**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. In Java, only references can be used to access an object.

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

primitives – byte (8 bits), short (16), int (32), long (64), float/double (64), char (16 bits), boolean (0 or 1)

synchronized – used to make a resource thread safe. When a synchronized resource is accessed by a thread, it will acquire lock and no other thread is allowed to access it

[volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) – a variable whose value is always accessed from JVM memory. Thus if different threads are accessing this variable all will be having same value

All objects have these methods – toString(), hashCode(), equals(), notify(), notifyAll(), wait()

**Method Overloading**: using same method name with different signatures. Improves code consistency, readability and re-usability



**Singleton**: a class with only one instance i.e., whenever this class is classed the same instance is called.

Singleton class is created with a private constructor and a *static* method that returns the instance of that class.

[Java Singleton Design Pattern Practices with Examples - GeeksforGeeks](https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/)

[How to prevent Singleton Pattern from Reflection, Serialization and Cloning? - GeeksforGeeks](https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/)

[**Class loaders**](https://www.baeldung.com/java-classloaders)

CL’s are part of JRE that are responsible for dynamically loading class definition into JVM.

Bootstrap is the parent class loader.

Extension CL’s load classes that are an extension of the standard core Java classes.

System or application CL’s load all the application level classes into JVM based on classpath environment variable.

**Shallow** vs. **Deep copy**

Shallow Copy: only main object is cloned i.e., inner objects are reference copies and any change in original object is reflected in the copy as well

Deep Copy: all objects are cloned. i.e., any change in clone will not be reflected in the original object

**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

1. **What is CopyOnWriteArrayList?**

* A thread-safe implementation of List interface
* This list creates a copy of the underlying ArrayList for every mutation i.e., set() and add()
* This can be used when there are huge number of iterations rather than mutations

[**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

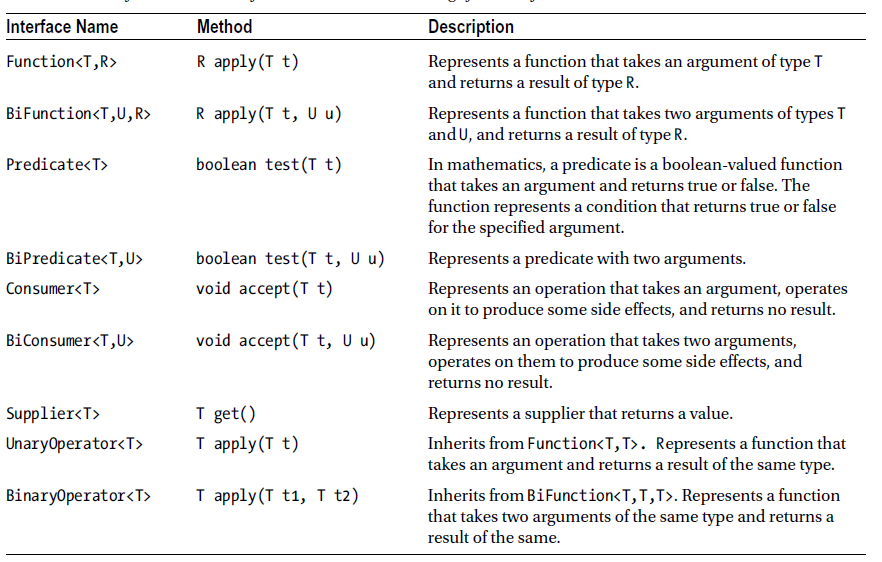


1. **Functional Interfaces** are interfaces that have exactly one abstract method but can have static, default and overridden class methods. Annotated using



1. **Optional** is a container object used to check if an object is non-null or empty





**JVM Internals**

**Concurrency & Multithreading**

**SQL – JDBC, JPA, Data JPA, jOOQ, Transaction Management & Entity Manager**

**Implement Cache – Redis vs Gemfire**

**Spring**

1. Dependency Injection, Front Controller, Factory and Template Design Patterns
2. Bean Scopes, Annotations, Lifecycle
3. Spring Boot
4. Config Server

**JUnit & Mockito**

**Security – OAuth2, JWT, Spring Security**

**AOP – Pointcuts, JointPoint, Advice**

**Connection Management**

1. Spring Cloud Config
2. Customizing HttpClient
3. SOAP Configurations
4. Hikaari Connection Pooling

Design Patterns - [oop - Examples of GoF Design Patterns in Java's core libraries - Stack Overflow](https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries)

Logging – log4j2, Slf4j, Splunk, Kibana

**Streaming**

1. StreamingResponseBody
   1. Spring offers support for asynchronous request processing via StreamingResponseBody
   2. Application can write data directly to the response OutputStream without holding up the Servlet container thread
   3. Always configure TaskExecutor when using this approach
   4. This approach can be used for Highly-concurrent applications
2. RabbitMQ – Offers guaranteed message delivery
3. Kafka – doesn’t offer guaranteed message delivery

Kafka is a streaming & RabbitMQ

**Resources**:

1. <https://www.marcobehler.com/guides>
2. [https://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html](https://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html#axzz6lKxDo7sZ)
3. <https://www.sivalabs.in/2020/09/all-the-resources-you-ever-need-as-a-java-spring-application-developer/>
4. <https://www.javaguides.net/p/spring-mvc-tutorial.html>
5. [Top 50 Java 8 Interview Questions & Answers 2021 - Intellipaat](https://intellipaat.com/blog/interview-question/java-8-interview-questions/)
6. <https://stackoverflow.com/a/12429953/14930422>

