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. It also adds up to runtime benefit because the bytecode instructions that do type checking are not generated.

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

[Collection](http://www.journaldev.com/1260/java-collections-framework-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/java-collections-framework-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/java-collections-framework-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/java-collections-framework-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/java-collections-framework-tutorial#queue-interface), [Dequeue](http://www.journaldev.com/1260/java-collections-framework-tutorial#dequeue-interface), [Iterator](http://www.journaldev.com/1260/java-collections-framework-tutorial#iterator-interface), [SortedSet](http://www.journaldev.com/1260/java-collections-framework-tutorial#sortedset-interface), [SortedMap](http://www.journaldev.com/1260/java-collections-framework-tutorial#sortedmap-interface) and [ListIterator](http://www.journaldev.com/1260/java-collections-framework-tutorial#listiterator-interface).

### 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.

### 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.

### 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.

### 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/java-arraylist-vs-copyonwritearraylist-and-exploring-iterator" \o "" \t "_blank).

### 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/hashmap-vs-concurrenthashmap-%E2%80%93-example-and-exploring-iterator" \o "" \t "_blank).

### 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.

### What is UnsupportedOperationException?

UnsupportedOperationException is the exception used to indicate that the operation is not supported. It’s used extensively in JDK classes, in collections framework java.util.Collections.UnmodifiableCollection throws this exception for all add andremove operations.

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.
   * If the class overrides equals() method, it should also override hashCode() method.
   * The class should follow the rules associated with equals() and hashCode() for all instances. Please refer earlier question for these rules.
   * If a class field is not used in equals(), you should not use it in hashCode() method.
   * 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.
2. //MyKey name argument passed is used for equals() and hashCode()
3. MyKey key = new MyKey('Pankaj'); //assume hashCode=1234
4. myHashMap.put(key, 'Value');
5. // Below code will change the key hashCode() and equals()
6. // but it's location is not changed.
7. key.setName('Amit'); //assume new hashCode=7890
8. //below will return null, because HashMap will try to look for key
9. //in the same index as it was stored but since key is mutated,
10. //there will be no match and it will return null.
11. myHashMap.get(new MyKey('Pankaj'));
    * This is the reason why String and Integer are mostly used as HashMap keys.
12. **What are different Collection views provided by Map interface?**

Map interface provides three collection views:

* **Set** 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.
* **Collection** 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.
* **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.