**ArrayList:**

1. ArrayList supports dynamic array that can grow as needed.

2. It can contain Duplicate elements and it also maintains the insertion order.

3. Manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

4. ArrayLists are not synchronized.

5. ArrayList allows random access because it works on the index basis.

**LinkedList**:

It allows null entry.

It is dynamic in nature i.e it allocates memory when required. Therefore insertion and deletion operations can be easily implemented.

It can contain duplicate elements and it is not synchronized.

Reverse Traversing is difficult in linked list.

In LinkedList, manipulation is fast because no shifting needs to be occurred.

**HashSet:**

1. HashSet does not allow duplicates, means it contains unique elements.

2. HashSet is unordered meaning it does not guarantee that the order will remain constant over time.

3. HashSet is backed by Hashtable (actually a HashMap instance).

4. HashSet permits the null elements.

5. HashSet implementation is not synchronized. It can be synchronized externally.

**HashMap:**

HashMap does not maintain order of its element.

It contains values based on the key.

It allows only unique keys.

It is unsynchronized.

Its initial default capacity is 16.

It permits null values and the null key

**Hashtable:**

It contains values based on the key.

It contains unique elements.

It doesn't allow null key or value.

It is synchronized.

The initial default capacity of Hashtable is 11**.**

**ConcurrentHashmap:**

ConcurrentHashMap is thread-safe that is the code can be accessed by single thread at a time .

ConcurrentHashMap synchronizes or locks on the certain portion of the Map . To optimize

the performance of ConcurrentHashMap , Map is divided into different partitions depending upon the Concurrency level . So that we do not need to synchronize the whole Map Object.

ConcurrentHashMap does not allow NULL values . So the key can not be null in ConcurrentHashMap .

**TreeMap:**

It contains only unique elements.

It cannot have a null key but can have multiple null values.

It is non synchronized.

It maintains ascending order.

**Throw:**

The throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

**Throws:**

The throws keyword is used to declare the list of exception that a method may throw during execution of program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword**.**

**Finally**

A finally keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, it lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

**StringBuffer:**

StringBuffer class is used to create a mutable string object. It means, it can be changed after it is created. It represents growable and writable character sequence. It is similar to String class in Java both are used to create string, but stringbuffer object can be changed.

So StringBuffer class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously.

**StringBuilder:**

StringBuilder is identical to StringBuffer except for one important difference that it is not synchronized, which means it is not thread safe. StringBuilder also used for creating string object that is mutable and non synchronized. The StringBuilder class provides no guarantee of synchronization. StringBuffer and StringBuilder both are mutable but if synchronization is not required then it is recommend to use StringBuilder class.

**Runnable Interface:**

By implementing Runnable , multiple threads can share an instance of your work. If you extended Thread, you'd have to create a new instance of your work for each thread.

Separating task as Runnable means we can reuse the task and also has freedom to execute it from different means, since you cannot restart a Thread once it completes. So again "implements Runnable" vs. "extends Thread" for task, implementing Runnable will be the best choice.

**Extends Thread**:

extend Thread class, all methods of Thread class will be inheriting to your class which you may not need. This will cause additional overhead. You can remove this overhead by implem **hashCode()**

The hashcode() is a method returns a hash code for this string. The hashCode() is used for bucketing in Hash implementations like HashMap, HashTable, HashSet, etc. The value received from hashCode() is used as the bucket number for storing elements of the set/map. This bucket number is the address of the element inside the set/map.

**equals()**

This particular method is used to make equal comparison between two objects. There are two types of comparisons in Java. One is using "==" operator and another is "equals()".enting Runnable interface .

**CheckedExceptions**

Checked exceptions are checked at compile time by the JVM(Java Virtual Machine) and its related to resources(files/db/stream/socket etc). The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException , SQLException etc.

**UncheckedExcepton**

The classes that extend RuntimeException are known as unchecked exceptions. Unchecked exceptions are purely programmatic errors, such as logic errors or improper use of an API, null data or even failures in business logic can lead to runtime exceptions .

**Comparable:**

Comparable Interface : Comparable is an public interfaces which is used to impose an natural ordering (if numbers then 1,2,3 or in alphabetical order 'a','b','c' ) of the class that implements it.

Now here the total ordering defines as the natural ordering which means in JVM that when we compare two objects using the comparable interfaces they are actually compared through their ASCII values which is the natural ordering. This means that the comparable by default uses the sorting technique of JVM i.e. Of sorting by the ASCII values.Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).

**Comparator:**

if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface

**ThreadPoolExecutor:**

ThreadPoolExecutor class implements Executor and ExecutorService interface. Task creation and its execution are separated by ThreadPoolExecutor. You only need to implement the Runnable objects and execute them at the executor. ThreadPoolExecutor takes care of instantiation, running, and execution with necessary threads.

**Java8 features:**

1. Functional Interface: Each functional interface has a single abstract method, called the functional method, implementation can be provided using the lambda expressions.

2. Lambda Expressions: It is a feature derived from functional programming. It is a function that does not belong to any class.

3. Optional: Instead of using null values Optional class is used for representing Optional values.

4. Stream API

5. Spliterator

6. Method References

7. New Date and Time API.

**Abstract Class:**

A class which is declared using abstract keyword known as abstract class. An abstract class may or may not have abstract methods. We cannot create object of abstract class.

It is used to achieve abstraction but it does not provide 100% abstraction because it can have concrete methods.

An abstract class must be declared with an abstract keyword.

It can have abstract and non-abstract methods.

It cannot be instantiated.

It is used for abstraction.

**Interface**:

Interface is a concept which is used to achieve abstraction in Java. This is the only way by which we can achieve full abstraction. Interfaces are syntactically similar to classes, but you cannot create instance of an Interface and their methods are declared without any body. It can have When you create an interface it defines what a class can do without saying anything about how the class will do it.

**Final Class**

A class can also be declared as final. A class declared as final cannot be inherited. The String class in java.lang package is an example of a final class. We can create our own final class so that no other class can inherit it.

**Synchronized Block:**

If want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it. It is capable to make any part of the object and method synchronized**.**

**Synchronized Method:**

When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution

A synchronized method in Java is very slow and can degrade performance.

**Transient**

While serializing an object, if we don't want certain data member of the object to be serialized we can mention it transient. transient keyword will prevent that data member from being serialized. Making a data member transient will prevent its serialization.

**Volatile**

Volatile modifier tells to the compiler that the volatile variable can be changed unexpectedly by other parts of a program. Volatile variables are used in case of multi-threading program. volatile keyword cannot be used with a method or a class. It can be only used with a variable.

**Wait,notify,notifyall**

Java provide benefits of avoiding thread pooling using inter-thread communication. The wait(), notify(), and notifyAll() methods of Object class are used for this purpose. These method are implemented as final methods in Object, so that all classes have them. All the three method can be called only from within a synchronized context

wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

notify() wakes up a thread that called wait() on same object.

notifyAll() wakes up all the thread that called wait() on same object.

**Bootstrap Class Loader:**

It loads standard JDK class files from rt.jar and other core classes. It is a parent of all class loaders. It doesn't have any parent. When we call String.class.getClassLoader() it returns null, and any code based on it throws NullPointerException. It is also called Primordial ClassLoader. It loads class files from jre/lib/rt.jar. For example, java.lang package class.

**Extensions Class Loader:**

It delegates class loading request to its parent. If the loading of a class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory as java.ext.dirs. It is implemented by sun.misc.Launcher$ExtClassLoader in JVM.

**System Class Loader:**

It loads application specific classes from the CLASSPATH environment variable. It can be set while invoking program using -cp or classpath command line options. It is a child of Extension ClassLoader. It is implemented by sun.misc.Launcher$AppClassLoader class. All Java ClassLoader implements java.lang.ClassLoader.

**Functional Interface:**

A functional interface is a concept that was introduced in Java 8. An interface that has the only a single abstract method and marked with @FunctionalInterface annotation is called functional interface. The functional interface is used to support the functional programming approach, lambda expression, and method reference as well. A functional interface is also known as SAM interface because of the Single Abstract Method (SAM). Let's see how to create a functional interface, although Java provides various built-in functional interfaces that we have summarized in the table at the end of the topic.

**Java 8 features:**

Java Lambda Expression

Java Reflection API Improvement

Java Base64 Class

Java Collectors Class

Java Parallel Array Sorting

Java Method Reference

Java forEach Method

Java StringJoiner Class

Java Interface Default and Static Methods

Java Functional Interface

Java Optional Class

Java Stream API

Java Type Inference

Java Date and Time API

Java Predicate

**LamdaExpression**

Lambda expression is a feature of Java language which was introduced in Java 8 version. It is a function that has no name and uses a functional approach to execute code. the lambda expression is also known as an anonymous function.

It is designed to provide the implementation of a functional interface. An interface that has only a single abstract method is known as a functional interface. Java provides an annotation @FunctionalInterface, which is used to declare an interface as a functional interface

**Advantage with lambda expression**

The body of a lambda expression can have one or more statements.

Curly brackets are optional if there is a single statement.

The return statement is optional, use only if the method signature has a return type.

We can pass zero, one, or more parameters to a lambda expression.

The type of parameters can be explicitly declared or it can be inferred from the context.

When there is a single parameter, it is not mandatory to use parentheses. Parentheses are optional.

**Consumer interface:**

Consumer can be used in all contexts where an object needs to be consumed,i.e. taken as input, and some operation is to be performed on the object without returning any result. Common example of such an operation is printing where an object is taken as input to the printing function and the value of the object is printed( we will expand upon the printing example in more detail below when understanding how to use Consumer interface).

Since Consumer is a functional interface, hence it can be used as the assignment target for a lambda expression or a method reference.

**Supplier Interface**

which has been introduced since Java 8, to implement functional programming in Java. It represents a function which does not take in any argument but produces a value of type T.

Hence this functional interface takes in only one generic namely:-

T: denotes the type of the result

The lambda expression assigned to an object of Supplier type is used to define its get() which eventually produces a value. Suppliers are useful when we don’t need to supply any value and obtain a result at the same time.

The Supplier interface consists of only one function:

1. get()

This method does not take in any argument but produces a value of type T.

**Difference between Intermediate and Terminal Operations**

Intermediate Operation- These operations are used to pipeline other methods and to transform into the other streams. They don’t produce results because these operation does not invoke until the terminal operation gets executed. Below are the examples −

sorted(Comparator<T>)

peek(Consumer<T>)

distinct()

Terminal operations - These operations are used to produce results. They can’t be used for chaining the other methods. Below are the examples −

forEach

count

toArray