Q.1 What are Lambda Expressions? Explain syntax and use of Lambda expressions with a suitable program.

Lambdas Expression:

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- It provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface:

- Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface.
- Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

Why use Lambda Expression:

- To provide the implementation of Functional interface.
- Less coding.

Java Lambda Expression Syntax:

(argument-list) -> {body}

Java lambda expression is consisted of three components:

1) Argument-list:

- It can be empty or non-empty as well.

2) Arrow-token:

- It is used to link arguments-list and body of expression.

3) Body:

- It contains expressions and statements for lambda expression.

Java Lambda Expression Example:

Now, we are implementing the above example with the help of lambda expression.

```
@FunctionalInterface            //It is optional
interface Drawable
{
   public void draw();
}
```

```
public class LambdaExpressionExample2
{
   public static void main(String[] args)
{
       int width=10;

      Drawable d2=()->              //with lambda

       {
          System.out.println("Drawing  "+width);
        };
      d2.draw();
  }
}
```

Output:
Drawing 10

Without Lambda Expression

```
interface Drawable
{
   public void draw();
}

public class LambdaExpressionExample
{
   public static void main(String[] args)
{
       int width=10;

      //without lambda, Drawable implementation using anonymous class
      Drawable d=new Drawable()
       {
         public void draw()
            {
                     System.out.println("Drawing  "+width);
            }
       };
       d.draw();
   }
}
```

Output:
Drawing 10

Q.2 What is Generics in java?

Generics in Java:

- Generics was first introduced in Java5. Now it is one of the most profound feature of java programming language.
- Generic programming enables the programmer to create classes, interfaces and methods in which type of data is specified as a parameter.
- It provides a facility to write an algorithm independent of any specific type of data.
- Generics also provide type safety. Type safety means ensuring that an operation is being performed on the right type of data before executing that operation.
- Using Generics, it has become possible to create a single class, interface or method that automatically works with all types of data (Integer, String, Float etc.). It has expanded the ability to reuse the code safely and easily.
- Before Generics was introduced, generalized classes, interfaces or methods were created using references of type Object because Object is the super class of all classes in Java, but this way of programming did not ensure type safety.

Advantage of Java Generics:

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety:

- We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required:

- There is no need to typecast the object.

3) Compile-Time Checking:

- It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Syntax for creating an object of a generic type:

```
Class_name <data type> reference_name = new Class_name<data type> ();
OR
Class_name <data type> reference_name = new Class_name<>();
```

This is also known as **Diamond Notation** of creating an object of Generic type.

Example of Generic class:

```
class Gen <T> //<> brackets indicates that the class is of generic type
{
T ob;     //an object of type T is declared
Gen(T o)  //constructor
{
  ob = o;
}
public T getOb()
{
  return ob;
}
}

class Test
{
public static void main (String[] args)
{
  Gen < Integer> iob = new Gen<>(100);    //instance of Integer type Gen Class.
  int  x = iob.getOb();
  System.out.println(x);
  Gen < String> sob = new Gen<>("Hello");   //instance of String type Gen Class.
  String str = sob.getOb();
  System.out.println(str);
}
}
```

Output:

```
100
Hello
```

In the above program, we first passed an Integer type parameter to the Generic class. Then, we passed a String type parameter to the same Generic class. Hence, we reused the same class for two different data types. Thus, Generics helps in code reusability with ease.

Generic Methods

- You can also create generic methods that can be called with different types of arguments. Based on the type of arguments passed to generic method, the compiler handles each method.
- The syntax for a generic method includes a type-parameter inside angle brackets, and should appear before the method's return type.

Syntax of Generic Methods:

<type-parameter> return_type method_name (parameters) {...}

Example of Generic method:

```
class GenTest
{
 static <b>< V, T></b> void display (V v, T t)
 {
  System.out.println(v.getClass().getName()+" = " +v);
  System.out.println(t.getClass().getName()+" = " +t);
 }
 public static void main(String[] args)
 {
  display(88,"This is string");
 }
}
```

Output:

```
java lang.Integer = 88
java lang.String = This is string
```

Generic Bounded type Parameter:

- You can also set restriction on the type that will be allowed to pass to a type-parameter. This is done with the help of **extends** keyword when specifying the type parameter.

```
 < T extends Number >
```

- Here we have taken **Number** class, it can be any wrapper class name. This specifies that T can be only be replaced by **Number** class data itself or any of its subclass.

Generic Method with bounded type Parameters:

```
class Gen
{
 static < T, V extends number> void display(T t, V v)
 {
  System.out.println(v.getClass().getName()+" = " +v);
  System.out.println(t.getClass().getName()+" = " +t);
 }
 public static void main(String[] args)
 {
 // display(88,"This is string");
  display ("this is string",99);
 }
}
```

Output:

```
java.lang.String = This is string
java.lang.Double = 99.O
```

- Type V is bounded to Number type and its subclass only.
- If display(88,"This is string") is uncommented, it will give an error of type incompatibility, as String is not a subclass of Number class.

Q.3 Write a short note on Collections in java.

Collections in Java:

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion, etc. can be achieved by Java Collections.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque, etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, etc.).

What is Collection in Java:

- A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java:

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

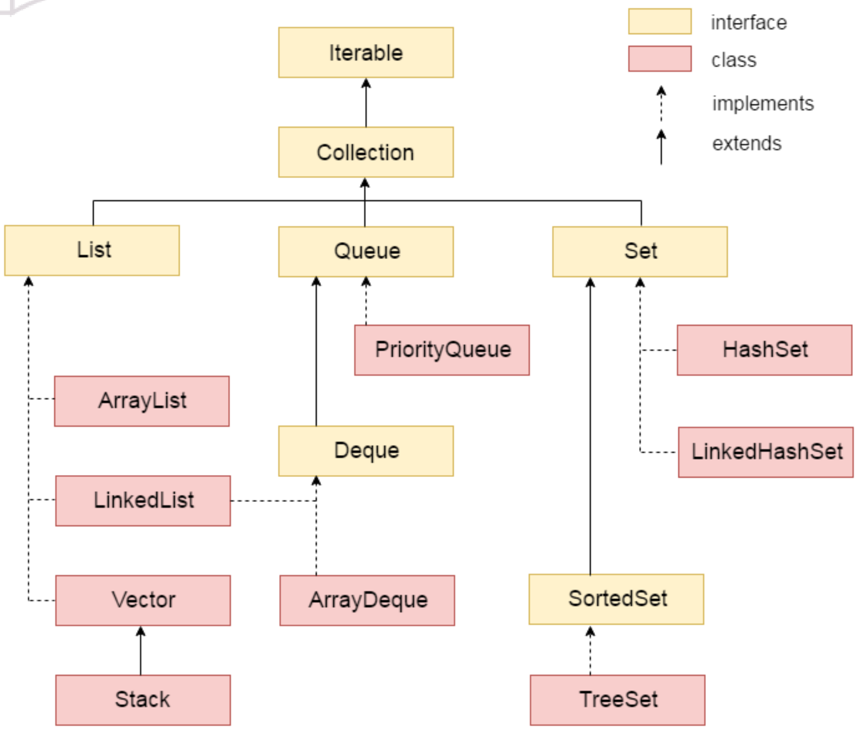
What is Collection framework:

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework:

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



Collection Framework:

- Collections framework was not a part of original Java release.
- Collections was added to J2SE 1.2. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects.
- Framework in java means hierarchy of classes and interfaces.
- Collections framework is contained in java.util package.
- It provides many important classes and interfaces to collect and organize group of alike objects.

Important Interfaces of Collection API:

Interface	Description
Collection	Enables you to work with groups of object; it is at the top of Collection hierarchy
Deque	Extends Queue to handle double ended queue.
List	Extends Collection to handle sequences list of object.
Queue	Extends Collection to handle special kind of list in which element are removed only from the head.
Set	Extends Collection to handle sets, which must contain unique element.
SortedSet	Extends Set to handle sorted set.