Nomura

**1. time complexity of collection?**

**https://www.baeldung.com/java-collections-complexity**

when we talk about time complexity, we refer to Big-O notation.

**add() – takes O(1)** time; however, worst-case scenario, when a new array has to be created and all the elements copied to it, it's O(n)

**add(index, element) –** on average runs in **O(n)** time

**get() –** is always a constant time **O(1)** operation

**remove() –** runs in linear **O(n)** time. We have to iterate the entire array to find the element qualifying for removal.

**indexOf() –** also runs in linear time. It iterates through the internal array and checks each element one by one, so the time complexity for this operation always requires **O(n)** time.

**contains() –** implementation is based on indexOf(), so it'll also run in **O(n)** time.

**adding and removing elements in LinkedList is quite fast.**

**2. volitile keyword?**

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile modifier is **used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory**. Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory.

**3. Difference between final and immutable?**

**Final** means that you can't change the object reference to point to another reference or another object, but you can still mutate its state (by using the setter method).

Where **immutable** means that the object's actual value can't be changed, but you can change its reference to another one.

4. **Arraylist and Linkedlist memory management and intrnal working?**

ArrayList is slow as array manipulation is slower. LinkedList is faster being node based as not much bit shifting required. ArrayList implements only List.

Deletion: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

ArrayList provides constant time for search operation, so **it is better to use ArrayList if searching is more frequent operation than add and remove operation**. The LinkedList provides constant time for add and remove operations. So it is better to use LinkedList for manipulation.

Manipulation with LinkedList is faster than ArrayList because **it uses a doubly linked list, so no bit shifting is required in memory**. 3) An ArrayList class can act as a list only because it implements List only. LinkedList class can act as a list and queue both because it implements List and Deque interfaces.

The **ArrayList always gives O(1) performance in best case or worst-case time complexity**. The HashMap get() method has O(1) time complexity in the best case and O(n) time complexity in worst case. ArrayList has any number of null elements. HashMap allows only one null Key and lots of null values.

LinkedList, ArrayDeque – Queue interface implementations can act as the stack, queue, and dequeue data structures. Generally, **ArrayDeque is faster than LinkedList**.

**5. Difference between string buffer and builder?**

StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.

StringBuffer is less efficient than StringBuilder.

StringBuffer was introduced in Java 1.0

StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.

StringBuilder is more efficient than StringBuffer.

StringBuilder was introduced in Java 1.5

**6. can we throws check exception from static block?**

There is one restriction though, you can't 'throw' an exception in a static block. The Java language doesn't allow this.

A **static block** is a set of statements, which will be executed by the JVM before the execution of the main() method. At the time of class loading if we want to perform any activity we have to define that activity inside a **static block** because this block executes at the**time of class loading**.

**static block** can throw only a **RunTimeException**, or there should be a try and catch block to catch a **checked exception.**

Trying to throw a checked exception from a **static block** is also not possible. We can have a try and catch block in a static block where a checked exception may be thrown from the try block but we have to resolve it within the catch block. We can’t propagate it further using a throw keyword.

**Usually, the stack stores the data that is short-lived. It includes local primitive variables, references of heap objects, and methods in execution. Heap allows dynamic memory allocation, stores the Java objects and JRE classes at the runtime.**

**7. what are the java8 features?**

* forEach() method in Iterable interface.
* default and static methods in Interfaces.
* Functional Interfaces and Lambda Expressions.
* Java Stream API for Bulk Data Operations on Collections.
* Java Time API.
* Collection API improvements.
* Concurrency API improvements.
* Java IO improvements.

**8. Difference between concurrent hashmap and hashmap?**

HashMap is a powerful data structure in Java used to store the key-pair values. It maps a value by its associated key. It allows us to store the null values and null keys. It is a non-synchronized class of Java collection. Whereas, ConcurrentHashMap is introduced as an alternative to the HashMap. The [ConcurrentHashMap](https://www.javatpoint.com/java-concurrenthashmap) is a synchronized collection class.

* As discussed above, the HashMap is a non-synchronized and non-Thread safe, while the ConcurrentHashMap is a synchronized and Thread-safe collection class. Though the ConcurrentHashMap can not match the synchronization level of Hashtable, it performs well for most of the practical cases.
* The HashMap can be synchronized using the **Collection.syncronizedMap;** It returns a collection that is almost equal to Hashtable.
* The synchronized HashMap is less scalable than the ConcurrentHashMap.
* In the multi-threaded environment, The ConcurrentHashMap has improved performance than Synchronized HashMap.
* In the single-threaded environment, The HashMap is slightly better than ConcurrentHashMap.
* In HashMap, if one thread is iterating the object and the other thread wants to modify the objects, we will get a **ConcurrentModificationException** runtime exception. But, in ConcurrentHashMap, one thread can perform modification while the other thread is running.

**9. can we have private methods in abstract class?**

Abstract classes can have private methods. Interfaces can't. Abstract classes can have instance variables (these are inherited by child classes).

If a method of a class is private, you cannot access it outside the current class, not even from the child classes of it. But, incase of an abstract method, you cannot use it from the same class, you need to override it from subclass and use. Therefore, **the abstract method cannot be private**

An Abstract class can have access modifiers like private, protected, and internal with class members. But **abstract members cannot have a private access modifier**.

If you declare a method in a class abstract to use it, you must override this method in the subclass. But, overriding is not possible with static methods. Therefore, **an abstract method cannot be static**.

**Differences Between Abstract Classes and Interfaces**

Abstract classes can have non-abstract methods that have method definitions. This gets inherited. Interfaces do not allow any method definitions.

Abstract classes can have constructors. This may seem a little silly because you can't construct objects from an abstract class. However, when you write child classes, it calls the constructor of the parent class, even if the parent class is abstract.

Interfaces can't have constructors.

Abstract classes can have private methods. Interfaces can't.

Abstract classes can have instance variables (these are inherited by child classes). Interfaces can't.

Finally, a concrete class can only extend one class (abstract or otherwise). However, a concrete class can implement many interfaces. This fact has nothing to do with abstract classes. A class can only have one parent class (although the parent class can have a parent class, and its parent can have a parent class, and so forth), regardless of whether the class is abstract or not.

**10. difference between for and for each loop?**

For Loops executes a block of code until an expression returns false while ForEach loop executed a block of code through the items in object collections.

For loop can execute with object collections or without any object collections while ForEach loop can execute with object collections only.

**11. Thread fail safe iterators in java?**

Iterator implementations that don't throw ConcurrentModificationException when a thread modifies the structure of a collection while another thread or same thread is iterating over it are known as fail-safe iterators as they work on a new copy of the original collection.

The collection classes that are thread-safe in Java are **Stack, Vector, Properties, Hashtable**, etc.

The examples of Fail Safe iterators are **ConcurrentHashMap, CopyOnWriteArrayList**, etc.

An LazyIteratorChain is **an Iterator that wraps a number of Iterators in a lazy manner**. This class makes multiple iterators look like one to the caller.

**12. fail safe?**

**Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.**

Whereas, **Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.**

[Iterators](https://contribute.geeksforgeeks.org/iterators-in-java/) in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw *ConcurrentModificationException* if there is **structural modification** of the collection. Structural modification means adding, removing any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.  
Fail-Safe iterators don’t throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that’s why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

**13. can we override static or final methods?**

NO, we can't override static methods since method overriding relies on dynamic binding at runtime, but static methods are bonded at compile time with static binding.

**No, we cannot override static methods** because method overriding is based on dynamic binding at runtime and the static methods are bonded using static binding at compile time. So, we cannot override static methods. The calling of method depends upon the type of object that calls the static method.

When a method is declared with the final keyword, it is called a final method in java. **We cannot override the final method in java**.

**No, we cannot override private or static methods in Java**. Private methods in Java are not visible to any other class which limits their scope to the class in which they are declared.

**14. String API memory execution?**

The String [constant pool](https://www.baeldung.com/jvm-constant-pool) is a special memory area. **When we declare a String literal, the**[**JVM**](https://www.baeldung.com/jvm-parameters)**creates the object in the pool and stores its reference on the stack.** Before creating each String object in memory, the JVM performs some steps to decrease the memory overhead.

The String constant pool uses a [Hashmap](https://www.baeldung.com/java-hashmap) in its implementation. Each bucket of the Hashmap contains a list of Strings with the same hash code. In earlier versions of Java, the storage area for the pool was a fixed size and could often lead to the “[*Could not reserve enough space for object heap*](https://www.baeldung.com/java-heap-memory-error)” error.

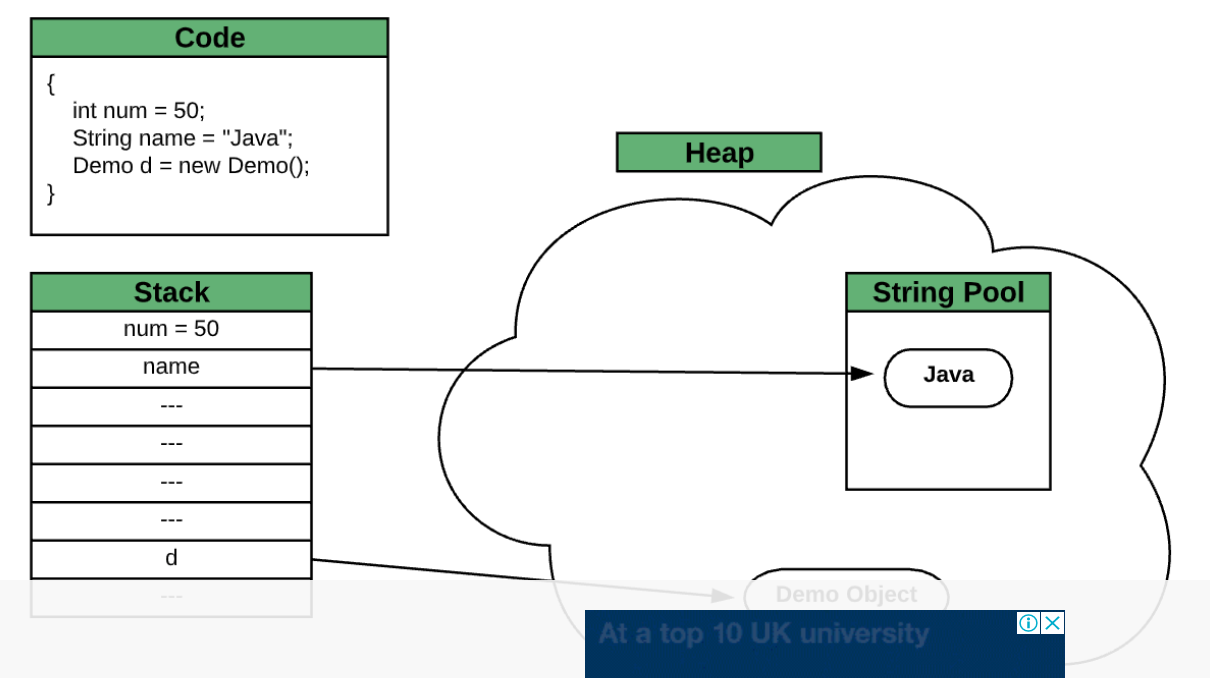
**When the system loads the classes, String literals of all classes go to the application-level pool.** It is because of the fact that equal String literals of different classes have to be the same Object. In these situations, data in the pool should be available to each class without any dependency.

Usually, the stack stores the data that is short-lived. It includes local primitive variables, references of heap objects, and methods in execution. Heap allows dynamic memory allocation, stores the Java objects and JRE classes at the runtime.

The heap allows global access and  data stores in the heap are available to all threads during the lifetime of the application, whereas the data stores on the stack have the private scope and only the owner thread can access them.

The stack stores the data in contiguous memory blocks and permits random access. If a class needs a random String from the pool, it might not be available due to the LIFO (last-in-first-out) rule of the stack. In contrast, the heap allocates the memory dynamically and allows us to access the data in any way.

Let's assume we have a code snippet consisting of different types of variables. The stack will store the value of the int literal and references of String and Demo objects. The value of any object will be stored in the heap, and all the String literals go in the pool inside the heap:

****

**15. Reflection in java and how to achieve it?**

In Java, reflection allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time. There is a class in Java named Class that keeps all the information about objects and classes at runtime. The object of Class can be used to perform reflection.

Reflection is a feature in the Java programming language. It **allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program**. For example, it's possible for a Java class to obtain the names of all its members and display them.

**Field.** **get(Object obj) method** returns the value of the field represented by this Field, on the specified object. The value is automatically wrapped in an object if it has a primitive type.

**16. parent and child class method execution?**

[**https://www.simplilearn.com/tutorials/java-tutorial/overriding-in-java**](https://www.simplilearn.com/tutorials/java-tutorial/overriding-in-java)

The object that is used to trigger a method specifies the variant of the process that is executed. If it implements a method with an object from a parent class, the parent class's version will be used. But if the method is triggered with an object from a subclass, the child class's version will be used.

When the method signature (name and parameters) are the same in the superclass and the child class, it's called overriding. When two or more methods in the same class have the same name but different parameters, it's called overloading.

## 17. Why Is Overriding in Java Useful?

As previously mentioned, overridden methods allow Java to accept polymorphism at runtime. Overridden methods are also another way Java embraces polymorphism's "one application, many methods" aspect.

The most effective object-oriented programming brings to bear on code reuse and robustness is Dynamic Process Execution. The ability to use existing code libraries to call methods on new class instances without re-compiling, while preserving a clean abstract interface, is an incredibly powerful weapon.

 Overridden methods allow one to call methods from any derived [class object](https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects) without identifying the form of the modified super-class.

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

## Advantage of method overloading

Method overloading increases the readability of the program.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

**17. 10m object in arraylist and linkedlist execution?**

**Summary** ArrayList with ArrayDeque are preferable in *many* more use-cases than LinkedList. If you're not sure — just start with ArrayList.

TLDR, in ArrayList accessing an element takes constant time [O(1)] and adding an element takes O(n) time [worst case]. In LinkedList inserting an element takes O(n) time and accessing also takes O(n) time but LinkedList uses more memory than ArrayList.

LinkedList and ArrayList are two different implementations of the List interface. LinkedList implements it with a doubly-linked list. ArrayList implements it with a dynamically re-sizing array.

As with standard linked list and array operations, the various methods will have different algorithmic runtimes.

For [LinkedList<E>](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/LinkedList.html)

* get(int index) is *O(n)* (with *n/4* steps on average), but *O(1)* when index = 0 or index = list.size() - 1 (in this case, you can also use getFirst() and getLast()). **One of the main benefits of** LinkedList<E>
* add(int index, E element) is *O(n)* (with *n/4* steps on average), but *O(1)* when index = 0 or index = list.size() - 1 (in this case, you can also use addFirst() and addLast()/add()). **One of the main benefits of** LinkedList<E>
* remove(int index) is *O(n)* (with *n/4* steps on average), but *O(1)* when index = 0 or index = list.size() - 1 (in this case, you can also use removeFirst() and removeLast()). **One of the main benefits of** LinkedList<E>
* Iterator.remove() is *O(1)*. **One of the main benefits of** LinkedList<E>
* ListIterator.add(E element) is *O(1)*. **One of the main benefits of** LinkedList<E>

Note: Many of the operations need*n/4*steps on average,*constant*number of steps in the best case (e.g. index = 0), and*n/2*steps in worst case (middle of list)

For [ArrayList<E>](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html)

* get(int index) is *O(1)*. **Main benefit of** ArrayList<E>
* add(E element) is *O(1)* amortized, but *O(n)* worst-case since the array must be resized and copied
* add(int index, E element) is *O(n)* (with *n/2* steps on average)
* remove(int index) is *O(n)* (with *n/2* steps on average)
* Iterator.remove() is *O(n)* (with *n/2* steps on average)
* ListIterator.add(E element) is *O(n)* (with *n/2* steps on average)

Note: Many of the operations need*n/2*steps on average,*constant*number of steps in the best case (end of list),*n*steps in the worst case (start of list)

LinkedList<E> allows for constant-time insertions or removals *using iterators*, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but finding a position in the list takes time proportional to the size of the list. Javadoc says *"operations that index into the list will traverse the list from the beginning or the end, whichever is closer"*, so those methods are *O(n)* (*n/4* steps) on average, though *O(1)* for index = 0.

ArrayList<E>, on the other hand, allow fast random read access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap. Also, if you add more elements than the capacity of the underlying array, a new array (1.5 times the size) is allocated, and the old array is copied to the new one, so adding to an ArrayList is *O(n)* in the worst case but constant on average.

So depending on the operations you intend to do, you should choose the implementations accordingly. Iterating over either kind of List is practically equally cheap. (Iterating over an ArrayList is technically faster, but unless you're doing something really performance-sensitive, you shouldn't worry about this -- they're both constants.)

The main benefits of using a LinkedList arise when you re-use existing iterators to insert and remove elements. These operations can then be done in *O(1)* by changing the list locally only. In an array list, the remainder of the array needs to be *moved* (i.e. copied). On the other side, seeking in a LinkedList means following the links in *O(n)* (*n/2* steps) for worst case, whereas in an ArrayList the desired position can be computed mathematically and accessed in *O(1)*.

Another benefit of using a LinkedList arises when you add or remove from the head of the list, since those operations are *O(1)*, while they are *O(n)* for ArrayList. Note that ArrayDeque may be a good alternative to LinkedList for adding and removing from the head, but it is not a List.

Also, if you have large lists, keep in mind that memory usage is also different. Each element of a LinkedList has more overhead since pointers to the next and previous elements are also stored. ArrayLists don't have this overhead. However, ArrayLists take up as much memory as is allocated for the capacity, regardless of whether elements have actually been added.

The default initial capacity of an ArrayList is pretty small (10 from Java 1.4 - 1.8). But since the underlying implementation is an array, the array must be resized if you add a lot of elements. To avoid the high cost of resizing when you know you're going to add a lot of elements, construct the ArrayList with a higher initial capacity.

If the data structures perspective is used to understand the two structures, a LinkedList is basically a sequential data structure which contains a head Node. The Node is a wrapper for two components : a value of type T [accepted through generics] and another reference to the Node linked to it. So, we can assert it is a recursive data structure (a Node contains another Node which has another Node and so on...). Addition of elements takes linear time in LinkedList as stated above.

**18. interface and abstract class, static non static keyword?**

**Is abstract class static or non static?**

If you declare a method in a class abstract to use it, you must override this method in the subclass. But, overriding is not possible with static methods. Therefore, **an abstract method cannot be static**.

**Can we use non static in abstract and interface?**

**No you cannot have non-static variables in an interface**. By default, All the members (methods and fields) of an interface are public. All the methods in an interface are public and abstract (except static and default).

**What is static and non static keywords?**

**Static variables are shared among all instances of a class.** **Non static variables are specific to that instance of a class**. Static variable is like a global variable and is available to all methods. Non static variable is like a local variable and they can be accessed through only instance of a class.

**Can we use non static method in abstract class?**  
You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, **you can declare fields that are not static and final**, and define public, protected, and private concrete methods.

**Can we use static keyword with interface?**

Since static methods don't belong to a particular object, they're not part of the API of the classes implementing the interface; therefore, **they have to be called by using the interface name preceding the method name**. Defining a static method within an interface is identical to defining one in a class.

**Can we override static in interface?**

**You cannot override the static method of the interface**; you can just access them using the name of the interface. If you try to override a static method of an interface by defining a similar method in the implementing interface, it will be considered as another (static) method of the class.

**Can we use static keyword in overloading?**

Can we overload static methods? The answer is '**Yes**'. We can have two or more static methods with the same name, but differences in input parameters.

**Can we make constructor as static?**

**No, we cannot define a static constructor in Java**, If we are trying to define a constructor with the static keyword a compile-time error will occur.

**Can we have constructor in interface?**

**An interface cannot contain a constructor** (as it cannot be used to create objects)

**Can we overload main method?**

The short answer to, can we overload main method in Java is **Yes, we can overload the main() method in Java**. A Java class can have any number of overloaded main() methods. But the very first thing JVM (Java Virtual Machine) seeks is the original main() method, i.e., public static void main(String[] args) to execute.

**Can we override private methods?**

**No, we cannot override private or static methods in Java**. Private methods in Java are not visible to any other class which limits their scope to the class in which they are declared.

**Can we override final methods?**

When a method is declared with the final keyword, it is called a final method in java. **We cannot override the final method in java**.

**Can we declare main method as private?**

**Yes, we can declare the main method as private in Java**. It compiles successfully without any errors but at the runtime, it says that the main method is not public.

**Is it possible to extend final class?**

**A final class cannot extended to create a subclass**. All methods in a final class are implicitly final .