**When to use abstract class and when to use Interface?**

**Answer :-** The decision to use an abstract class or an interface depends on the specific requirements and design considerations of your program. Here are some guidelines to help you decide when to use each:

Use an abstract class when:

You want to provide a common base implementation or functionality to multiple related classes.

You need to define instance variables or non-static fields that are inherited by the subclasses.

You want to define abstract methods that must be implemented by the subclasses.

You need to provide default implementations for some methods, which can be overridden by subclasses if needed.

You want to restrict the instantiation of a class by making it abstract.

Use an interface when:

You want to define a contract or a set of methods that multiple unrelated classes can implement.

You need to achieve multiple inheritance of type, as a class can implement multiple interfaces but can only extend a single class.

You want to provide a common behavior that can be shared across different classes.

You need to separate the declaration of methods from the implementation details.

You want to define constants that are shared across implementing classes using interface variables, as Java doesn't support multiple inheritance of state.

In Java 8, interfaces gained some additional capabilities that blur the lines between abstract classes and interfaces. With the introduction of default methods (methods with an implementation) in interfaces, it became possible to provide common behavior directly within an interface. However, remember that interfaces still cannot have instance variables, and they are primarily used to define contracts and allow multiple inheritance of type.

It's important to analyze the specific requirements of your program and choose the appropriate approach. In some cases, a combination of abstract classes and interfaces might be the best solution to achieve the desired design and functionality.

**Internerl Working of Hashmap?**

**Answer :-** HashMap is a data structure in Java that provides an efficient way to store and retrieve key-value pairs. It is part of the Java Collections Framework and is implemented using an array of linked lists known as a bucket.

Here's a high-level overview of how HashMap works in Java:

Hashing: When you insert a key-value pair into a HashMap, the key's hash code is calculated using the key's hashCode() method. The hash code is an integer that represents the key and is used to determine the index in the underlying array where the entry will be stored.

Bucket and Index Calculation: The hash code is then transformed into a valid index within the array by performing a bitwise AND operation with the array's length minus one. This index represents the bucket where the entry will be stored. Multiple keys can map to the same bucket due to potential hash code collisions.

Collision Resolution: If two or more keys have the same index after hashing, a linked list is used to handle collisions. Each bucket can store multiple key-value pairs in the form of nodes. If a collision occurs, a new node is added to the existing linked list at that index.

Retrieving Values: When you want to retrieve a value associated with a specific key, the hash code of the key is calculated again. The index is computed using the bitwise AND operation as described earlier. The linked list in the corresponding bucket is traversed, comparing each key in the list to the desired key until a match is found. Once the match is found, the corresponding value is returned.

Performance: HashMap provides constant-time performance for the basic operations (get, put, and remove) on average, assuming a well-distributed set of hash codes. However, if the hash codes are poorly distributed or many collisions occur, the performance can degrade to O(n), where n is the number of elements in the map.

It's worth noting that HashMap does not guarantee the order of its elements. If you need a specific ordering, you can use LinkedHashMap, which maintains the insertion order, or TreeMap, which sorts the entries based on their natural ordering or a custom comparator.

Overall, HashMap provides an efficient way to store and retrieve key-value pairs by leveraging hashing and linked lists to handle collisions.

**Internal working of ArrayList?**

**Answer:-** ArrayList is a dynamic array-based data structure in Java that provides resizable arrays. It is part of the Java Collections Framework and allows for the storage and manipulation of elements in a flexible manner.

Here's a high-level overview of how ArrayList works in Java:

Array-based Structure: ArrayList internally uses a regular Java array to store its elements. When you create an ArrayList, an initial array is created with a default capacity (typically 10). As elements are added or removed, the array is resized dynamically to accommodate the changes.

Dynamic Resizing: If the number of elements exceeds the current capacity of the internal array, ArrayList automatically increases the capacity by creating a new, larger array and copying the existing elements to the new array. This resizing process allows the ArrayList to handle a variable number of elements efficiently.

Adding Elements: When you add an element to an ArrayList using the add() method, the element is appended to the end of the internal array. If the internal array is full, it is resized by creating a new array with a larger capacity. The existing elements are copied to the new array, and the new element is added at the end.

Accessing Elements: You can access elements in an ArrayList using the get() method, which takes an index as an argument. The ArrayList performs a direct array access to retrieve the element at the specified index.

Removing Elements: To remove an element from an ArrayList, you can use the remove() method, which can take either the index of the element to be removed or the object itself. When an element is removed, the subsequent elements are shifted to fill the gap left by the removed element, ensuring a contiguous structure.

Performance: ArrayList provides constant-time performance for adding and removing elements at the end (amortized O(1)), as long as the internal array does not require resizing. However, if the internal array needs to be resized, the operation takes O(n) time, where n is the number of elements in the ArrayList. Random access and retrieval of elements by index have constant-time complexity O(1).

Iteration and Traversal: ArrayList supports iteration using enhanced for loops (for-each loop) or by using an iterator obtained through the iterator() method. Iteration allows you to access each element in the ArrayList sequentially.

ArrayList provides various methods to manipulate and query the elements it contains, such as size(), isEmpty(), contains(), indexOf(), and more.

Remember that ArrayList is not synchronized, meaning it is not thread-safe by default. If you require thread-safe access to an ArrayList, you can use Collections.synchronizedList() to obtain a synchronized version or consider using other thread-safe alternatives like Vector or CopyOnWriteArrayList.

Overall, ArrayList provides a flexible and efficient way to store and manipulate collections of elements by dynamically resizing its underlying array as needed.

**Internal working of LinkedHashMap?**

**Answer: -** LinkedHashMap is a data structure in Java that extends the functionality of HashMap by maintaining a doubly-linked list to preserve the insertion order of elements. It combines the key-value mapping of a HashMap with a linked list to provide predictable iteration order.

Here's a high-level overview of how LinkedHashMap works in Java:

1. Hashing: Like HashMap, LinkedHashMap uses hashing to store and retrieve key-value pairs. Each key's hash code is calculated using the key's hashCode() method to determine the index in the internal array where the entry will be stored.
2. Bucket and Index Calculation: The hash code is transformed into a valid index within the array using the bitwise AND operation as explained earlier. This index represents the bucket where the entry will be stored. Multiple keys can map to the same bucket due to potential hash code collisions.
3. Collision Resolution: If two or more keys have the same index after hashing, LinkedHashMap handles collisions by using a doubly-linked list to maintain the order of the entries. Each bucket in the internal array can store multiple key-value pairs in the form of nodes. If a collision occurs, a new node is added to the linked list at that index, preserving the insertion order.
4. Iteration Order: LinkedHashMap maintains the insertion order of elements, allowing for predictable iteration. The order of elements is determined by the linked list, with nodes being linked in the order in which they were inserted. Iterating over a LinkedHashMap will produce the elements in the same order in which they were added.
5. Performance: LinkedHashMap provides similar performance characteristics to HashMap. The basic operations (get, put, and remove) have constant-time complexity on average (O(1)), assuming a well-distributed set of hash codes. However, the performance can degrade to O(n) in case of poor hash code distribution or many collisions, as the iteration and removal of elements require traversing the linked list.

LinkedHashMap also offers constructors to create instances with access-ordering, where the order of elements is based on the access history. This means that the most recently accessed elements are moved to the end of the linked list, allowing for efficient implementations of LRU (Least Recently Used) caching strategies.

Overall, LinkedHashMap combines the key-value mapping of a HashMap with a linked list to maintain insertion order. It provides predictable iteration order and additional access-ordering capabilities, making it suitable for scenarios where the order of elements is important.

**Internally working of ConcurrentHashMap?**

**Answer: -** ConcurrentHashMap is a thread-safe implementation of the Map interface provided by Java's Concurrent Collections framework. It allows multiple threads to access and modify the map concurrently without the need for external synchronization. ConcurrentHashMap achieves this concurrency by dividing the map into multiple segments, each of which can be accessed independently.

Here's a high-level overview of how ConcurrentHashMap works in Java:

1. Segmentation: ConcurrentHashMap internally maintains an array of segments, where each segment is a separate hash table. The number of segments is determined by the concurrency level specified during construction (default is 16). The purpose of segmentation is to allow concurrent access to different segments, reducing contention between threads.
2. Hashing and Segment Selection: When you perform operations on a ConcurrentHashMap, the key's hash code is used to determine the segment in which the key-value pair will be stored or retrieved. The hash code is divided by the total number of segments, and the remainder is used to select the appropriate segment.
3. Segment Structure: Each segment in ConcurrentHashMap is similar to a regular HashMap, consisting of an array of hash buckets. The buckets store the actual key-value entries. When two or more entries hash to the same bucket, a linked list or a balanced tree (depending on the number of entries) is used to handle collisions within that bucket.
4. Concurrency Control: Each segment in ConcurrentHashMap uses its own lock, allowing multiple threads to access different segments concurrently. This reduces contention and allows for higher concurrency. When a thread accesses a specific segment, it acquires the lock associated with that segment, ensuring that only one thread modifies that segment at a time. Other threads can simultaneously access different segments without blocking each other.
5. Thread-Safe Operations: ConcurrentHashMap provides thread-safe operations for common operations like put, get, and remove. These operations are executed within the context of the corresponding segment's lock, ensuring thread-safety within a segment.
6. Expansion and Resizing: ConcurrentHashMap automatically expands its capacity when the number of elements exceeds a certain threshold. Unlike a regular HashMap, ConcurrentHashMap performs resizing per segment, reducing the impact on concurrency. During resizing, new segments are added, and existing elements are redistributed among the new and existing segments.

ConcurrentHashMap achieves a balance between thread-safety and concurrent access performance. It allows multiple threads to operate on different segments concurrently while ensuring that operations within each segment are thread-safe. This makes it suitable for scenarios where high concurrency is required, such as in multi-threaded applications.

It's important to note that although ConcurrentHashMap provides thread-safe operations, it does not guarantee consistency for compound actions like putIfAbsent() or iteration. For such scenarios, additional synchronization or concurrent utility classes should be used.

How to create immutable class in java

Answer: - To create an immutable class in Java, follow these guidelines:

1. Declare the class as final: By making the class final, you ensure that it cannot be subclassed and its behavior cannot be modified.
2. Declare the fields as private and final: Make the instance variables private to encapsulate the internal state of the class. Additionally, mark them as final to make them unmodifiable once initialized.
3. Don't provide setter methods: Omit any setter methods that allow modifying the state of the class. This prevents external modification of the object's state.
4. Provide getters for accessing the values: Create getter methods for retrieving the values of the instance variables. These methods should only return the values and not provide any means to modify them.
5. Initialize the fields in the constructor: Set the values of the instance variables in the class constructor. Once assigned, the values should remain constant throughout the object's lifetime.
6. Ensure defensive copying: If your class contains mutable reference fields, ensure that you perform a defensive copy of these fields in the constructor. This prevents the original objects from being modified through external references.
7. Make sure any mutable components are not exposed: If your class contains collections or other mutable objects, avoid returning direct references to these objects from the getter methods. Instead, return copies or immutable views of these objects to maintain immutability.
8. Override the equals() and hashCode() methods: Override these methods to provide correct equality comparisons based on the class's state.
9. Make the class Serializable: If the class needs to be serialized, implement the Serializable interface and provide a serialVersionUID to maintain compatibility.
10. Document immutability: Clearly document that the class is immutable in its class-level documentation or by adding appropriate comments.

By following these guidelines, you can create a class that is immutable, meaning its state cannot be changed once created. Immutable classes are beneficial for concurrent programming, thread safety, and reducing complexity in certain scenarios.

It's worth noting that there are libraries, such as Apache Commons Lang and Guava, that provide utility classes to simplify the creation of immutable classes, such as ImmutableMap, ImmutableList, etc. These libraries handle the implementation details and provide immutable variants of common collection classes.

**How to create singleton class in java**

**Answer: -** To create a singleton class in Java, you can follow the following approach using the Singleton design pattern:

1. Declare a private constructor: This ensures that the class cannot be instantiated from outside the class itself.
2. Create a private static instance variable: This will hold the single instance of the class.
3. Create a public static method to access the instance: This method will be responsible for creating and returning the instance of the class. It should check if the instance variable is null and create a new instance if it is. If the instance variable is not null, it should simply return the existing instance.
4. Optionally, implement lazy initialization: By default, the instance will be created when the class is first accessed. However, if you want to defer the instance creation until the getInstance() method is called, you can add lazy initialization logic inside the getInstance() method.

Here's an example implementation of a singleton class:

javaCopy code

public class Singleton {

private static Singleton instance; // Step 2: Private static instance variable

private Singleton() {

// Step 1: Private constructor

}

public static Singleton getInstance() {

if (instance == null) {

// Lazy initialization (optional)

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

In this example, the Singleton class has a private constructor to prevent direct instantiation. The getInstance() method checks if the instance variable is null and creates a new instance if needed. The use of synchronized ensures thread-safety during the lazy initialization process.

To access the singleton instance, you would call Singleton.getInstance().

It's important to note that while the above example provides a basic implementation of a singleton class, it may not be thread-safe in all scenarios. For more complex scenarios, you might need to consider additional techniques like eager initialization, using an enum, or using double-checked locking for thread safety.

**How to create singleton class in java with thread safety**

**Answer: -** To create a thread-safe singleton class in Java, you can use either eager initialization or lazy initialization with double-checked locking. Here's an example of each approach:

1. Eager Initialization:

javaCopy code

public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {

// Private constructor

}

public static Singleton getInstance() {

return instance;

}

}

In this approach, the instance is created eagerly when the class is loaded. This ensures that the instance is thread-safe, as the JVM guarantees that class initialization is thread-safe. However, it may not be memory efficient if the instance is not always needed.

1. Lazy Initialization with Double-Checked Locking:

javaCopy code

public class Singleton {

private static volatile Singleton instance;

private Singleton() {

// Private constructor

}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

In this approach, the instance is created lazily when the getInstance() method is called. The double-checked locking technique is used to ensure thread safety. The use of the volatile keyword ensures that changes to the instance variable are immediately visible to all threads, preventing any potential issues with thread caching.

Note that the double-checked locking approach requires Java 5 or later, as prior to Java 5, the volatile keyword had different semantics.

Both approaches ensure thread safety by either leveraging class initialization or using synchronization to handle concurrent access. Choose the approach that best fits your requirements based on factors such as initialization overhead, memory efficiency, and the likelihood of needing the instance in your application.