**Terminology**

Class – a logical entity that acts as a blueprint for objects

Instance/Objects – when a class is instantiated with *new* or in other ways. Multiple instances/objects can be created for any class

Reference – location of the object in memory. References can be copied or passed as parameters to methods & constructors multiple times

super – used to access/call the parent class variable and methods except in static context

this – used to call current class members except in static context

**fail-fast** vs. **fail-safe**

1. fail-fast: these iterators fail as soon as the structure of a Collection has been modified since the iteration began. These iterators maintain a *modCount* which is increased for every structural modification. When the count doesn’t match *expectedModCount*, these iterators fail immediately- ConcurrentModificationException
2. fail-safe: these iterators work of the clone of a collection and hence do not throw any exception. Ex – CopyOnWriteArrayList and ConcurrentHashMap.keySet() iterator
3. Collections that support fail-safe iterators should be used in multi-threaded environment

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| **Fail-Fast** Iterators **does not allow** us to make any **modification** to the collection while iterating | **Fail-Safe** Iterators **allows** us to **modify** to the collection while iterating |
| Throws **ConcurrentModificationException** when the collection is modified | **Doesn’t** throw **ConcurrentModificationException** when the collection is modified |
| Uses the **original collection** for the iteration | Uses the **snapshot of the collection** for the iteration |
| We are allowed to **modify the collection** using the **iterator instance** | We are **not allowed** to modify the collection using the **iterator instance** |
| It **does not** need **additional memory** as the iteration happens on the original collection | Requires **additional memory**, as it takes a **snapshot** of the **original collection** |
| Iterators of **ArrayList, HashMap, LinkedList, Vector** | Iterators of **CopyOnWriteArrayList, ConcurrentHashMap, ConcurrentMap** |

**Interface**



Interfaces are used to achieve abstraction in Java along with abstract classes

* + interfaces are abstract and public
  + static & default methods should have body
  + static methods cannot be overridden
  + cannot contain constructors and hence they cannot be used to create objects

**Final**



1. final modifier prevents

* modification of a variable
* overriding a method
* using a class as a base class

**Static**

1. **Static variables** are also known as class variables whose value remain same across all instances/objects of that class.
   * referenced using <ClassName>.<staticMember> i.e., *Math.log() || Collections.emptyMap() || Math.E*
   * created in *Metaspace*(Java8+) when a class is loaded
   * these cannot be referenced/called from a non-static method
   * only one instance of a static variable is created in Metaspace per class
2. Use static fields when
   * value of the variable is *independent of objects*
   * value is *shared across* all the *objects*
3. Static variables can only be created at the *class level*
4. Static methods in an interface need to be implemented and *cannot be overridden* since these are resolved at compile time and method overriding is runtime polymorphism
5. *this* and *super* keywords cannot be used in a static method
6. static methods cannot be abstract
7. **Static blocks** are used for initializing static variables that require additional, multi-statement logic while initialization
8. **Static nested classes** are most widely used approach to create singleton objects as is it doesn't require any synchronization and is easy to learn and implement

**equals() & ==**

****

**==** -> checks the object reference. Returns true if both point to the same memory location

equals() -> checks the contents of the objects not their memory location

**Arrays**



1. Arrays is a data structure that stores multiple values of the same type
2. An array of *n* elements is indexed from 0 to n-1
3. We cannot perform add() or addAll() on the above list since it is backed by the arrays that are of fixed length. Only remove() is permitted.

**ArrayList**

****

1. **What are the differences between ArrayList & LinkedList?**

* Both these implement the *List* interface that are index-based
* Preserve the insertion order and allow duplicates & null values
* Neither synchronized nor thread-safe
* *ArrayList*: internally uses dynamic array to store elements and it is slow since any removal of element/s causes bit shifting in memory.
* *LinkedList*: internally uses double linked list to store elements and it is faster since any removal of element/s doesn’t require bit shifting. This structure implements *Deque* along with List
* ArrayList is preferred when there are high number of get() & set() ops whereas LinkedList is preferred for add() & remove() ops

1. **How do we remove duplicates from an ArrayList?**

* Using *HashSet/LinkedHashSet*(to preserve insertion order) to remove duplicates
* Using Stream.distinct()

1. **How do we synchronize an ArrayList?**

* Using *Collections.synchronizedList()*
* Using *CopyOnWriteArrayList* instead of ArrayList

[**HashMap**](https://www.java67.com/2017/08/top-10-java-hashmap-interview-questions.html)



1. **How does a HashMap work?**

* This data structure works on *hashing* principle and allows for retrieval of data in constant time O(1) if the key is known.
* non-synchronized
* one null key and any number of null values
* put() & get() to store & retrieve data respectively
* when {k, v} are passed to the put(),
  + hashCode() of the key object is called
  + this *hashcode* is used by its hashing function to find a *bucket location*
  + {k, v} are stored as Map.Entry in that bucket
* when a key is passed to the get(),
  + hashCode() of the key object is called
  + this hashcode is used to identify the bucket location
  + equals() is used identify the correct node in the LinkedList to return the associated value object

1. **What happens when two different objects have the same hashcode?**

* *collision* occurs due to the same bucket location
* HashMap stores this object in a *LinkedList*

1. **How do we retrieve the value if two keys have the same hashcode?**

* by calling equals() of the key, we can identify the correct node in the LinkedList to return the value
* always suggested to use an immutable, final object for keys in HashMap

1. **What is the difference between *Concurrent* & *Synchronized* HashMap?**

* Both are thread-safe
* *Synchronized*: object-level lock. Every read & write operation needs to acquire lock. There is a performance overhead due to locking the entire collection. Null support depends on the input Map
* *Concurrent:* hashmap bucket-level lock known as lock stripping. Reads are faster whereas write is done with a lock hence does not throw a ConcurrentModificationException. This uses multitudes of locks. Doesn’t allow null in keys or values

1. **How is the HashMap implemented internally?**

* Array is used for buckets
* LinkedList is used to store {k, v} in each array
* once the LinkedList crosses a certain threshold, it is replaced by Binary Search Tree (jdk 8+)

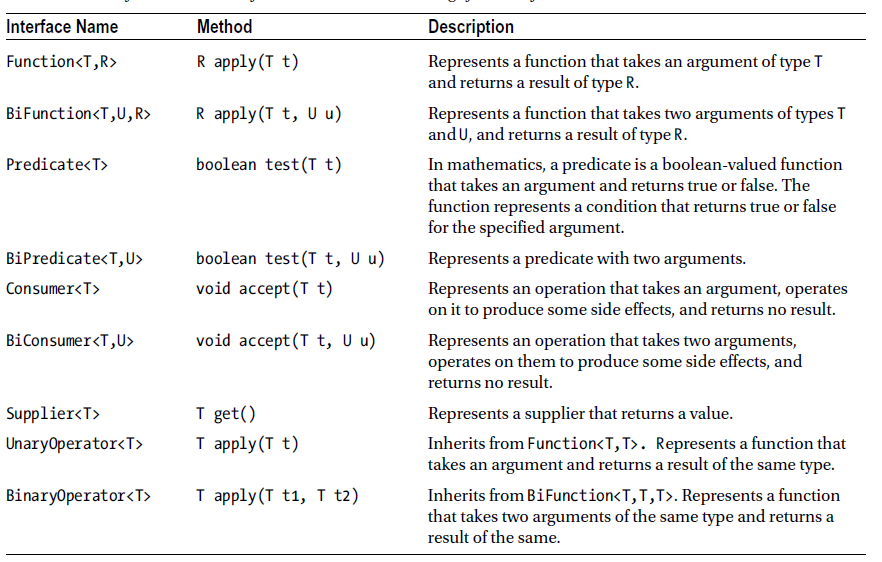
**Functional**

1. **Lambdas** are anonymous functions that are called using a functional set of parameters with a lambda entity



1. **Functional Interfaces** are interfaces that have exactly one abstract method but can have static, default and overridden class methods. Annotated using





**Functional**

