unique elements** for faster **search operation** or **more number of search operation** without concerning **insertion-order**
* Then, HashSet is the very apt choice

## **When to use LinkedHashSet ?**

* This is exactly same as that of **HashSet**, but underlying **data structure** to hold items is different
* It uses doubly-linked list which allows to hold items as per **insertion-order**
* So, if business requirement is to store **unique elements** for faster **search operation** or **more number of search operation** concerning/maintaining **insertion-order**
* Then, LinkedHashSet is the very apt choice which **maintains insertion-order**
* So while iterating through LinkedHashSet, we will get items as per insertion order (as against random in HashSet)

# **HashSet v/s LinkedHashSet v/s TreeSet:**

|  |  |  |
| --- | --- | --- |
| **HashSet** | **LinkedHashSet** | **TreeSet** |
| Uses **hashtable** to store element/objects where **duplicate** element/objects are **NOT** allowed | Uses combination of **(hashtable + LinkedList)** to store element/objects where **duplicate** element/objects are **NOT** allowed | Uses **balanced-tree** to store element/objects where **duplicate** element/objects are **NOT**allowed |
| **Insertion-order** is **NOT**maintained, as it uses **hashing technique** to store element/objects | **Insertion-order** is maintained, as it uses **doubly-linked list** to store element/objects | **Insertion-order** is **NOT** maintained, as element/objects are stored according to some **sorting-order** |
| HashSet doesn’t deal with **sorting-order**;  but it can be **converted** to TreeSet using inter-conversion constructor, which sorts element/objects in **sorting-order**  TreeSet ts = new TreeSet(hashSet); | LinkedHashSet doesn’t deal with **sorting-order**;  but it can be **converted** to TreeSet using inter-conversion constructor, which sorts element/objects in **sorting-order**  TreeSet ts = new TreeSet(linkedHashSet); | Element/objects stored in TreeSet are according to some **sorting-order;**  it could be either **default natural sorting-order** or programmer defined **customized sorting-order** |
| While iterating HashSet, we will get items in **random-order** | While iterating LinkedHashSet, we will get items as per **insertion-order** | While iterating TreeSet, we will get items in **sorted-order;**  either **natural-ordering** or **customized sorting-order** |
| This is introduced in original collection framework in **Java 1.2** version | This is introduced in **Java 1.4** version | This is also introduced in original collection framework in **Java 1.2** version |
| Allows **NULL insertion** but maximum of only one NULL value | Allows **NULL insertion** but maximum of only one NULL value | From **Java 1.7 version**, NULL is not allowed to insert;  Till **Java version 1.6**, only one NULL is allowed that too as 1st element |

# **HashMap v/s HashSet:**

|  |  |
| --- | --- |
| **HashMap** | **HashSet** |
| **HashMap** implements **Map** interface | **HashSet** implements **Set** interface |
| Used to store **key-value pairs** using **put** method  **Example:** hm.put(key, value); | Used to store **only unique objects** using **add** method  **Example:** hs.add(object); |
| HashMap **doesn’t allow duplicate keys** but **values can be duplicated** | HashSet **doesn’t allow duplicate** **objects** |
| HashMap allows **maximum of one null key** but **any number of NULL values** allowed | HashSet allows **maximum of one null object** to be added |
| HashMap internally uses **an array of Entry<K,V> objects** | HashSet internally uses **HashMap to store unique objects** |
| **Performance-wise**, HashMap is **faster** than HashSet | **Performance-wise**, HashSet is **slower** than HashMap |

## **When to use HashMap ?**

* HashMap stores **key-value pairs** which uses **hashing** technique to store **key-value pairs** where **methods are NOT synchronized**
* So, **search operation** is faster with **multiple threads access**
* So, if business requirement is to store **key-value pairs** for faster **search operation** or **more number of search operation**on the basis of keys; without concerning **concurrent access** of map
* Then, HashMap is the very apt choice

## **When to use HashSet ?**

* HashSet stores **unique elements** using **hashing** technique
* So, **search operation** is faster
* So, if business requirement is to store **unique elements** for faster **search operation** or **more number of search operation** without concerning **insertion order**
* Then, HashSet is the very apt choice

# **Enumeration v/s Iterator v/s ListIterator:**

* All 3 cursors are used to iterate over collection items
* but there are certain differences between each one of them

|  |  |  |
| --- | --- | --- |
| **Enumeration** | **Iterator** | **ListIterator** |
| This is part of **Legacy collection** introduced in **Java 1.0** version | This is part of **Collection framework** introduced in **Java 1.2** version | This is part of **Collection framework** introduced in **Java 1.2** version |
| Using **Enumeration interface**, we can enumerate only **legacy classes** like Hashtable or Vector or Properties | **Iterator interface** is applicable for **every collection classes** like ArrayList, HashSet or Hashtable | **ListIterator interface** is applicable only for **List objects** like ArrayList or LinkedList or Vector |
| We can enumerate legacy collection items only in **FORWARD direction** | Here, too we can iterate through collection items only in **FORWARD direction** | But with ListIterator, we can iterate through list items either in **FORWARD** or **BACKWARD directions** |
| That is, it is **unidirectional** or single directional cursor | That is, it is **unidirectional** or single directional cursor | That is, it is **bi-directional**cursor |
| Using **Enumeration interface**, we can enumerate **to read or get element/object** from legacy collection | Using **Iterator interface**, we can **read** as well as **remove**collection items, while iterating | **Addition or replacement** of new objects is possible alongside **read and remove**operation in **ListIterator interface** |
| To get an **Enumeration object**, we can use **elements()**method of any **legacy collection** class  **For example,**  Vector v = new Vector();  Enumeration e = v.elements(); | To get an **Iterator object**, we can use **iterator()** method of any **collection** class  **For example,**  Iterator itr = col.iterator();  Where col = any collection class | To get a **ListIterator object**, we can use **listIterator()**method of any **List** classes  **For example,**  ListIterator ltr = list.listIterator();  Where list = any List objects |
| Enumeration interface has **2 important methods** to enumerate through legacy collection objects  **boolean hasMoreElements();**  **Object nextElement();** | Iterator interface has **3 important methods** to iterate through any collection objects  **boolean hasNext();**  **Object next();**  **void remove();** | ListIterator interface has **9 important methods** to iterate through any List objects  Read here, for details of [**ListIterator methods**](http://www.benchresources.net/listiterator-interface-in-java/) |

### **Best practice:**

#### **Enumeration interface:**

* Use this cursor only with legacy collection, to work with thread-safe environment

#### **Iterator interface:**

* This is very popular among 3 cursors, as it is applicable for any collection class

#### **ListIterator interface:**

* Again, this is applicable only for List objects.
* Use this cursor, to benefit from iterating through List items in both directions
* i.e.; both FORWARD & BACKWARD directions

# **Comparable v/s Comparator:**

|  |  |
| --- | --- |
| **Comparable interface** | **Comparator interface** |
| Present in **java.lang** package | Present in **java.util** package |
| Defines only one important method i.e.;  public int **compareTo**(Object obj); | Defines 2 method i.e.;  public int **compare**(Object obj1, Object obj2);  public boolean **equals**(Object object); |
| It is basically used for **default natural sorting order**[DNSO] | This is preferred for **customized sorting order** [CSO] |
| This interface need to be **implemented** in the **same class** for which **sorting** is required | **Separate class** is required to **implement** Comparator interface |
| Elements of List can be sorted using comparable interface using sort(); method of Collections class, as shown below  **Example:** Collection.sort(listItems); | Elements of List can be sorted using comparator interface using sort(); method of Collections class, as shown below  **Example:** Collection.sort(listItems, comparator); |
| String & wrapper classes’ like Integer, Double, etc implement **comparable** interface | There are very few classes’ which implements **Comparator** interface |

### **Example on Comparable and Comparator interface:**

1. [**Comparable interface**](http://www.benchresources.net/comparable-interface-in-java/)
2. [**Comparator interface**](http://www.benchresources.net/comparator-interface-in-java/)

|  |  |
| --- | --- |
| **Comparable interface** | **Comparator interface** |
| Present in **java.lang** package | Present in **java.util** package |
| Defines only one important method i.e.;  public int **compareTo**(Object obj); | Defines 2 method i.e.;  public int **compare**(Object obj1, Object obj2);  public boolean **equals**(Object object); |
| It is basically used for **default natural sorting order** [DNSO] | This is preferred for **customized sorting order** [CSO] |
| This interface need to be **implemented** in the **same class** for which **sorting** is required | **Separate class** is required to **implement** Comparator interface |
| Elements of List can be sorted using comparable interface  **Example:** Collection.sort(listItems); | Elements of List can be sorted using comparator interface  **Example:** Collection.sort(listItems, comparator); |
| String & wrapper classes’ like Integer, Double, etc implement **comparable interface** | There are very few classes’ which implements **Comparator interface** |

# **HashMap v/s LinkedHashMap:**

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| Uses **hash table** to store map entries (i.e.; key-value pairs) | Uses combination of **hash table + LinkedList** to store map entries (i.e.; key-value pairs) |
| Doesn’t maintain **insertion-order** i.e.; while iterating through HashMap, we will get map entries in **random-order** | Since, it uses doubly-linked list to store map entries (i.e.; key-value pairs), maintains **insertion-order** |
| This is introduced in the original collection framework in **Java 1.2** version | This is introduced in **Java 1.4** version |

### **When to use HashMap ?**

* HashMap stores **key-value pairs** which uses **hashing** technique to store key-value pairs
* So, **search operation** is faster
* So, if business requirement is to store **key-value pairs** for faster **search operation** or **more number of search operation**on the baisis of keys; without concerning **insertion-order**
* Then, HashMap is the very apt choice

### **When to use LinkedHashMap ?**

* This is exactly same as that of **HashMap**, but underlying **data structure** to hold **key-value pairs** is different
* It uses doubly-linked list which allows to hold **key-value pairs** as per **insertion-order**
* So, if business requirement is to store **key-value pairs** for faster **search operation** or **more number of search operation**concerning/maintaining **insertion-order**
* Then, LinkedHashSet is the very apt choice which maintains insertion-order
* So while iterating through LinkedHashSet, we will get map entry (**Key-Value pairs)** as per insertion-order, as against random-order in HashMap

## **HashMap v/s LinkedHashMap v/s TreeMap :**

|  |  |  |
| --- | --- | --- |
| **HashMap** | **LinkedHashMap** | **TreeMap** |
| Uses **hash table** to store **key-value pairs** (i.e.; map entries) where **duplicate keys** are **NOT** allowed | Uses combination of **(hash table + LinkedList)** to store **key-value pairs**(i.e.; map entries) where **duplicate keys** are **NOT** allowed | Uses **Red-Black tree** to store **key-value pairs** (i.e.; map entries) where **duplicate keys** are **NOT** allowed |
| **Insertion-order** is **NOT** maintained, as it uses **hashing technique** to store **key-value pairs** (i.e.; map entries) | **Insertion-order** is maintained, as it uses **doubly-linked list** to store **key-value pairs** (i.e.; map entries) | **Insertion-order** is **NOT** maintained, as **key-value pairs** (i.e.; map entries) are stored according to some **sorting-order** |
| HashMap doesn’t deal with **sorting-order**; but it can be **converted** to TreeMap to store **key-value pairs** (i.e.; map entries) in some sorting-order  TreeMap ts = new TreeMap(hashMap); | LinkedHashMap doesn’t deal with **sorting-order**; but it can be **converted** to TreeMap to store **key-value pairs** (i.e.; map entries) in some sorting-order  TreeMap ts = new TreeMap(linkedHashMap); | Keys in TreeMap are sorted, according to some **sorting-order;** it could be either **default natural sorting-order**or programmer defined **customized sorting-order** |
| While iterating HashMap, we will get items in **random-order** | While iterating LinkedHashMap, we will get items as per **insertion-order** | While iterating TreeMap, we will get items in **sorted-order;**  either **natural-ordering** or **customized sorting-order** |
| This is introduced in original collection framework in **Java 1.2** version | This is introduced in **Java 1.4** version | This is also introduced in original collection framework in **Java 1.2** version |
| **Key:** Allows **NULL insertion** but maximum of only one NULL value  **Value:** No upper limit for NULL values against any unique key | **Key:** Allows **NULL insertion** but maximum of only one NULL value  **Value:** No upper limit for NULL values against any unique key | **Key:** From **Java 1.7 version**, NULL is not allowed to insert; But with Java version less than **1.7**, maximum of 1 NULL allowed as 1stelement**Value:** No upper limit for NULL values against any unique key |

# **HashMap v/s HashSet:**

|  |  |
| --- | --- |
| **HashMap** | **HashSet** |
| **HashMap** implements **Map** interface | **HashSet** implements **Set** interface |
| Used to store **key-value pairs** using **put** method  **Example:** hm.put(key, value); | Used to store **only unique objects** using **add** method  **Example:** hs.add(object); |
| HashMap **doesn’t allow duplicate keys** but **values can be duplicated** | HashSet **doesn’t allow duplicate** **objects** |
| HashMap allows **maximum of one null key** but **any number of NULL values** allowed | HashSet allows **maximum of one null object** to be added |
| HashMap internally uses **an array of Entry<K,V> objects** | HashSet internally uses **HashMap to store unique objects** |
| **Performance-wise**, HashMap is **faster** than HashSet | **Performance-wise**, HashSet is **slower** than HashMap |

## **When to use HashMap ?**

* HashMap stores **key-value pairs** which uses **hashing** technique to store **key-value pairs** where **methods are NOT synchronized**
* So, **search operation** is faster with **multiple threads access**
* So, if business requirement is to store **key-value pairs** for faster **search operation** or **more number of search operation**on the basis of keys; without concerning **concurrent access** of map
* Then, HashMap is the very apt choice

## **When to use HashSet ?**

* HashSet stores **unique elements** using **hashing** technique
* So, **search operation** is faster
* So, if business requirement is to store **unique elements** for faster **search operation** or **more number of search operation** without concerning **insertion order**
* Then, HashSet is the very apt choice

# **HashMap v/s Hashtable:**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| HashMap is introduced in collection framework in **Java 1.2**version | Hashtable is a **legacy** class and introduced in **Java 1.0**version |
| HashMap is **NOT synchronized** | Hashtable is **synchronized** |
| All methods of HashMap is NOT synchronized i.e.; it is **not thread-safe** | All methods of HashMap is synchronized i.e.; **thread-safe** |
| **Multiple threads** are allowed to **access** | **Only one thread** is allowed access; other threads has to **wait** to get access, after obtaining lock/monitor |
| **Performance-wise**, this is relatively **high** comparing with Hashtable, as there is no wait time | **Performance-wise**, this is relatively **slow** due synchronized methods as there is only **one thread allowed to access**, at any given point of time |
| **NULL insertion** allowed for both **keys** and **values** | **NULL insertion** is not **allowed** for both keys and **values** |
| Maximum of **one NULL key** and there is **no upper limit** for **values** | Simply, **not allowed** for both **keys & values** |

**Note:** both uses **hash table** data structure to store **key-value** pairs

### **When to use HashMap ?**

* HashMap stores **key-value pairs** which uses **hashing** technique to store **key-value pairs** where **methods are NOT synchronized**
* So, **search operation** is faster with **multiple threads access**
* So, if business requirement is to store **key-value pairs** for faster **search operation** or **more number of search operation**on the basis of keys; without concerning **concurrent access** of map
* Then, HashMap is the very apt choice

### **When to use Java Hashtable ?**

* This is exactly same as that of **HashMap**, but every **methods** is **synchronized**
* **Performance-wise** is relatively slower than comparing HashMap
* So, if business requirement is to store **key-value pairs** for faster **search operation** with synchronized access
* Then, Java Hashtable is preferred choice over HashMap

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| Uses **hash table** to store **key-value pairs** (i.e.; map entries) where **duplicate keys** are **NOT** allowed | Uses **Red-Black tree** to store **key-value pairs** (i.e.; map entries) where **duplicate keys** are **NOT** allowed |
| **Insertion-order** is **NOT** maintained, as it uses **hashing technique** to store **key-value pairs** (i.e.; map entries) | **Insertion-order** is **NOT** maintained, as **key-value pairs** (i.e.; map entries) are stored according to some **sorting-order** |
| HashMap doesn’t deal with **sorting-order**; but it can be **converted** to TreeMap to store **key-value pairs** (i.e.; map entries) in some sorting order  TreeMap ts = new TreeMap(hashMap); | Keys in TreeMap are sorted, according to some **sorting-order;** it could be either **default natural sorting-order** or programmer defined **customized sorting-order** |
| While iterating HashMap, we will get items in **random-order** | While iterating TreeMap, we will get items in **sorted-order;**  either **natural-ordering** or **customized sorting-order** |
| This is introduced in original collection framework in **Java 1.2** version | This is also introduced in original collection framework in **Java 1.2** version |
| **Key:** Allows **NULL insertion** but maximum of only one NULL value  **Value:** No upper limit for NULL values against any unique key | **Key:** From **Java 1.7 version**, NULL is not allowed to insert; But with Java version than **1.6** or less, only as 1st element allowed (for keys)**Value:** No upper limit for NULL values against any unique key |

|  |  |
| --- | --- |
| **HashMap** | **ConcurrentHashMap** |
| HashMap is not synchronized | ConcurrentHashMap is synchronized |
| In multi-threaded environment, HashMap is **faster** than ConcurrentHashMap as **multiple threads can operate**  Hence, **performance** is **high** as there is no need to acquire lock | As it is synchronized, **lock need** to be **acquired** before operating, although for **certain portion of the Map**  Hence**, performance** is relatively **low** when comparing with HashMap |
| **NULL insertion** is possible for key but **maximum of one null key** and any number of null values against any key | **NULL insertion** isn’t allowed for both **keys and values** |
| While one thread **iterating** HashMap items, if any other thread tries to **modify** Map items then **ConcurrentModificationException** is thrown | While one thread **iterating** ConcurrentHashMap items, other thread are happily can **modify** Map items  And it **never** throws **ConcurrentModificationException** |
| That’s it is **fail-fast iterator** | That’s it is **fail-safe iterator** |
| This is introduced in original collection framework in **Java 1.2** version | This is introduced in **Java 1.5** version |
| We can **convert** this Map item into **synchronized** map by using Collections class utility method  But still, **only one thread** is allowed to operate on Map object | There is **no such need** here, as it is already **thread-safe** and **multiple threads** can operate after **acquiring bucket-level** or segment-level locking strategies |

# Arrays v/s ArrayList:

|  |  |
| --- | --- |
| **Arrays** | **ArrayList** |
| **Arrays** is fixed in length  For **example**, int iArr = new int[7]; | **ArrayList** uses dynamic resizable/grow-able arrayFor **example**, ArrayList al = new ArrayList(); |
| It allows to **store primitive types & Objects** | It allows to **store only Object** whereas primitive types like int, float, double, etc **aren’t allowed**But its **equivalent wrapper Object** **types** like Integer, Float, Double, etc are allowed |
| While **adding** elements to Array, **type is bounded** i.e.; it allows to store element of any specific data-type or specific class  Trying to **add another data-type**, other than declared data-type results in throwing **ArrayStoreException** at runtime | Using **Generics** while declaring ArrayList makes it is **type-bounded** i.e.; if ArrayList is declared to accept only String or any specific class then adding any other type results in throwing **compile-time error** |
| **Storing** elements inside **Array is easy**, as simple **assignment operator** is enough  For example, intArr[0] = 10; | For **adding** element to ArrayList, use **add()** or **addAll()** methods of **java.util.Collection**interface |
| For Array, **length** variable provides the **length of an Array** | For ArrayList, **size()** method of java.util.Collection interface can be used to determine size of an **ArrayList** |
| For **Array iteration**, use following options   1. for-loop 2. enhanced for-loop | For [**ArrayList iteration**](http://www.benchresources.net/various-ways-to-iterate-through-arraylist-in-java/)**, use following** options   1. for-loop 2. enhanced for-loop 3. Iterator 4. ListIterator 5. forEach from Java 8 |
| **Performance-wise**, it always **stays constant** over time | **add()** & **get()** operations nearly provide same performance as that of ArrayBut with **modify**operation like **deletion** **will yield poor performance** because it involves lot of **shifting**With **capacity reaching maximum** **will result in again poor performance** **as it involves copying data** from **old array** **into new array** |
| **Example:** Refer [**Arrays**](http://www.benchresources.net/arrays-class-in-java/) **for details** | **Example:** Refer [**ArrayList**](http://www.benchresources.net/arraylist-class-in-java/) **for details** |

**What is fail-safe and fail-fast Iterator in Java ?**

|  |  |
| --- | --- |
| **Fail-fast** | **Fail-safe** |
| While iterating collection items if any modification is done, then **ConcurrentModificationException** is thrown  This is said to be **fail-fast** | While iterating collection items if any modification is done and if **ConcurrentModificationException** is never thrown, then it is said to be **fail-safe** |
| Generally, Collection classes introduced in **Java 1.2 version** like ArrayList or HashSet falls under this category | New concurrent classes introduced in **Java 1.5 version** is fail-safe and never throws ConcurrentModificationException |
| Here, there is **no concept of cloned copy**. Hence, both iteration & modification happening in the same original copy leading to throwing of ConcurrentModificationException | This is because, modification happens in a **separate cloned copy & later JVM merges** both original with cloned copies |
| **Example:** ArrayList, LinkedList, HashSet, TreeSet | **Example:** CopyOnWriteArrayList, CopyOnWriteArraySet, ConcurrentHashMap |
| Above listed classes comes from **java.util** package | Above listed classes comes from **java.util.concurrent** package |

|  |  |
| --- | --- |
| **CopyOnWriteArrayList** | **ArrayList** |
| CopyOnWriteArrayList is **synchronized** or newly introduced thread-safe class | ArrayList is **not** synchronized |
| For every **update** operation, a **new separate cloned copy** is created and there is **memory** & **merging overhead for JVM**  Hence**, performance** is relatively **low** when comparing with **ArrayList** | In multi-threaded environment, ArrayList is **faster** than **CopyOnWriteArrayList** as multiple threads **can operate**  Hence, **performance** is **high** as there is no need to acquire lock |
| While one thread **iterating CopyOnWriteArrayList** items, other threads happily can **modify**, as it **works** on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** ArrayList items, if any other thread tries to **modify** same ArrayList object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| **Iterator of CopyOnWriteArrayList** can perform **read operation** safely; while iterating through **COWAL** items  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **ArrayList** can perform both **read** and **remove** operations; while iterating through AL items |

|  |  |
| --- | --- |
| **Concurrent Collection** | **Synchronized Collection** |
| **Concurrent Collection** are newly introduced **thread-safe** class (i.e.; synchronized) | This is **thread-safe** version of Collection class |
| **Multiple threads** are allowed to operate on **Concurrent Collection**, as it works on separate **cloned copy** for **update/modify** operations | Only **one thread** is allowed to operate on synchronized collection, by **locking over complete list** object |
| While one thread **iterating** **Concurrent Collection** object, other threads happily can **modify**, as it works on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** on synchronized collection object, if any other threads tries to **modify** same object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| There is **no such restriction** while iterating on **Concurrent Collection**  We can safely iterate **outside synchronized block** | While **iterating synchronized Collection**, make sure to **iterate inside synchronized block**;  Otherwise we may face **non-deterministic behavior** |
| **Iterator of Concurrent Collection** can perform **read operation** safely; while iterating  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **synchronized collection** can perform both **read** and **remove** operations; whil |

# **ArrayList v/s ArrayList<T>:**

|  |  |
| --- | --- |
| **ArrayList** | **ArrayList<T>** |
| This is **non-generics** version of ArrayList | This is **generics version** of ArrayList with type-parameter T |
| In this non-generics version, ArrayList **allows to add any type of Objects** like String, Integer, references-types, etc. | But Generics version of ArrayList **allows to add specific type of objects only**  Like, if type-parameter T is replaced by String then only String-type of Objects are allowed to add to ArrayList |
| Basically, it **doesn’t** assures **type-safety** as any type of Objects can be added to ArrayList | It assures **type-safety,** as it allows to store same type of Objects only |
| During iteration of ArrayList, compulsorily **explicit type-casting** needs to be done even if ArrayList stores same type of Objects | In Generics version of ArrayList, **no explicit type-casting** is required  **Reason:** Generics ArrayList stores same type of Objects only, therefore **type-casting isn’t required** at the time of iteration or getting Objects |

# **ClassNotFoundException v/s** **NoClassDefFoundError**

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| This is generally occurs, when required .class is missing when program encounters class load statement such as,   * Class.forName(“class.name”); * ClassLoader.loadClass(“class.name”); * ClassLoader.findSystemClass(“class.name”);   **Reason:** required file **missing** in the class-path due to execution of program **without updating JAR file** at runtime | This is very much similar but the difference is required .class file is available during compile-time & missing at runtime  **Possible Reason:**   * It is **deleted** after compilation or * there could be **version** mismatch |
| Fully qualified class name is **java.lang.ClassNotFoundException** | Fully qualified class name is **java.lang.NoClassDefFoundError** |
| It falls under the category of Exception i.e.; direct sub-class of **java.lang.Exception** | It falls under the category of Error i.e.; sub-class of **java.lang.Error** through **java.lang.LinkageError** |
| It is a **checked exception**, therefore it needs to be **handled**, whenever class loading statement is encountered as stated in point no.1 | All errors come under **unchecked exception**category, therefore **NoClassDefFoundError** is also unchecked exception |
| As it is **checked exception**, programmer can provide handling code either using **try-catch** block or can declare **throws clause**  Therefore, it is **recoverable** | Errors are thrown by **Java Runtime system** during program execution  Therefore, it is **non-recoverable** |

|  |  |
| --- | --- |
| [**try-catch block**](http://www.benchresources.net/try-catch-block-in-java-exception-handling/) | [**throws keyword**](http://www.benchresources.net/throws-keyword-in-java-exception-handling/) |
| Using try-catch block, we can handle exception surrounding code that might raise an exception | Whereas using throws keyword, we can simply declare exception that might raise from that method |
| Caught exception in the catch block can be re-thrown after some alteration | There is no such flexibility, as its directly throws exception |
| try-catch block ensures graceful termination for that particular method  **Except** one scenario when catch block throws exception | Doesn’t guarantee graceful termination  In most cases, throws declaration leads to abnormal termination |

|  |  |
| --- | --- |
| **throw clause/keyword** | **throws clause/keyword** |
| **throw keyword** is used to throw exception explicitly | **throws** keyword is used to declare exception to delegate/indicate exception handling responsibility to caller-method |
| throw keyword is always followed by instance of **Throwable type** or **exception type** | throws keyword is always followed by exception list (with comma separating them) |
| throw keyword is used **within method** i.e.; to throw exception from try-catch block enclosed within method | throws keyword is used **next** to method signature |
| **Syntax: throw** instanceOfExceptionType;  [?](http://www.benchresources.net/difference-between-throw-vs-throws-keyword-in-java-exception-handling/#) | **Syntax:**  [?](http://www.benchresources.net/difference-between-throw-vs-throws-keyword-in-java-exception-handling/#)  access-modifier  **return**-type method-name()  **throws** exception-list; |
| Maximum of **only one exception** can be thrown using **throw keyword**  Thrown exception can be **checked exception** or **unchecked exception** or **user-defined exception** | **Any number of exception** can be declared (to be thrown) using **throws keyword**  But they are all separated by **comma *(,)*** |

# **Serializable v/s Externalizable:**

|  |  |
| --- | --- |
| **Serializable** | **Externalizable** |
| **Serializable** is a marker interface which **doesn’t contain** any **methods** and JVM provides special ability during serialization process | **Externalizable** is a sub-interface of Serializable interface and **contains 2 methods** viz.;   1. readExternal(); 2. writeExternal(); |
| During Serialization process, **all member variables** of an object is serialized, even if some of the variables aren’t required to be serialized | But in Externalization, **programmer** has to provide **serialization logic** |
| That’s why, it is referred as **default serialization** | This is referred as **custom serialization**, as programmer has to write/code custom logic for serialization to happen |
| From above stated points, it is clear that **JVM** takes complete control over serialization process | **Programmer** has complete control over serialization process to write custom logic for required variables to be serialized |
| **Performance-wise**, Serializable is relatively **low** as**complete object** need to be serialized, even if we require only partial object | **Performance** is **high** in extenalizable, as programmer design what all **required variable** need to be serialized |
| **Doesn’t** require any **public no-argument constructor** for serializable | **Public no-argument constructor is very must in externalizable**  Otherwise **InvalidClassException** is thrown  This is mainly required during **readExternal();** method;  i.e.; while restoring object back to heap memory from file storage |
| For variable that needn’t to be serialized use **transient**modifier but still its default value is stored into file  Transient modifier play a very important role in serializable | Variable with **transient** modifier not required; as programmer can write/code **custom logic** to ignore those variables which is not required  So, transient modifier doesn’t play any important role in externaizable |
| This is the best suit; when **whole/complete** object required to be serialized to file storage | This is the best suit; when **partial** object or **few of the member variables** of an object need to be serialized to file storage |