# Assignment 03 Application Design: Patterns and Frameworks 44642

**1. What are generics?**

Generics are parameterized types. The idea is to allow type (Integer, String... etc., and user-defined types) to be passed as a parameter to methods, classes, and interfaces. Generics are used to create classes that work with different data types. A generic entity is a class, interface, or method that operates on a parameterized type.

**package** Q1;

**public** **class** Driver {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

// instance of Integer type

Test iObj = **new** Test<Integer>(15);

System.***out***.println(iObj.getObject());

// instance of String type

Test<String> sObj = **new** Test<String>("Murali Krishna");

System.***out***.println(sObj.getObject());

}

}

**package** Q1;

**public** **class** Test<G> {

// Java program to show working of user defined

// Generic classes

G obj;

Test(G obj) {

**this**.obj = obj;

} // constructor

**public** G getObject() {

**return** **this**.obj;

}

}

// B class to test above

**2. Can we change the scope of the overridden method in the subclass for private, public, default, and protected? Explain how it can be changed for each scope.**

Yes, we can change the scope of the overridden method in the subclass, but we cannot decrease the method's accessibility.

* Private can be changed to public, protected, or default.
* The protected setting can be changed to public or default.
* The default setting can be changed to the public.
* The public will remain public.

**3. What is the covariant return type?**

The covariant return type is the overriding method return type. It only works with non-primitive return types.

The co-variant return type is based on the [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle" \t "_blank).

**package** Q3;

**class** A{

A get(){**return** **this**;}

}

**package** Q3;

**class** B1 **extends** A{

@Override

B1 get(){**return** **this**;}

**void** message(){System.***out***.println("welcome to covariant return type");}

**public** **static** **void** main(String args[]){

**new** B1().get().message();

}

}

**4. Can we override the static and private methods? Why?**

In Java, we cannot override private or static methods because private methods are only visible within the class and cannot be seen outside of the class, and static methods are bonded at compile time using static binding, which we cannot override.

**package** Q4;

**public** **class** OverloadStaticMethodExample {

//static method

**public** **static** **void** sum1(**int** a, **int** b)

{

**int** c=a+b;

System.***out***.println("The sum is: "+c);

}

//non-static method

**public** **static** **void** sum(**int** a, **int** b)

{

**int** c=a+b;

System.***out***.println("The sum is: "+c);

}

//main method

**public** **static** **void** main(String args[])

{

//calling static method by using the class name

OverloadStaticMethodExample.*sum*(12, 90);

}

}

**5. Difference between String Buffer and StringBuilder?**

|  |  |  |
| --- | --- | --- |
| 1) | String Buffer is synchronized i.e., thread-safe. It means two threads can't call the methods of String Buffer simultaneously. | StringBuilder is non-synchronized i.e., not thread-safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | String Buffer is less efficient than StringBuilder. | StringBuilder is more efficient than String Buffer. |
| 3) | String Buffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |

**package** Q5;

**public** **class** BufferTest {

//Java Program to demonstrate the use of StringBuffer class.

**public** **static** **void** main(String[] args){

StringBuffer buffer=**new** StringBuffer("hello");

buffer.append("java");

System.***out***.println(buffer);

}

}

**package** Q5;

//Java Program to demonstrate the use of StringBuilder class.

**public** **class** BuilderTest{

**public** **static** **void** main(String[] args){

StringBuilder builder=**new** StringBuilder("hello");

builder.append("java");

System.***out***.println(builder);

}

}

**6. Difference between String class and a String Buffer?**

|  |  |  |
| --- | --- | --- |
| 1) | The String class is immutable. | The String Buffer class is mutable. |
| 2) | A string is slow and consumes more memory when we concatenate too many strings because every time it creates a new instance. | String Buffer is fast and consumes less memory when we concatenate t strings. |
| 3) | The string class overrides the equals() method of the Object class. So you can compare the contents of two strings by the equals() method. | String Buffer class doesn't override the equals() method of an Object class. |
| 4) | The string class is slower while performing concatenation operations. | String Buffer class is faster while performing concatenation operations. |
| 5) | String class uses String constant pool. | String Buffer uses Heap memory |

**package** Q6;

**public** **class** ConcatTest{

**public** **static** String concatWithString() {

String t = "Java";

**for** (**int** i=0; i<10000; i++){

t = t + "Tpoint";

}

**return** t;

}

**public** **static** String concatWithStringBuffer(){

StringBuffer sb = **new** StringBuffer("Java");

**for** (**int** i=0; i<10000; i++){

sb.append("Tpoint");

}

**return** sb.toString();

}

**public** **static** **void** main(String[] args){

**long** startTime = System.*currentTimeMillis*();

*concatWithString*();

System.***out***.println("Time taken by Concating with String: "+(System.*currentTimeMillis*()-startTime)+"ms");

startTime = System.*currentTimeMillis*();

*concatWithStringBuffer*();

System.***out***.println("Time taken by Concating with StringBuffer: "+(System.*currentTimeMillis*()-startTime)+"ms");

}

}

**7. Can we declare the constructor as final?**

A constructor in Java cannot be abstract, final, static, native, or strict. As a result, Java does not have a static constructor.

Because a static constructor is used to initialize static data, the specified task will only be executed once throughout the program. When the first instance is created or any static member is referenced, a static constructor is usually called automatically. The static constructor is declared explicitly by using the static keyword. The static constructor, on the other hand, is not permitted in Java.

The static constructor has the following key features:

It accepts no parameters or access modifiers.

A class can only have one static constructor.

It prohibits inheritance and overloading.

It is triggered automatically and cannot be accessed directly or explicitly.

**package** Q7;

**public** **class** StaticConstructorDemo

{

//Declaring a static constructor

**public** StaticConstructorDemo()

{

System.***out***.println("Static Constructor of the class");

}

**public** **static** **void** main(String args[])

{

// Creating a constructor object to invoke it

StaticConstructorDemo obj = **new** StaticConstructorDemo();

}

}

**8. Can we try without a catch block in java?**

Yes, it is possible to have a try block without a catch block by using the final block, which will execute even if the try block fails.

**package** Q8;

**public** **class** TryCatchExample {

**public** **static** **void** main(String[] args) {

**try**

{

**int** data=50/0; //may throw exception

}

//handling the exception

**catch**(ArithmeticException e)

{

System.***out***.println(e);

}

System.***out***.println("rest of the code");

}

}

**9. What is the try with the resource?**

Java introduced a **try-with-resource** feature in Java 7 that helps to close resources automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network, etc) explicitly, try-with-resource close that automatically by using the AutoClosable interface.

In Java 7, try-with-resources has a limitation that requires resources to declare locally within its block.

**10. Can we modify the throws clause of the superclass method while overriding it in the subclass?**

Yes, it is possible to modify the superclass method throws when overriding it in the subclass.

If the superclass method does not declare an exception, the overridden method of the subclass cannot declare the checked exception, but it can declare the unchecked exception.

If the superclass method declares an exception, the subclass overridden method can declare the same exception, a subclass exception, or no exception, but not the parent exception.

**11. What are an association, aggregation, and composition in UML?**

**Association:** Association can be represented by a line between these classes **with an arrow indicating the navigation direction**. In case an arrow is on both sides, the association is known as a bidirectional association.

A single student can associate with multiple teachers:

Association multiplicity example 1

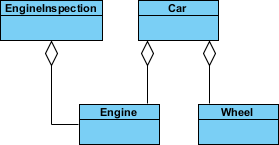
We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

Association multiplicity example 3

**Aggregation:** Aggregation implies a relationship where the child can exist independently of the parent.

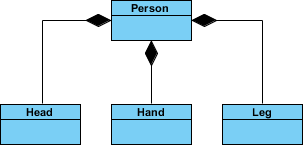
Example: It's important to note that the aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when the parent deleted all its children are being deleted as a result) between the two. Quite the opposite! The aggregation link is usually used to stress the point that the Class A instance is not the exclusive container of the Class B instance, as, in fact, the same Class B instance has another container/s.

To sum it up association is a very generic term used to represent when one class used the functionalities provided by another class. We say it's a composition if one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object. If it can then it is called Aggregation.



**Composition:** Composition implies a relationship where the child cannot exist independently of the parent.

Example: We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



**12. Difference between final, finally, and finalize()?**

**Final:**

final is the keyword and access modifier that is used to apply restrictions on a class, method,o r variable.

The final keyword is used with the classes, methods, and variables.

(1) Once declared, the final variable becomes constant and cannot be modified.  
(2) final method cannot be overridden by a subclass.  
(3) final class cannot be inherited.

A final method is executed only when we call it.

**package** Q12;

**public** **class** FinalExampleTest {

//declaring final variable

**final** **int** age = 18;

**void** display() {

// reassigning value to age variable

// gives compile time error

age = 55;

}

**public** **static** **void** main(String[] args) {

FinalExampleTest obj = **new** FinalExampleTest();

// gives compile time error

obj.display();

}

}

**Finally:**

finally, is the block in Java Exception Handling to execute the important code whether the exception occurs or not.

Finally, a block is always related to the try-and-catch block in exception handling.

(1) finally block runs the important code even if an exception occurs or not.  
(2) finally block cleans up all the resources used in try block.

Finally block is executed as soon as the try-catch block is executed.

**package** Q12;

**public** **class** FinallyExample {

**public** **static** **void** main(String args[]){

**try** {

System.***out***.println("Inside try block");

// below code throws divide by zero exception

**int** data=25/0;

System.***out***.println(data);

}

// handles the Arithmetic Exception / Divide by zero exception

**catch** (ArithmeticException e){

System.***out***.println("Exception handled");

System.***out***.println(e);

}

// executes regardless of exception occurred or not

**finally** {

System.***out***.println("finally block is always executed");

}

System.***out***.println("rest of the code...");

}

}

**Finalize:**

finalize is the method in Java that is used to perform clean-up processing just before an object is a garbage collected.

finalize() method is used with the objects.

finalize method performs the cleaning activities with respect to the object before its destruction.

finalize method is executed just before the object is destroyed.

**package** Q12;

**public** **class** FinalizeExample {

**public** **static** **void** main(String[] args)

{

FinalizeExample obj = **new** FinalizeExample();

// printing the hashcode

System.***out***.println("Hashcode is: " + obj.hashCode());

obj = **null**;

// calling the garbage collector using gc()

System.*gc*();

System.***out***.println("End of the garbage collection");

}

// defining the finalize method

**protected** **void** finalize()

{

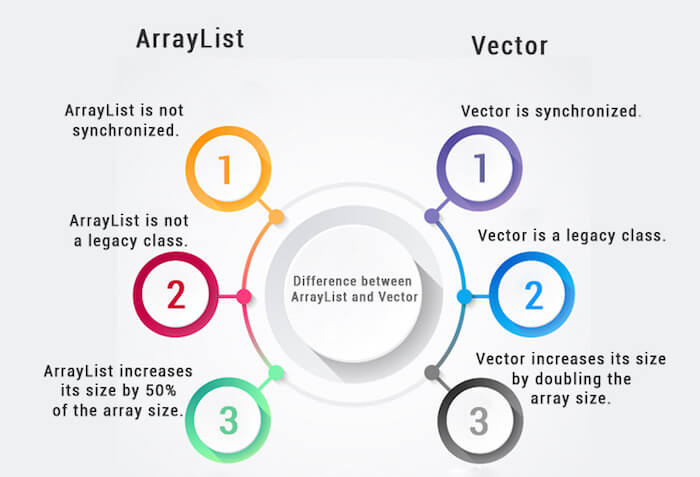
System.***out***.println("Called the finalize() method");

}

}

**13. Difference between ArrayList and Vector?**

|  |  |
| --- | --- |
| ArrayList | Vector |
| 1) ArrayList is not synchronized. | The vector is synchronized. |
| 2) ArrayList increments 50% of the current array size if the number of elements exceeds its capacity. | Vector increments of 100% mean double the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is not a legacy class. It is introduced in JDK 1.2. | Vector is a legacy class. |
| 4) ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until the current thread releases the lock of the object. |
| 5) ArrayList uses the Iterator interface to traverse the elements. | A Vector can use the Iterator interface or Enumeration interface to traverse the elements. |



**package** Q13;

**import** java.util.\*;

**class** TestArrayList21{

**public** **static** **void** main(String args[]){

List<String> al=**new** ArrayList<String>();//creating arraylist

al.add("Design Patterns");//adding object in arraylist

al.add("Web Apps");

al.add("ADB");

al.add("IOS");

//traversing elements using Iterator

Iterator itr=al.iterator();

**while**(itr.hasNext()){

System.***out***.println(itr.next());

}

}

}

**package** Q13;

**import** java.util.\*;

**class** TestVector1{

**public** **static** **void** main(String args[]){

Vector<String> v=**new** Vector<String>();//creating vector

v.add("Murali");//method of Collection

v.addElement("Krishna");//method of Vector

v.addElement("Kethavath");

//traversing elements using Enumeration

Enumeration e=v.elements();

**while**(e.hasMoreElements()){

System.***out***.println(e.nextElement());

}

}

}

**14. What are the different ways to make ArrayList methods synchronized**

They are two ways to create the ArrayList synchronized. They are:

Using collections.synchronizedList() method and using copyonwriteArrayList

We can use Collections.synchronizedList(List<T>) method to synchronize collections in java. The synchronizedList(List<T>) method is used to return a synchronized (thread-safe) list backed by the specified list.

**package** Q14;

**import** java.util.\*;

**public** **class** SyncronizeArrayList {

**public** **static** **void** main(String args[]) {

// Non Synchronized ArrayList

List<String> fruitList = **new** ArrayList<String>();

fruitList.add("Mango");

fruitList.add("Banana");

fruitList.add("Apple");

fruitList.add("Strawberry");

fruitList.add("Pineapple");

// Synchronizing ArrayList in Java

List<String> furitList = Collections.*synchronizedList*(fruitList);

// we must use synchronize block to avoid non-deterministic behavior

**synchronized** (fruitList) {

Iterator<String> itr = fruitList.iterator();

**while** (itr.hasNext()) {

System.***out***.println(itr.next());

}

}

}

}

**15. Difference between Hash Map and Hash Table?**

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 1) HashMap is non-synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code. | The hashtable is synchronized. It is thread-safe and can be shared with many threads. |
| 2) HashMap allows one null key and multiple null values. | Hashtable doesn't allow any null key or value. |
| 3) HashMap is a new class introduced in JDK 1.2. | Hashtable is a legacy class. |
| 4) HashMap is fast. | Hashtable is slow. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is traversed by Iterator. | Hashtable is traversed by Enumerator and Iterator. |
| 7) Iterator in HashMap is fail-fast. | The enumerator in Hashtable is not fail-fast. |
| 8) HashMap inherits AbstractMap class. | Hashtable inherits Dictionary class. |

HashMap and Hashtable both use almost the same set of the library but have helped boost the key and value pair lists when it comes to maintaining them. The data structure has also helped the collection framework by providing developers leverage to play around with a huge set of elements defined in terms of keys and values.

**16. In Java 8, explain how Hasp Map internally works?**

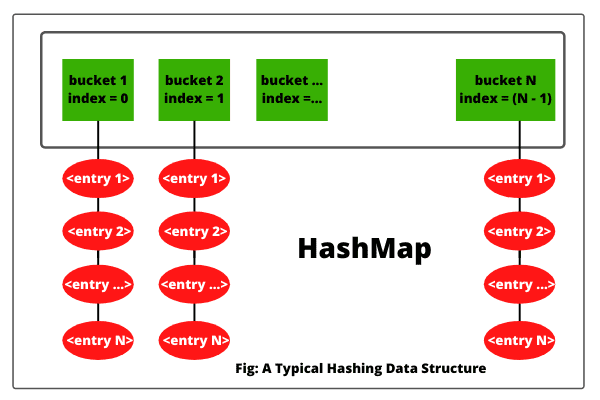
HashMap in Java is basically an array of buckets (also known as a bucket table of HashMap) where each bucket uses a linked list to hold elements. A linked list is a list of nodes where each node contains a key-value pair.

In simple words, a bucket is a linked list of nodes where each node is an object of class Node<K,V>. The key of the node is used to obtain the hash value and this hash value is used to find the bucket from Bucket Table.

HashMap works on the principle of hashing data structure or technique that uses an object’s hashcode to place that object inside the map.

Hashing involves Bucket, Hash function (hashCode() method), and Hash value. It provides the best time complexity of O(1) for the insertion and retrieval of objects.

Therefore, it is the best-suited data structure for storing key-value pairs that later on can be retrieved in minimum time.



Load Factor and Initial Capacity are two important factors that plays important role in the internal working of HashMap in Java.

**Initial Capacity** is a measure of the number of buckets or size of bucket array internally by HashMap at the time of the creation of HashMap.

**Load Factor** is a factor that is internally used by HashMap to determine when the size of the Bucket array requires to be increased. By default, it is 0.75.

**17. Difference between fail fast and fail-safe iterator?**

**Fail-Fast Iterator :**

These iterators throw ConcurrentModificationException if a collection is modified while iterating over it.

They use the original collection to traverse over the elements of the collection.

These iterators don’t require extra memory.

Ex : Iterators returned by ArrayList, Vector, HashMap.

**package** Q17;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**public** **class** FailFast {

**public** **static** **void** main(String[] args)

{

Map<String, String> cityCode = **new** HashMap<String, String>();

cityCode.put("Delhi", "India");

cityCode.put("Moscow", "Russia");

cityCode.put("New York", "USA");

Iterator<String> iterator = cityCode.keySet().iterator();

**while** (iterator.hasNext()) {

System.***out***.println(cityCode.get(iterator.next()));

// adding an element to Map

// exception will be thrown on next call

// of next() method.

cityCode.put("Istanbul", "Turkey");

}

}

}

**Fail Safe Iterator:**

There is no term fail-safe given in many places as Java SE specifications do not use this term. I am using this term to demonstrate the difference between Fail Fast and Non-Fail Fast Iterator. These iterators make a copy of the internal collection (object array) and iterate over the copied collection. Any structural modification done to the iterator affects the copied collection, not the original collection. So, the original collection remains structurally unchanged. 

Fail-safe iterators allow modifications of a collection while iterating over it.

These iterators don’t throw any Exceptions if a collection is modified while iterating over it.

They use copies of the original collection to traverse over the elements of the collection.

These iterators require extra memory for cloning of collection. Ex: ConcurrentHashMap, CopyOnWriteArrayList.

package Q17;

//Java code to illustrate

//Fail Safe Iterator in Java

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.Iterator;

class FailSafe {

public static void main(String args[])

{

CopyOnWriteArrayList<Integer> list

= new CopyOnWriteArrayList<Integer>(new Integer[] { 1, 3, 5, 8 });

Iterator itr = list.iterator();

while (itr.hasNext()) {

Integer no = (Integer)itr.next();

System.out.println(no);

if (no == 8)

// This will not print,

// hence it has created separate copy

list.add(14);

}

}

}

**18. Can we start the thread twice?**

No. After starting a thread, it can never be started again. If you do so, an *IllegalThreadStateException* is thrown. In such case, the thread will run once but for the second time, it will throw an exception.

**package** Q18;

**public** **class** TestThreadTwice1 **extends** Thread{

**public** **void** run(){

System.***out***.println("running...");

}

**public** **static** **void** main(String args[]){

TestThreadTwice1 t1=**new** TestThreadTwice1();

t1.start();

t1.start();

}

}

**19. What are the different ways to create a thread in java? Which one is preferred?**

There are two ways to create a thread:

* By extending the Thread class
* By implementing a Runnable interface.

The Thread class provides constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interfaces have only one method named run().

**package** Q19;

**class** Multi **extends** Thread{

**public** **void** run(){

System.***out***.println("thread is running...");

}

**public** **static** **void** main(String args[]){

Multi t1=**new** Multi();

t1.start();

}

}s

### **Commonly used Constructors of Thread class:**

* Thread()
* Thread(String name)
* Thread(Runnable r)

public void run(): is used to perform action for a thread.

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

* A new thread starts(with new callstack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.

**20. What are the different states a thread will go through?**

The thread goes through different states they are new, runnable, blocked, waiting, time waiting , and terminated.

Diagram

Description automatically generated

**21. What is Serialization? How do we achieve it?**

**Serialization in Java** is a mechanism of *writing the state of an object into a byte stream*. It is mainly used in Hibernate, RMI, JPA, EJB, and JMS technologies.

The reverse operation of serialization is called *deserialization* where the byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream*class, and for deserialization, we call the **readObject()** method of *ObjectInputStream* class.



## java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

The Serializable interface must be implemented by the class whose object needs to be persisted.

The String class and all the wrapper classes implement the java.io.Serializable interface by default.

**22. What is immutable class? Is String class immutable?**

Immutable class means once we create a object we cannot change the content of it. In java all wrapper classes like integer, Boolean , Byte, Short and string class are immutable.

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.



As you can see in the above figure that two objects are created but *s* reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

**23. Do immutable classes thread safe? If yes then how?**

* Objects of an immutable class are called immutable objects. While dealing with immutable objects, we are not allowed to change an object’s state after its creation. Whenever the state of an object is changed, we get a new object.
* Immutable classes do not provide any methods for changing their content.
* In Immutable classes, only getter methods are available and not setter methods.

24. Can we call the garbage collector explicitly? Will it trigger the garbage collector?

When there are no more references to an object, the object is finalized and when the **Garbage Collection** starts these finalized objects get collected this will done automatically by the **JVM**.

We can call the **Garbage Collection**explicitly in two ways

* **System.gc() method**
* **Runtime.gc() method**

The **java.lang.Runtime.freeMemory()** method returns the amount of free memory in the **Java Virtual Machine (JVM)**. Calling the **gc()**method may result in increasing the value returned by the **freeMemory**.

Some of the important features of java are:

1. forEach() method in Iterable interface -> when we need to traverse through the collection we need to create iterator method who has logic in the loop for each of the elements in the collection

**Example:**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Consumer;

**public** **class** foreach {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

List<String> data = **new** ArrayList<>();

data.add("New Delhi");

data.add("New York");

data.add("Mumbai");

data.add("London");

data.forEach(**new** Consumer<String>() {

@Override

**public** **void** accept(String t)

{

System.***out***.println(t);

}

});

}

}

1. default and static methods in Interfaces -> default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility. A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods. Static method in interface are part of the interface class can’t implement or override it whereas class can override the default method.

**Example:**

**Interface:**

**public** **interface** defaultstaticinterface {

**default** **void** show() {

System.***out***.println("Hello world");

}

**static** **void** display() {

System.***out***.println("in the interface");

}

}

**Driver class**

**public** **class** defaultstatic **implements** defaultstaticinterface {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

defaultstaticinterface.*display*();

defaultstatic defaultStaticExampleClass = **new** defaultstatic();

// Call default method on Class

defaultStaticExampleClass.show();

}

}

1. Functional Interfaces and Lambda Expressions -> A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda Expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable, ActionListener,Comparable** are some of the examples of functional interfaces.

Example:

**public** **class** Functionallambda {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

// lambda expression to create the object

**new** Thread(() -> {

System.***out***.println("New thread created");

}).start();

}

}

1. Java Stream API for Bulk Data Operations on Collections -> A new java.util.stream has been added in Java 8 to perform filter/map/reduce like operations with the collection. Stream API will allow sequential as well as parallel execution. This is one of the best features for me because I work a lot with Collections and usually with Big Data, we need to filter out them based on some conditions. Collection interface has been extended with stream() and parallelStream() default methods to get the Stream for sequential and parallel execution.
2. Java Time API -> Each date time instance is composed of fields that are conveniently made available by the APIs. For lower level access to the fields refer to the java.time.temporal package. Each class includes support for printing and parsing all manner of dates and times. Refer to the java.time.format package for customization options.
3. Collection API improvements -> we have some new method for collection api improvements 1. Iterator default method foreachRemaining(action) to perform given action for each remaining element until all elements have been processed or it thows an exception 2. Collection default method removeif to remove all the elements of the collections 3. Performance improvement for hashMap class with key collisions
4. Concurrency API improvements -> some important concurrent API improvements they are compute(), foreach(), foreachentry(), foreachkey(), foreachvalue(), merge(), reduce(), search().
5. Java IO improvements -> Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.Files.lines(Path path) that reads all lines from a file as a Stream. Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file. BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

26. How to make a pure singleton?

The purpose of the singleton object is to control the object creation limiting the objects limiting to only one. It allows only one entry point to create a new instance of the class

**Example:**

**package** Ques26;

**public** **class** pureSingleton {

**private** **static** **volatile** pureSingleton *sSoleInstance* = **new** pureSingleton();

//private constructor.

**private** pureSingleton(){}

**public** **static** pureSingleton getInstance() {

**return** *sSoleInstance*;

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

pureSingleton instance1 = pureSingleton.*getInstance*();

//Instance 2

pureSingleton instance2 = pureSingleton.*getInstance*();

//now lets check the hash key.

System.***out***.println("Instance 1 hash:" + instance1.hashCode());

System.***out***.println("Instance 2 hash:" + instance2.hashCode());

}

}

27. How to make a singleton synchronized?

Synchronization prevents a block of code to be executed by more than one thread at the same time. To improve our Singleton pattern I have just added synchronized keyword in method declaration. In the following example only one thread can enter the getInstance() method and execute code at the time.