1. **What are Collection related features in Java 8?**

Java 8 has brought major changes in the Collection API. Some of the changes are:

* 1. [Java Stream API](http://www.journaldev.com/2774/java-8-stream) for collection classes for supporting sequential as well as parallel processing
  2. [Iterable interface is extended with forEach()](http://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach) default method that we can use to iterate over a collection. It is very helpful when used with [lambda expressions](http://www.journaldev.com/2763/java-8-functional-interfaces) because it’s argument Consumer is a [function interface](http://www.journaldev.com/2763/java-8-functional-interfaces).
  3. Miscellaneous Collection API improvements such as forEachRemaining(Consumer action)method in Iterator interface, Map replaceAll(), compute(), merge() methods.

1. **What is Java Collections Framework? List out some benefits of Collections framework?**

Collections are used in every programming language and initial java release contained few classes for collections: **Vector**, **Stack**, **Hashtable**, **Array**. But looking at the larger scope and usage, Java 1.2 came up with Collections Framework that group all the collections interfaces, implementations and algorithms.  
Java Collections have come through a long way with usage of Generics and Concurrent Collection classes for thread-safe operations. It also includes blocking interfaces and their implementations in java concurrent package.  
Some of the benefits of collections framework are;

* 1. Reduced development effort by using core collection classes rather than implementing our own collection classes.
  2. Code quality is enhanced with the use of well tested collections framework classes.
  3. Reduced effort for code maintenance by using collection classes shipped with JDK.
  4. Reusability and Interoperability

1. **What is the benefit of Generics in Collections Framework?**

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error.  
This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don’t need to use casting and *instanceof* operator. I would highly recommend to go through [**Java Generic Tutorial**](http://www.journaldev.com/1663/java-generics-example-method-class-interface) to understand generics in a better way.

1. **What are the basic interfaces of Java Collections Framework?**

[Collection](http://www.journaldev.com/1260/collections-in-java-tutorial#collection-interface) is 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.

[Set](http://www.journaldev.com/1260/collections-in-java-tutorial#set-interface) is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

[List](http://www.journaldev.com/1260/collections-in-java-tutorial#list-interface) is an ordered collection and can contain duplicate elements. You can access any element from it’s index. List is more like array with dynamic length.

A [Map](http://www.journaldev.com/1260/collections-in-java-tutorial#map-interface) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

Some other interfaces are [Queue](http://www.journaldev.com/1260/collections-in-java-tutorial#queue-interface), [Dequeue](http://www.journaldev.com/1260/collections-in-java-tutorial#dequeue-interface), [Iterator](http://www.journaldev.com/1260/collections-in-java-tutorial#iterator-interface), [SortedSet](http://www.journaldev.com/1260/collections-in-java-tutorial#sortedset-interface), [SortedMap](http://www.journaldev.com/1260/collections-in-java-tutorial#sortedmap-interface) and [ListIterator](http://www.journaldev.com/1260/collections-in-java-tutorial#listiterator-interface).

1. **Why Collection doesn’t extend Cloneable and Serializable interfaces?**

Collection interface specifies group of Objects known as elements. How the elements are maintained is left up to the concrete implementations of Collection. For example, some Collection implementations like List allow duplicate elements whereas other implementations like Set don’t.  
A lot of the Collection implementations have a public clone method. However, it does’t really make sense to include it in all implementations of Collection. This is because Collection is an abstract representation. What matters is the implementation.  
The semantics and the implications of either cloning or serializing come into play when dealing with the actual implementation; so concrete implementation should decide how it should be cloned or serialized, or even if it can be cloned or serialized.  
So mandating cloning and serialization in all implementations is actually less flexible and more restrictive. The specific implementation should make the decision as to whether it can be cloned or serialized.

1. **Why Map interface doesn’t extend Collection interface?**

Although Map interface and it’s implementations are part of Collections Framework, Map are not collections and collections are not Map. Hence it doesn’t make sense for Map to extend Collection or vice versa.  
If Map extends Collection interface, then where are the elements? Map contains key-value pairs and it provides methods to retrieve list of Keys or values as Collection but it doesn’t fit into the “group of elements” paradigm.

1. **What is an Iterator?**

Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using *iterator()* method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Java Collection iterator provides a generic way for traversal through the elements of a collection and implements [**Iterator Design Pattern**](http://www.journaldev.com/1716/iterator-design-pattern-java).

1. **What is difference between Enumeration and Iterator interface?**

Enumeration is twice as fast as Iterator and uses very less memory. Enumeration is very basic and fits to basic needs. But Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it.  
Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make it’s functionality clear.

1. **Why there is not method like Iterator.add() to add elements to the collection?**

The semantics are unclear, given that the contract for Iterator makes no guarantees about the order of iteration. Note, however, that ListIterator does provide an add operation, as it does guarantee the order of the iteration.

1. **Why Iterator don’t have a method to get next element directly without moving the cursor?**

It can be implemented on top of current Iterator interface but since it’s use will be rare, it doesn’t make sense to include it in the interface that everyone has to implement.

1. **What is different between Iterator and ListIterator?**
   1. We can use Iterator to traverse Set and List collections whereas ListIterator can be used with Lists only.
   2. Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both the directions.
   3. ListIterator inherits from Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.
2. **What are different ways to iterate over a list?**

We can iterate over a list in two different ways – using iterator and using for-each loop.

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

//using for-each loop

for(String obj : strList){

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext()){

String obj = it.next();

System.out.println(obj);

}

Using iterator is more thread-safe because it makes sure that if underlying list elements are modified, it will throw ConcurrentModificationException.

1. **What do you understand by iterator fail-fast property?**

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

1. **What is difference between fail-fast and fail-safe?**

Iterator fail-safe property work with the clone of underlying collection, hence it’s not affected by any modification in the collection. By design, all the collection classes in java.util package are fail-fast whereas collection classes in java.util.concurrent are fail-safe.  
Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException.  
Check this post for [CopyOnWriteArrayList Example](http://www.journaldev.com/1289/copyonwritearraylist-java).

1. **How to avoid ConcurrentModificationException while iterating a collection?**

We can use concurrent collection classes to avoid ConcurrentModificationException while iterating over a collection, for example CopyOnWriteArrayList instead of ArrayList.  
Check this post for [ConcurrentHashMap Example](http://www.journaldev.com/122/java-concurrenthashmap-example-iterator).

1. **Why there are no concrete implementations of Iterator interface?**

Iterator interface declare methods for iterating a collection but it’s implementation is responsibility of the Collection implementation classes. Every collection class that returns an iterator for traversing has it’s own Iterator implementation nested class.  
This allows collection classes to chose whether iterator is fail-fast or fail-safe. For example ArrayList iterator is fail-fast whereas CopyOnWriteArrayList iterator is fail-safe.

1. **What is UnsupportedOperationException?**

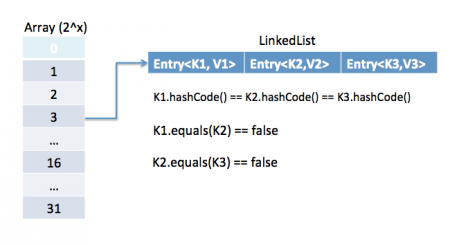
UnsupportedOperationException is the exception used to indicate that the operation is not supported. It’s used extensively in [JDK](http://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm) classes, in collections framework java.util.Collections.UnmodifiableCollection throws this exception for all add and removeoperations.

1. **How HashMap works in Java?**

HashMap stores key-value pair in Map.Entry static nested class implementation. HashMap works on hashing algorithm and uses hashCode() and equals() method in put and get methods.

When we call put method by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair. The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.

When we call get method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it’s value. Below image will explain these detail clearly.

[](http://cdn.journaldev.com/wp-content/uploads/2013/01/java-hashmap-entry-impl.png)

The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is **16** and load factor is 0.75. 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. The capacity is always power of 2, so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it’s good idea to initialize the HashMap with correct capacity and load factor.

1. **What is the importance of hashCode() and equals() methods?**

HashMap uses Key object hashCode() and equals() method to determine the index to put the key-value pair. These methods are also used when we try to get value from HashMap. If these methods are not implemented correctly, two different Key’s might produce same hashCode() and equals() output and in that case rather than storing it at different location, HashMap will consider them same and overwrite them.

Similarly all the collection classes that doesn’t store duplicate data use hashCode() and equals() to find duplicates, so it’s very important to implement them correctly. The implementation of equals() and hashCode() should follow these rules.

* 1. If o1.equals(o2), then o1.hashCode() == o2.hashCode()should always be true.
  2. If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

1. **Can we use any class as Map key?**

We can use any class as Map Key, however following points should be considered before using them.

* 1. If the class overrides equals() method, it should also override hashCode() method.
  2. The class should follow the rules associated with equals() and hashCode() for all instances. Please refer earlier question for these rules.
  3. If a class field is not used in equals(), you should not use it in hashCode() method.
  4. Best practice for user defined key class is to make it immutable, so that hashCode() value can be cached for fast performance. Also immutable classes make sure that hashCode() and equals() will not change in future that will solve any issue with mutability.  
     For example, let’s say I have a class MyKey that I am using for HashMap key.
  5. //MyKey name argument passed is used for equals() and hashCode()
  6. MyKey key = new MyKey("Pankaj"); //assume hashCode=1234
  7. myHashMap.put(key, "Value");
  8. // Below code will change the key hashCode() and equals()
  9. // but it's location is not changed.
  10. key.setName("Amit"); //assume new hashCode=7890
  11. //below will return null, because HashMap will try to look for key
  12. //in the same index as it was stored but since key is mutated,
  13. //there will be no match and it will return null.

myHashMap.get(new MyKey("Pankaj"));

This is the reason why String and Integer are mostly used as HashMap keys.

1. **What are different Collection views provided by Map interface?**

Map interface provides three collection views:

* 1. **Set<K> keySet()**: Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator’s own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.
  2. **Collection<V> values()**: Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator’s own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.
  3. **Set<Map.Entry<K, V>> entrySet()**: Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator’s own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

1. **What is difference between HashMap and Hashtable?**

HashMap and Hashtable both implements Map interface and looks similar, however there are following difference between HashMap and Hashtable.

* 1. HashMap allows null key and values whereas Hashtable doesn’t allow null key and values.
  2. Hashtable is synchronized but HashMap is not synchronized. So HashMap is better for single threaded environment, Hashtable is suitable for multi-threaded environment.
  3. LinkedHashMap was introduced in Java 1.4 as a subclass of HashMap, so incase you want iteration order, you can easily switch from HashMap to LinkedHashMap but that is not the case with Hashtable whose iteration order is unpredictable.
  4. HashMap provides Set of keys to iterate and hence it’s fail-fast but Hashtable provides Enumeration of keys that doesn’t support this feature.
  5. Hashtable is considered to be legacy class and if you are looking for modifications of Map while iterating, you should use ConcurrentHashMap.

1. **How to decide between HashMap and TreeMap?**

For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative. If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.

1. **What are similarities and difference between ArrayList and Vector?**

ArrayList and Vector are similar classes in many ways.

* 1. Both are index based and backed up by an array internally.
  2. Both maintains the order of insertion and we can get the elements in the order of insertion.
  3. The iterator implementations of ArrayList and Vector both are fail-fast by design.
  4. ArrayList and Vector both allows null values and random access to element using index number.

These are the differences between ArrayList and Vector.

* 1. Vector is synchronized whereas ArrayList is not synchronized. However if you are looking for modification of list while iterating, you should use CopyOnWriteArrayList.
  2. ArrayList is faster than Vector because it doesn’t have any overhead because of synchronization.
  3. ArrayList is more versatile because we can get synchronized list or read-only list from it easily using Collections utility class.

1. **What is difference between Array and ArrayList? When will you use Array over ArrayList?**

Arrays can contain primitive or Objects whereas ArrayList can contain only Objects.  
Arrays are fixed size whereas ArrayList size is dynamic.  
Arrays doesn’t provide a lot of features like ArrayList, such as addAll, removeAll, iterator etc.

Although ArrayList is the obvious choice when we work on list, there are few times when array are good to use.

* 1. If the size of list is fixed and mostly used to store and traverse them.
  2. For list of primitive data types, although Collections use autoboxing to reduce the coding effort but still it makes them slow when working on fixed size primitive data types.
  3. If you are working on fixed multi-dimensional situation, using [][] is far more easier than List<List<>>

1. **What is difference between ArrayList and LinkedList?**

ArrayList and LinkedList both implement List interface but there are some differences between them.

* 1. ArrayList is an index based data structure backed by Array, so it provides random access to it’s elements with performance as O(1) but LinkedList stores data as list of nodes where every node is linked to it’s previous and next node. So even though there is a method to get the element using index, internally it traverse from start to reach at the index node and then return the element, so performance is O(n) that is slower than ArrayList.
  2. Insertion, addition or removal of an element is faster in LinkedList compared to ArrayList because there is no concept of resizing array or updating index when element is added in middle.
  3. LinkedList consumes more memory than ArrayList because every node in LinkedList stores reference of previous and next elements.

1. **Which collection classes provide random access of it’s elements?**

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements. Download [java collections pdf](http://cdn.journaldev.com/wp-content/uploads/2013/01/java-collections-framework.pdf) for more information.

1. **What is EnumSet?**

java.util.EnumSet is Set implementation to use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. EnumSet is not synchronized and null elements are not allowed. It also provides some useful methods like copyOf(Collection c), of(E first, E… rest) and complementOf(EnumSet s).

Check this post for [java enum tutorial](http://www.journaldev.com/716/java-enum).

1. **Which collection classes are thread-safe?**

Vector, Hashtable, Properties and Stack are synchronized classes, so they are thread-safe and can be used in multi-threaded environment. Java 1.5 Concurrent API included some collection classes that allows modification of collection while iteration because they work on the clone of the collection, so they are safe to use in multi-threaded environment.

1. **What are concurrent Collection Classes?**

Java 1.5 Concurrent package (java.util.concurrent) contains thread-safe collection classes that allow collections to be modified while iterating. By design Iterator implementation in java.utilpackages are fail-fast and throws ConcurrentModificationException. But Iterator implementation in java.util.concurrent packages are fail-safe and we can modify the collection while iterating. Some of these classes are CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet.

Read these posts to learn about them in more detail.

* 1. [Avoid ConcurrentModificationException](http://www.journaldev.com/378/java-util-concurrentmodificationexception)
  2. [CopyOnWriteArrayList Example](http://www.journaldev.com/1289/copyonwritearraylist-java)
  3. [HashMap vs ConcurrentHashMap](http://www.journaldev.com/122/java-concurrenthashmap-example-iterator)

1. **What is BlockingQueue?**

java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem. We don’t need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as it’s handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.  
Check this post for use of BlockingQueue for [producer-consumer problem](http://www.journaldev.com/1034/java-blockingqueue-example).

1. **What is Queue and Stack, list their differences?**

Both Queue and Stack are used to store data before processing them. java.util.Queue is an interface whose implementation classes are present in java concurrent package. Queue allows retrieval of element in First-In-First-Out (FIFO) order but it’s not always the case. There is also Deque interface that allows elements to be retrieved from both end of the queue.  
Stack is similar to queue except that it allows elements to be retrieved in Last-In-First-Out (LIFO) order.  
Stack is a class that extends Vector whereas Queue is an interface.

1. **What is Collections Class?**

java.util.Collections is a utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, “wrappers”, which return a new collection backed by a specified collection, and a few other odds and ends.

This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

1. **What is Comparable and Comparator interface?**

Java provides Comparable interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has compareTo(T obj) method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as argument.

But, in most real life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where we need to use Comparator interface because Comparable.compareTo(Object o) method implementation can sort based on one field only and we can’t chose the field on which we want to sort the Object.

Comparator interface compare(Object o1, Object o2) method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.

Check this post for use of Comparable and Comparator interface to [sort objects](http://www.journaldev.com/780/comparable-and-comparator-in-java-example).

1. **What is difference between Comparable and Comparator interface?**

Comparable and Comparator interfaces are used to sort collection or array of objects.

Comparable interface is used to provide the natural sorting of objects and we can use it to provide sorting based on single logic.  
Comparator interface is used to provide different algorithms for sorting and we can chose the comparator we want to use to sort the given collection of objects.

1. **How can we sort a list of Objects?**

If we need to sort an array of Objects, we can use Arrays.sort(). If we need to sort a list of objects, we can use Collections.sort(). Both these classes have overloaded sort() methods for natural sorting (using Comparable) or sorting based on criteria (using Comparator).  
Collections internally uses Arrays sorting method, so both of them have same performance except that Collections take sometime to convert list to array.

1. **While passing a Collection as argument to a function, how can we make sure the function will not be able to modify it?**

We can create a read-only collection using Collections.unmodifiableCollection(Collection c)method before passing it as argument, this will make sure that any operation to change the collection will throw UnsupportedOperationException.

1. **How can we create a synchronized collection from given collection?**

We can use Collections.synchronizedCollection(Collection c) to get a synchronized (thread-safe) collection backed by the specified collection.

1. **What are common algorithms implemented in Collections Framework?**

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections.  
Some of them are sorting, searching, shuffling, min-max values.

1. **What is Big-O notation? Give some examples?**

The Big-O notation describes the performance of an algorithm in terms of number of elements in a data structure. Since Collection classes are actually data structures, we usually tend to use Big-O notation to chose the collection implementation to use based on time, memory and performance.

Example 1: ArrayList get(index i) is a constant-time operation and doesn’t depend on the number of elements in the list. So it’s performance in Big-O notation is O(1).  
Example 2: A linear search on array or list performance is O(n) because we need to search through entire list of elements to find the element.

1. **What are best practices related to Java Collections Framework?**
   1. Chosing the right type of collection based on the need, for example if size is fixed, we might want to use Array over ArrayList. If we have to iterate over the Map in order of insertion, we need to use TreeMap. If we don’t want duplicates, we should use Set.
   2. Some collection classes allows to specify the initial capacity, so if we have an estimate of number of elements we will store, we can use it to avoid rehashing or resizing.
   3. Write program in terms of interfaces not implementations, it allows us to change the implementation easily at later point of time.
   4. Always use Generics for type-safety and avoid ClassCastException at runtime.
   5. Use immutable classes provided by JDK as key in Map to avoid implementation of hashCode() and equals() for our custom class.
   6. Use Collections utility class as much as possible for algorithms or to get read-only, synchronized or empty collections rather than writing own implementation. It will enhance code-reuse with greater stability and low maintainability.
2. **What is Java Priority Queue?**

PriorityQueue is an unbounded queue based on a priority heap and the elements are ordered in their natural order or we can provide [Comparator](http://www.journaldev.com/780/comparable-and-comparator-in-java-example) for ordering at the time of creation. PriorityQueue doesn’t allow null values and we can’t add any object that doesn’t provide natural ordering or we don’t have any comparator for them for ordering. Java PriorityQueue is not [thread-safe](http://www.journaldev.com/1061/thread-safety-in-java) and provided O(log(n)) time for enqueing and dequeing operations. Check this post for [java priority queue example](http://www.journaldev.com/1642/java-priority-queue-priorityqueue-example).

1. **Why can’t we write code as List<Number> numbers = new ArrayList<Integer>();?**

Generics doesn’t support sub-typing because it will cause issues in achieving type safety. That’s why List<T> is not considered as a subtype of List<S> where S is the super-type of T. To understanding why it’s not allowed, let’s see what could have happened if it has been supported.

List<Long> listLong = new ArrayList<Long>();

listLong.add(Long.valueOf(10));

List<Number> listNumbers = listLong; // compiler error

listNumbers.add(Double.valueOf(1.23));

As you can see from above code that IF generics would have been supporting sub-typing, we could have easily add a Double to the list of Long that would have caused ClassCastException at runtime while traversing the list of Long.

1. **Why can’t we create generic array? or write code as List<Integer>[] array = new ArrayList<Integer>[10];**

We are not allowed to create generic arrays because array carry type information of it’s elements at runtime. This information is used at runtime to throw ArrayStoreException if elements type doesn’t match to the defined type. Since generics type information gets erased at runtime by Type Erasure, the array store check would have been passed where it should have failed. Let’s understand this with a simple example code.

List<Integer>[] intList = new List<Integer>[5]; // compile error

Object[] objArray = intList;

List<Double> doubleList = new ArrayList<Double>();

doubleList.add(Double.valueOf(1.23));

objArray[0] = doubleList; // this should fail but it would pass because at runtime intList and doubleList both are just List

Arrays are covariant by nature i.e S[] is a subtype of T[] whenever S is a subtype of T but generics doesn’t support covariance or sub-typing as we saw in last question. So if we would have been allowed to create generic arrays, because of type erasure we would not get array store exception even though both types are not related.

1. **What is Exception in Java?**

Exception is an error event that can happen during the execution of a program and disrupts it’s normal flow. Exception can arise from different kind of situations such as wrong data entered by user, hardware failure, network connection failure etc.

Whenever any error occurs while executing a java statement, an exception object is created and then [**JRE**](http://www.journaldev.com/546/difference-jdk-vs-jre-vs-jvm) tries to find exception handler to handle the exception. If suitable exception handler is found then the exception object is passed to the handler code to process the exception, known as **catching the exception**. If no handler is found then application throws the exception to runtime environment and JRE terminates the program.

**Java Exception handling** framework is used to handle runtime errors only, compile time errors are not handled by exception handling framework.

1. **What are the Exception Handling Keywords in Java?**

There are four keywords used in java exception handling.

* 1. **throw**: Sometimes we explicitly want to create exception object and then throw it to halt the normal processing of the program. **throw** keyword is used to throw exception to the runtime to handle it.
  2. **throws**: When we are throwing any checked exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it’s caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with **main()** method also.
  3. **try-catch**: We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
  4. **finally**: finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurrs or not.

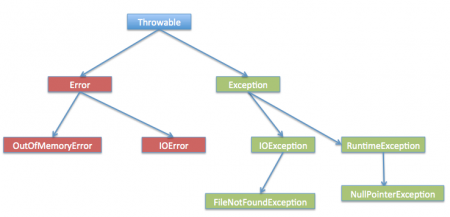
1. **Explain Java Exception Hierarchy?**

Java Exceptions are hierarchical and [inheritance](http://www.journaldev.com/644/inheritance-in-java-example) is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into checked exceptions and runtime exception.

**Errors** are exceptional scenarios that are out of scope of application and it’s not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error.

**Checked Exceptions** are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException. We should catch this exception and provide useful message to user and log it properly for debugging purpose. Exception is the parent class of all Checked Exceptions.

**Runtime Exceptions** are caused by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime. RuntimeException is the parent class of all runtime exceptions.

[](http://cdn.journaldev.com/wp-content/uploads/2013/07/exception-hierarchy.png)

1. **What are important methods of Java Exception Class?**

Exception and all of it’s subclasses doesn’t provide any specific methods and all of the methods are defined in the base class Throwable.

* 1. **String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it’s constructor.
  2. **String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.
  3. **synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
  4. **String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
  5. **void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

1. **Explain Java 7 ARM Feature and multi-catch block?**

If you are catching a lot of exceptions in a single try block, you will notice that catch block code looks very ugly and mostly consists of redundant code to log the error, keeping this in mind Java 7 one of the feature was multi-catch block where we can catch multiple exceptions in a single catch block. The catch block with this feature looks like below:

catch(IOException | SQLException | Exception ex){

logger.error(ex);

throw new MyException(ex.getMessage());

}

Most of the time, we use finally block just to close the resources and sometimes we forget to close them and get runtime exceptions when the resources are exhausted. These exceptions are hard to debug and we might need to look into each place where we are using that type of resource to make sure we are closing it. So java 7 one of the improvement was **try-with-resources** where we can create a resource in the try statement itself and use it inside the try-catch block. When the execution comes out of try-catch block, runtime environment automatically close these resources. Sample of try-catch block with this improvement is:

try (MyResource mr = new MyResource()) {

System.out.println("MyResource created in try-with-resources");

} catch (Exception e) {

e.printStackTrace();

}

Read more about this at [**Java 7 ARM**](http://www.journaldev.com/592/java-try-with-resources).

1. **What is difference between Checked and Unchecked Exception in Java?**
   1. Checked Exceptions should be handled in the code using try-catch block or else main() method should use throws keyword to let JRE know about these exception that might be thrown from the program. Unchecked Exceptions are not required to be handled in the program or to mention them in throws clause.
   2. Exception is the super class of all checked exceptions whereas RuntimeException is the super class of all unchecked exceptions.
   3. Checked exceptions are error scenarios that are not caused by program, for example FileNotFoundException in reading a file that is not present, whereas Unchecked exceptions are mostly caused by poor programming, for example NullPointerException when invoking a method on an object reference without making sure that it’s not null.
2. **What is difference between throw and throws keyword in Java?**

throws keyword is used with method signature to declare the exceptions that the method might throw whereas throw keyword is used to disrupt the flow of program and handing over the exception object to runtime to handle it.

1. **How to write custom exception in Java?**

We can extend Exception class or any of it’s subclasses to create our custom exception class. The custom exception class can have it’s own variables and methods that we can use to pass error codes or other exception related information to the exception handler.

A simple example of custom exception is shown below.

package com.journaldev.exceptions;

import java.io.IOException;

public class MyException extends IOException {

private static final long serialVersionUID = 4664456874499611218L;

private String errorCode="Unknown\_Exception";

public MyException(String message, String errorCode){

super(message);

this.errorCode=errorCode;

}

public String getErrorCode(){

return this.errorCode;

}

}

1. **What is OutOfMemoryError in Java?**

OutOfMemoryError in Java is a subclass of java.lang.VirtualMachineError and it’s thrown by JVM when it ran out of heap memory. We can fix this error by providing more memory to run the java application through java options.

$>java MyProgram -Xms1024m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=256m

1. **What are different scenarios causing “Exception in thread main”?**

Some of the common main thread exception scenarios are:

* 1. **Exception in thread main java.lang.UnsupportedClassVersionError**: This exception comes when your java class is compiled from another JDK version and you are trying to run it from another java version.
  2. **Exception in thread main java.lang.NoClassDefFoundError**: There are two variants of this exception. The first one is where you provide the class full name with .class extension. The second scenario is when Class is not found.
  3. **Exception in thread main java.lang.NoSuchMethodError: main**: This exception comes when you are trying to run a class that doesn’t have main method.
  4. **Exception in thread “main” java.lang.ArithmeticException**: Whenever any exception is thrown from main method, it prints the exception is console. The first part explains that exception is thrown from main method, second part prints the exception class name and then after a colon, it prints the exception message.

Read more about these at [Java Exception in Thread main](http://www.journaldev.com/611/java-exception-in-thread-main-understanding-with-examples).

1. **What is difference between final, finally and finalize in Java?**

final and finally are keywords in java whereas finalize is a method.

final keyword can be used with class variables so that they can’t be reassigned, with class to avoid extending by classes and with methods to avoid overriding by subclasses, finally keyword is used with try-catch block to provide statements that will always gets executed even if some exception arises, usually finally is used to close resources. finalize() method is executed by Garbage Collector before the object is destroyed, it’s great way to make sure all the global resources are closed.

Out of the three, only finally is related to java exception handling.

1. **What happens when exception is thrown by main method?**

When exception is thrown by main() method, Java Runtime terminates the program and print the exception message and stack trace in system console.

1. **Can we have an empty catch block?**

We can have an empty catch block but it’s the example of worst programming. We should never have empty catch block because if the exception is caught by that block, we will have no information about the exception and it wil be a nightmare to debug it. There should be at least a logging statement to log the exception details in console or log files.

1. **Provide some Java Exception Handling Best Practices?**

Some of the best practices related to Java Exception Handling are:

* 1. Use Specific Exceptions for ease of debugging.
  2. Throw Exceptions Early (Fail-Fast) in the program.
  3. Catch Exceptions late in the program, let the caller handle the exception.
  4. Use Java 7 ARM feature to make sure resources are closed or use finally block to close them properly.
  5. Always log exception messages for debugging purposes.
  6. Use multi-catch block for cleaner close.
  7. Use custom exceptions to throw single type of exception from your application API.
  8. Follow naming convention, always end with Exception.
  9. Document the Exceptions Thrown by a method using @throws in javadoc.
  10. Exceptions are costly, so throw it only when it makes sense. Else you can catch them and provide null or empty response.

Read more about them in detail at [Java Exception Handling Best Practices](http://www.journaldev.com/1696/exception-handling-in-java#exception-best-practices).

1. **What is the problem with below programs and how do we fix it?**

In this section, we will look into some programming questions related to java exceptions.

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. import java.io.FileNotFoundException;
  4. import java.io.IOException;
  5. public class TestException {
  6. public static void main(String[] args) {
  7. try {
  8. testExceptions();
  9. } catch (FileNotFoundException | IOException e) {
  10. e.printStackTrace();
  11. }
  12. }


  16. public static void testExceptions() throws IOException, FileNotFoundException{
  18. }

}

Above program won’t compile and you will get error message as “The exception FileNotFoundException is already caught by the alternative IOException”. This is because FileNotFoundException is subclass of IOException, there are two ways to solve this problem.

First way is to use single catch block for both the exceptions.

try {

testExceptions();

}catch(FileNotFoundException e){

e.printStackTrace();

}catch (IOException e) {

e.printStackTrace();

}

Another way is to remove the FileNotFoundException from multi-catch block.

try {

testExceptions();

}catch (IOException e) {

e.printStackTrace();

}

You can chose any of these approach based on your catch block code.

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. import java.io.FileNotFoundException;
  4. import java.io.IOException;
  5. import javax.xml.bind.JAXBException;
  6. public class TestException1 {
  7. public static void main(String[] args) {
  8. try {
  9. go();
  10. } catch (IOException e) {
  11. e.printStackTrace();
  12. } catch (FileNotFoundException e) {
  13. e.printStackTrace();
  14. } catch (JAXBException e) {
  15. e.printStackTrace();
  16. }
  17. }
  18. public static void go() throws IOException, JAXBException, FileNotFoundException{
  20. }

}

The program won’t compile because FileNotFoundException is subclass of IOException, so the catch block of FileNotFoundException is unreachable and you will get error message as “Unreachable catch block for FileNotFoundException. It is already handled by the catch block for IOException”.

You need to fix the catch block order to solve this issue.

try {

go();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (JAXBException e) {

e.printStackTrace();

}

Notice that JAXBException is not related to IOException or FileNotFoundException and can be put anywhere in above catch block hierarchy.

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. import java.io.IOException;
  4. import javax.xml.bind.JAXBException;
  5. public class TestException2 {
  6. public static void main(String[] args) {
  7. try {
  8. foo();
  9. } catch (IOException e) {
  10. e.printStackTrace();
  11. }catch(JAXBException e){
  12. e.printStackTrace();
  13. }catch(NullPointerException e){
  14. e.printStackTrace();
  15. }catch(Exception e){
  16. e.printStackTrace();
  17. }
  18. }
  19. public static void foo() throws IOException{
  21. }

}

The program won’t compile because JAXBException is a checked exception and foo() method should throw this exception to catch in the calling method. You will get error message as “Unreachable catch block for JAXBException. This exception is never thrown from the try statement body”.

To solve this issue, you will have to remove the catch block of JAXBException.

Notice that catching NullPointerException is valid because it’s an unchecked exception.

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. public class TestException3 {
  4. public static void main(String[] args) {
  5. try{
  6. bar();
  7. }catch(NullPointerException e){
  8. e.printStackTrace();
  9. }catch(Exception e){
  10. e.printStackTrace();
  11. }
  13. foo();
  14. }
  15. public static void bar(){
  17. }
  19. public static void foo() throws NullPointerException{
  21. }

}

This is a trick question, there is no problem with the code and it will compile successfully. We can always catch Exception or any unchecked exception even if it’s not in the throws clause of the method.

Similarly if a method (foo) declares unchecked exception in throws clause, it is not mandatory to handle that in the program.

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. import java.io.IOException;
  4. public class TestException4 {
  5. public void start() throws IOException{
  6. }
  8. public void foo() throws NullPointerException{
  10. }
  11. }
  12. class TestException5 extends TestException4{
  14. public void start() throws Exception{
  15. }
  17. public void foo() throws RuntimeException{
  19. }

}

The above program won’t compile because start() method signature is not same in subclass. To fix this issue, we can either change the method singnature in subclass to be exact same as superclass or we can remove throws clause from subclass method as shown below.

@Override

public void start(){

}

* 1. **What is the problem with below program?**
  2. package com.journaldev.exceptions;
  3. import java.io.IOException;
  4. import javax.xml.bind.JAXBException;
  5. public class TestException6 {
  6. public static void main(String[] args) {
  7. try {
  8. foo();
  9. } catch (IOException | JAXBException e) {
  10. e = new Exception("");
  11. e.printStackTrace();
  12. }catch(Exception e){
  13. e = new Exception("");
  14. e.printStackTrace();
  15. }
  16. }
  17. public static void foo() throws IOException, JAXBException{
  19. }

}

The above program won’t compile because exception object in multi-catch block is final and we can’t change it’s value. You will get compile time error as “The parameter e of a multi-catch block cannot be assigned”.

We have to remove the assignment of “e” to new exception object to solve this error.

**Why do we need change to Java again?**

Oracle Corporation has introduced a lot of new concepts in Java SE 8 to introduce the following benefits:

* **To Utilize Current Multi-Core CPUs Efficiently**

Recently, we can observe drastic changes in Hardware. Now-a-days, all systems are using Multi-Core CPUs(2,4,8,16-Core etc.) to deploy and run their Applications. We need new Programming Constructs in Java to utilize these Multi-Core Processors efficiently to develop Highly Concurrently and Highly Scalable applications.

* **To Utilize FP Features**

Oracle Corporation has introduced a lot of FP(Functional Programming) concepts as part of Java SE 8 to utilize the advantages of FP.

**Java SE 8 New Features?**

* Lambda Expressions
* Functional Interfaces
* Stream API
* Date and Time API
* Interface Default Methods and Static Methods
* Spliterator
* Method and Constructor References
* Collections API Enhancements
* Concurrency Utils Enhancements
* Fork/Join Framework Enhancements
* Internal Iteration
* Parallel Array and Parallel Collection Operations
* Optional
* Type Annotations and Repeatable Annotations
* Method Parameter Reflection
* Base64 Encoding and Decoding
* IO and NIO2 Enhancements
* Nashorn JavaScript Engine
* javac Enhancements
* JVM Changes
* Java 8 Compact Profiles: compact1,compact2,compact3
* JDBC 4.2
* JAXP 1.6
* Java DB 10.10
* Networking
* Security Changes

**Advantages of Java SE 8 New Features?**

We can get the following benefits from Java SE 8 New Features:

* More Concise and Readable code
* More Reusable code
* More Testable and Maintainable Code
* Highly Concurrent and Highly Scalable Code
* Write Parallel Code
* Write Database Like Operations
* Better Performance Applications
* More Productive code

**What is Lambda Expression?**

Lambda Expression is an anonymous function which accepts a set of input parameters and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

**What are the three parts of a Lambda Expression? What is the type of Lambda Expression?**

A Lambda Expression contains 3 parts:

* Parameter List

A Lambda Expression can contain zero or one or more parameters. It is optional.

* Lambda Arrow Operator

“->” is known as Lambda Arrow operator. It separates parameters list and body.

* Lambda Expression Body

The type of “Journal Dev” is java.lang.String. The type of “true” is Boolean. In the same way, what is the type of a Lambda Expression?  
The Type of a Lambda Expression is a Functional Interface.

Example:- What is the type of the following Lambda Expression?

|  |  |
| --- | --- |
| 1 | () -> System.out.println("Hello World"); |

This Lambda Expression does not have parameters and does return any results. So it’s type is “java.lang.Runnable” Functional Interface.

**What is a Functional Interface? What is SAM Interface?**

A Functional Interface is an interface, which contains one and only one abstract method. Functional Interface is also know as SAM Interface because it contains only one abstract method.

SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

**Is is possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?**

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

We need to follow these rules to define a Functional Interface:

* Define an interface with one and only one abstract method.
* We cannot define more than one abstract method.
* Use @FunctionalInterface annotation in interface definition.
* We can define any number of other methods like Default methods, Static methods.
* If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

**Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?**

It is not mandatory to define a Functional Interface with @FunctionalInterface annotation. If we don’t want, We can omit this annotation. However, if we use it in Functional Interface definition, Java Compiler forces to use one and only one abstract method inside that interface.

Why do we need Functional Interfaces? The type of a Java SE 8’s Lambda Expression is a Functional Interface. Whereever we use Lambda Expressions that means we are using Functional Interfaces.

**When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?**

When our Java project wants to perform the following operations, it’s better to use Java 8 Stream API to get lot of benefits:

* When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
* When want to Perform operations Lazily.
* When we want to write Functional Style programming.
* When we want to perform Parallel Operations.
* When want to use Internal Iteration
* When we want to perform Pipelining operations.
* When we want to achieve better performance.

**Explain Differences between Collection API and Stream API?**

|  |  |  |
| --- | --- | --- |
| S.NO. | COLLECTION API | STREAM API |
| 1. | It’s available since Java 1.2 | It is introduced in Java SE8 |
| 2. | It is used to store Data(A set of Objects). | It is used to compute data(Computation on a set of Objects). |
| 3. | We can use both Spliterator and Iterator to iterate elements. | We can use both Spliterator and Iterator to iterate elements. |
| 4. | It is used to store limited number of Elements. | It is used to store either Limited or Infinite Number of Elements. |
| 5. | Typically, it uses Internal Iteration concept to iterate Elements. | It uses External Iteration to iterate Elements. |
| 6. | Collection Object is constructed Eagerly. | Stream Object is constructed Lazily. |
| 7. | We add elements to Collection object only after it is computed completely. | We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand. |
| 8. | We can iterate and consume elements from a Collection Object at any number of times. | We can iterate and consume elements from a Stream Object only once. |

**What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?**

Spliterator stands for Splitable Iterator. It is newly introduced by Oracle Corporation as part Java SE 8.  
Like Iterator and ListIterator, It is also one of the Iterator interface.

|  |  |  |
| --- | --- | --- |
| S.NO. | SPLITERATOR | ITERATOR |
| 1. | It is introduced in Java SE 8. | It is available since Java 1.2. |
| 2. | Splitable Iterator | Non-Splitable Iterator |
| 3. | It is used in Stream API. | It is used for Collection API. |
| 4. | It uses Internal Iteration concept to iterate Streams. | It uses External Iteration concept to iterate Collections. |
| 5. | We can use Spliterator to iterate Streams in Parallel and Sequential order. | We can use Spliterator to iterate Collections only in Sequential order. |
| 6. | We can get Spliterator by calling spliterator() method on Stream Object. | We can get Iterator by calling iterator() method on Collection Object. |
| 7. | Important Method: tryAdvance() | Important Methods: next(), hasNext() |

**What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?**

**Optional:**  
Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.

It is used to represent optional values that is either exist or not exist. It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.

It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

**Main Advantage of Optional is:**

* It is used to avoid null checks.
* It is used to avoid “NullPointerException”.

**What is Type Inference? Is Type Inference available in older versions like Java 7 and Before 7 or it is available only in Java SE 8?**

Type Inference means determining the Type by compiler at compile-time.

It is not new feature in Java SE 8. It is available in Java 7 and before Java 7 too.

**Before Java 7:-**  
Let us explore Java arrays. Define a String of Array with values as shown below:

|  |  |
| --- | --- |
| 1 | String str[] = { "Java 7", "Java 8", "Java 9" }; |

Here we have assigned some String values at right side, but not defined it’s type. Java Compiler automatically infers it’s type and creates a String of Array.

**Java 7:-**  
Oracle Corporation has introduced “Diamond Operator” new feature in Java SE 7 to avoid unnecessary Type definition in Generics.

|  |  |
| --- | --- |
| 1 | Map<String,List<Customer>> customerInfoByCity = new HashMap<>(); |

Here we have not defined Type information at right side, simply defined Java SE 7’s Diamond Operator “”.

**Java SE 8:-**  
Oracle Corporation has enhanced this Type Inference concept a lot in Java SE 8. We use this concept to define Lambda Expressions, Functions, Method References etc.

|  |  |
| --- | --- |
| 1 | ToIntBiFunction<Integer,Integer> add = (a,b) -> a + b; |

Here Java Compiler observes the type definition available at left-side and determines the type of Lambda Expression parameters a and b as Integers.

That’s it about Java 8 Interview Questions.

**What is Internal Iteration in Java SE 8?**

Before Java 8, We don’t Internal Iteration concept. Java 8 has introduced a new feature known as “Internal Iteration”. Before Java 8, Java Language has only External Iteration to iterate elements of an Aggregated Object like Collections, Arrays etc.

Internal Iteration means “Iterating an Aggregated Object elements one by one internally by Java API”. Instead of Java Application do iteration externally, We ask Java API to do this job internally.

**Differences between External Iteration and Internal Iteration?**

|  |  |  |
| --- | --- | --- |
| S.NO. | EXTERNAL ITERATION | INTERNAL ITERATION |
| 1. | Available before Java 8 too. | It is introduced in Java SE 8 |
| 2. | Iterating an Aggregated Object elements externally. | Iterating an Aggregated Object elements internally (background). |
| 3. | Iterate elements by using for-each loop and Iterators like Enumeration, Iterator, ListIterator. | Iterate elements by using Java API like “forEach” method. |
| 4. | Iterating elements in Sequential and In-Order only. | Not required to iterate elements in Sequential order. |
| 5. | It follows OOP approach that is Imperative Style. | It follows Functional Programming approach that is Declarative Style. |
| 6. | It does NOT separate responsibilities properly that is, it defines both “What is to be done” and “How it is to be done”. | It defines only “What is to be done”. No need to worry about “How it is to be done”. Java API takes care about “How to do”. |
| 7. | Less Readable Code. | More Readable code. |

**What are the major drawbacks of External Iteration?**

External Iteration has the following drawbacks:

* We need to write code in Imperative Style.
* There is no clear separation of Responsibilities. Tightly-Coupling between “What is to be done” and “How it is to be done” code.
* Less Readable Code.
* More Verbose and Boilerplate code.
* We have to iterate elements in Sequential order only.
* It does not support Concurrency and Parallelism properly.

**What are the major advantages of Internal Iteration over External Iteration?**

Compare to External Iteration, Internal Iteration has the following advantages:

* As it follows Functional Programming style, we can write Declarative Code.
* More Readable and concise code.
* Avoids writing Verbose and Boilerplate code
* No need to iterate elements in Sequential order.
* It supports Concurrency and Parallelism properly.
* We can write Parallel code to improve application performance.
* Clear separation of Responsibilities. Loosely-Coupling between “What is to be done” and “How it is to be done” code.
* We need to write code only about “What is to be done” and Java API takes care about “How it is to be done” code.

**What is the major drawback of Internal Iteration over External Iteration?**

Compare to External Iteration, Internal Iteration has one major drawback:

* In Internal Iteration, as Java API takes care about Iterating elements internally, we do NOT have control over Iteration.

**What is the major advantage of External Iteration over Internal Iteration?**

Compare to Internal Iteration, External Iteration has one major advantage:

* In External Iteration, as Java API does NOT take care about Iterating elements, we have much control over Iteration.

**When do we need to use Internal Iteration? When do we need to use External Iteration?**

We need to understand the situations to use either Internal Iteration or External Iteration.

* When we need more control over Iteration, we can use External Iteration.
* When we do NOT need more control over Iteration, we can use Internal Iteration.
* When we need to develop Highly Concurrency and Parallel applications and we , we should use Internal Iteration.

**Differences between Intermediate Operations and Terminal Operations of Java 8’s Stream API?**

|  |  |  |
| --- | --- | --- |
| S.NO. | STREAM INTERMEDIATE OPERATIONS | STREAM TERMINAL OPERATIONS |
| 1. | Stream Intermediate operations are not evaluated until we chain it with Stream Terminal Operation. | Stream Terminal Operations are evaluated on it’s own. No need other operations help. |
| 2. | The output of Intermediate Operations is another Stream. | The output of Intermediate Operations is Not a Stream. Something else other than a Stream. |
| 3. | Intermediate Operations are evaluated Lazily. | Terminal Operations are evaluated Eagerly. |
| 4. | We can chain any number of Stream Intermediate Operations. | We can NOT chain Stream Terminal Operations. |
| 5. | We can use any number of Stream Intermediate Operations per Statement. | We can use only one Stream Terminal Operation per Statement. |

**Is it possible to provide method implementations in Java Interfaces? If possible, how do we provide them?**

In Java 7 or earlier, It is not possible to provide method implementations in Interfaces. Java 8 on-wards, it is possible.

In Java SE 8, We can provide method implementations in Interfaces by using the following two new concepts:

* Default Methods
* Static Methods

**What is a Default Method? Why do we need Default methods in Java 8 Interfaces?**

A Default Method is a method which is implemented in an interface with “default” keyword. It’s new featured introduced in Java SE 8.

**We need Default Methods because of the following reasons:**

* It allow us to provide method’s implementation in Interfaces.
* To add new Functionality to Interface without breaking the Classes which implement that Interface.
* To provide elegant Backwards Compatibility Feature.
* To ease of extend the existing Functionality.
* To ease of Maintain the existing Functionality.

**What is a Static Method? Why do we need Static methods in Java 8 Interfaces?**

A Static Method is an Utility method or Helper method, which is associated to a class (or interface). It is not associated to any object.

**We need Static Methods because of the following reasons:**

* We can keep Helper or Utility methods specific to an interface in the same interface rather than in a separate Utility class.
* We do not need separate Utility Classes like Collections, Arrays etc to keep Utility methods.
* Clear separation of Responsibilities. That is we do not need one Utility class to keep all Utility methods of Collection API like Collections etc.
* Easy to extend the API.
* Easy to Maintain the API.

**Differences between Functional Programming and Object-Oriented Programming?**

|  |  |
| --- | --- |
| FUNCTIONAL PROGRAMMING | OOP |
| Does not exist State | Exists State |
| Uses Immutable data | Uses Mutable data |
| It follows Declarative Programming Model | It follows Imperative Programming Model |
| Stateless Programming Model | Stateful Programming Model |
| Main Fcous on: “What you are doing” | Main focus on “How you are doing” |
| Good for Parallel (Concurrency) Programming | Poor for Parallel (Concurrency) Programming |
| Good for BigData processing and analysis | NOT Good for BigData processing and analysis |
| Supports pure Encapsulation | It breaks Encapsulation concept |
| Functions with No-Side Effects | Methods with Side Effects |
| Functions are first-class citizens | Objects are first-class citizens |
| Primary Manipulation Unit is “Function” | Primary Manipulation Unit is Objects(Instances of Classes) |
| Flow Controls: Function calls, Function Calls with Recursion | Flow Controls: Loops, Conditional Statements |
| It uses “Recursion” concept to iterate Collection Data. | It uses “Loop” concept to iterate Collection Data. For example:-For-each loop in Java |
| Order of execution is less importance. | Order of execution is must and very important. |
| Supports both “Abstraction over Data” and “Abstraction over Behavior”. | Supports only “Abstraction over Data”. |
| We use FP when we have few Things with more operations. | We use OOP when we have few Operations with more Things. For example: Things are classes and Operations are Methods in Java. |

NOTE:- For more information about FP, IP and OOP comparisons, Please go through my previous post at: “[Compare FP, OOP(IP)](http://www.journaldev.com/8693/functional-imperative-object-oriented-programming-comparison)”

**Explain issues of Old Java Date API? What are the advantages of Java 8’s Date and Time API over Old Date API and Joda Time API?**

Java’s OLD Java Date API means Date API available before Java SE 8 that is Date, Calendar, SimpleDateFormat etc.

Java’s Old Date API has the following Issues or Drawbacks compare to Java 8’s Date and Time API and Joda Time API.

* Most of the API is deprecated.
* Less Readability.
* java.util.Date is Mutable and not Thread-Safe.
* java.text.SimpleDateFormat is not Thread-Safe.
* Less Performance.

Java SE 8’s Date and Time API has the following Advantages compare to Java’s OLD Date API.

* Very simple to use.
* Human Readable Syntax that is More Readability.
* All API is Thread-Safe.
* Better Performance.

**Why do we need new Date and Time API in Java SE 8?Explain how Java SE 8 Data and Time API solves issues of Old Java Date API?**

We need Java 8’s Date and Time API to develop Highly Performance, Thread-Safe and Highly Scalable Java Applications.

Java 8’s Date and Time API solves all Java’s Old Date API issues by following Immutability and Thread-Safety principles.

**What are the Differences between Java’s OLD Java Date API and Java 8’s Date and Time API?**

**Differences between Java’s OLD Java Date API and Java 8’s Date and Time API:**

|  |  |  |
| --- | --- | --- |
| S.NO. | JAVA’S OLD JAVA DATE API | JAVA 8’S DATE AND TIME API |
| 1. | Available before Java 8 too. | It is introduced in Java SE 8 |
| 2. | Not Thread Safe. | Thread Safe. |
| 3. | Mutable API. | Immutable API. |
| 4. | Less Performance. | Better Performance. |
| 5. | Less Readability. | More Readability. |
| 6. | It’s not recommended to use as its deprecated. | It’s always recommended to use. |
| 7. | Not Extendable. | Easy to Extend. |
| 8. | It defines months values from 0 to 11, that is January = 0. | It defines months values from 1 to 12, that is January = 1. |
| 9. | It’s an old API. | It’s a new API. |

**What is Multiple Inheritance? How Java 8 supports Multiple Inheritance?**

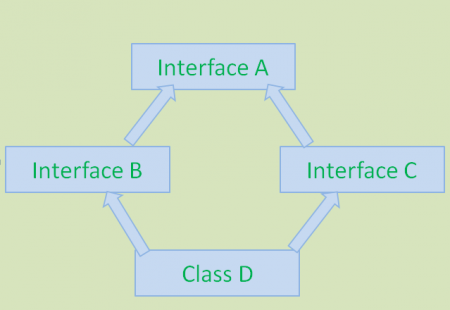
Multiple Inheritance means a class can inherit or extend characteristics and features from more than one parent class.

In Java 7 or Earlier, Multiple Inheritance is not possible because Java follows “A class should extend one and only one class or abstract class” Rule. However, it’s possible to provide Multiple Implementation Inheritance using Interface because Java follows “A class can extend any number of Interfaces” Rule.

However, Java 8 supports “Implementing Methods in Interfaces” by introducing new features: Default methods in Interface. Because of this feature, Java 8 supports Multiple Inheritance with some limitations.

**What is Diamond Problem in Inheritance? How Java 8 Solves this problem?**

A Diamond Problem is a Multiple Inheritance problem. In Java, It occurs when a Class extends more than one Interface which have same method implementation (Default method).



This above diagram shows Diamond Problem. To avoid this problem, Java 7 and Earlier versions does not support methods implementation in interface and also doesn’t support Multiple Inheritance. Java 8 has introduced new feature: Default methods to support Multiple Inheritance with some limitations.

Sample Java SE 8 Code to show this Diamond Problem:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface A{    default void display() { //code goes here }  }  public interface B extends A{ }  public interface C extends A{ }  public class D implements B,C{ } |

In the above code snippet, class D gives compiltime errors because Java Compiler will get bit confusion about which display() has to provide in class D. Class D inherits display() method from both interfaces B and C. To solve this problem, Java SE 8 has given the following remedy:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public interface A{    default void display() { //code goes here }  }  public interface B extends A{ }  public interface C extends A{ }  public class D implements B,C{    void display() {      B.super.display();    }  } |

This **B.super.display();** will solve this Diamond Problem.