C28101

Advanced Java : Assignment 1

Q1. What is the advantages of collections over primitive datatypes?

→

- Collections provide a wider level of abstraction & convenience when dealing with manipulation & data storage compared to primitive datatypes.
- There are an important part of the Java collections framework, which provides a wider range of algorithms & datastructures for various programming scenarios.
- Collections provide the greater range of "generics" to ensure type safety & provide compile time type checking, which is not possible with primitive types
- We can create custom collections classes by implementing Java's collection framework in interfaces, allowing us to add collections to specific needs, while customizing primitive is very difficult
- Collections are more lucid to work on because they handle many low-level details for us, on the other hand primitive type needs manual coding for common operations.
- Collections are typically type-safe ensuring that we can, add elements of certain specific type, reducing the risk at run-time.
- Java collections come with standardized APIs making it lot easier to use & switch between different collection types, whereas primitive type lack such consistant interfaces.

Q2 → What is generic in Java? List down the advantages of it.

Generics:-

- Generic in Java can simply be defined as "Set of methods (or) Set of related methods (or) Set of similar types"

- In other words, Generic can be defined as a way to creating reusable code that works with different object types.

- Generic allows types, Integer, String, and even user-defined types to be passed as a parameter to classes, methods (or) interfaces.

- Generics in Java is similar to templates in C++

- Example:-

```
public class Box <T> {
    private T content;
  public Box () {
        content = null;
    }
  public Box (T content) {
        this.content = content;
  }
  public T getContent () {
        return content;
    }
  public void setContent (T content) {
        this. content = content;
    }
```

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

    Box<Integer> intBox = new Box<>();
    intBox.setContent(42);
    int intContent = intBox.getContent();
    System.out.println("Integer Content:"+intContent);

    Box<String> stringBox = new Box<>();
    stringBox.setContent("Saiganesh");
    String stringContent = stringBox.getContent();
    System.out.println("String Content:"+stringContent);
  }
}
```

O/P:-

```
Integer Content: 42
String Content : Saiganesh.
```

* Advantages:-

1. Bugs can be detected at compile time.
2. Type-casting is not required.
3. Code reusability.

Q3. Explain the concept of Bounded parameters.

→

The concept of bounded parameters typically means the usage of "Generics" in defining classes or methods with constraints on the type they can operate on.

- Generics allow us to create classes or methods that can work with different types while ensuring type safety.
- Bounded parameters are the way to specify these constrain -ts
- Bounded parameters help enforce type safety & allow us to write more flexible & reusable code by specifying constraints on types that can be used with generic methods & classes.
- There are 2 types of Bounded parameters we can apply in Java:
  1. Upper Bounded wildcards
  2. Lower Bounded wildcards.

1. - We can use Upper Bounded wildcards to specify type parameters must be a subtype of a certain type.

e.g:- public class Box <T extends Number> {

// .....

}

here the types under Number are ~~the~~ not only types that are allowed. i.e Integer & Double.

2. - We can use Lower Bounded wildcards to specify that a generic type parameters must be a supertype of certain type.

- This is less common but useful in specific situations

C2310)

e.g:-

```
public void addCollections(List<? super Integer> list){
                //.....
}
```

- It is achieved using the super keyword.

Q4. Difference between List, Set & Maps.

→

List:- It is a sub interface, located in 'java.util' package

e.g:-

```
import java.util.*;
public class JavaList{
  public static void main (String[] args){
    Set <String> set = new HashSet <String>;
      al = new Arraylist<>();
      al.add ("DELL");
      al.add ("ACER");
    for (String computer: al)
      System.out.println (computer);
  }
}
```

O/P:- DELL
ACER.

Set :- It uses an unordered approach, located in "java.util' package.

e.g:-

```
import java.util.*;
public class JavaSet{
  public static void main (String[] args){
    Set <String> set = new HashSet <String>();
     set.add ("ACER");
     set.add ("DELL");
   System.out.println (set); }}
```

O/P:- [DELL, ACER]

Q4. Map :-

e.g:-
```
import java.util.*;
class JavaMap{
public static void main (String [] args){
  Map<String, Integer> map = new HashMap <String, Integ
                                          >();

  map.put ("DELL", 9000);
  map.put ("ACER", 7000);
for (Map.Entry m : map.entrySet ()){
      System.out.println(m.getKey()+"   "+m.getValue());
  }
}
}
```

O/P:-

DELL 9000
ACER 7000

Differences:-

	List	Set	Map
1.	Elements can be duplicated	Element duplication is not permitted	Element duplication is not permitted.
2.	Linklist & Arraylist are classes for implementation	HashSet, Linked Hash Set & TreeSet are used for implementation.	HashMap, HashTable, ConcurrentHashMap & Linked HashMap for implementation
3.	Null value can be added	Only one null value is there in the set	Null values can be present in any number & upto one null key.

Q5 What is wildcard characters ? Explain various types of wildcard characters.

→

- The "question mark" is known as the wildcard eh in generic programming.
- It represents the unknown type, it can be used in variety of situations such as type of parameters, field (or) local variable sometimes as a return type
- Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly.
- It is often used in context of import statements & is typically used to all import all classes in a particular package

Types:- 1. Upper Bounded
2. Lower Bounded.

1. This type is mainly used to relax the restrictions We use Lower Bounded to specify that a generic type must be super type of a particular type.

declaration:-

public static void ~~addList~~ add(List<? extends Number> list)

here extends keyword is very important

Implementation:- public static void main (String [] args){

List<Integer> list1 = Arrays.asList (4,5,6,7);

System.out.println ("Total sum is:" + sum(list1));

}

// similarly we can do the same for Double.

O/P:-

Total sum is: 22.0

2. Syntax:- CollectionType <? super A>

Implementation:-

```
import java.util.*
class WildcardDemo {
public static void main (String[] args){
  List<Integer> list1 = Arrays.asList (4,5,6,7);
printOnlyIntegerClassOrSuperClass (list1);
}}
// Similarly we can create classes for other types.
public static void printOnlyIntegerClassOrSuperClassC
          List <? super Integer > list)
        {
          System.out.println (list);
        }
    }
```

O/P:-

[4,5,6,7]

Q6. Difference between ArrayList & Linked list.

→

	ArrayList	Linked list.
1.	They are internally use a dynamic array to store the elements.	1. They internally use doubly linked lists to store elements.
2.	Manipulation is slow as elements are stored in an array. If an element is removed, other elements are moved up/shifted in memory.	2. Manipulation is faster than Arraylist as elements are stored in doubly linked list, so no bit shifting is required
3.	Arraylist class can act as list only as implements lists only.	3. Linked list class can act as both list & queue as both are implemented
4.	Arraylist is better for sharing & accessing data	4. Linkedlist is better for manipulating data.
5.	The memory location for the elements of Arraylist is contagious	5. The memory location for the elements is not contagious.
6.	To be precise, an Arraylist is a resizable Array.	6. Linked list implements of doubly linked list of list interface.

Q7. Difference between HashMap, LinkedHashMap & TreeMap with an example programs.

→

- The HashMap, LinkedHashMap & TreeMap are all kinds of Map Interface in Java.
- But all these three have different characteristics in terms of performance & ordering.

HashMap:-

- It offers O(1) average time complexity for basic operations like retrieval, insertion, deletion but have collisions.
- Also, HashMap doesnot maintain any order of elements.

e.g:-

```
import java.util.*;
public static void main (String[] args){
    HashMap <Integer, String> hashm = new HashMap<>();
        hashm.put (1, "One");
        hashm.put (2, "Two");
        hashm.put (3, "Three");
    System.out.println(hashm);
}
}
```

O/P:- 1 = One , 2 = Two, 3 = Three

LinkedHashMap:-

- It maintains insertion order of elements.
- Provides slightly a slow performance than HashMap, but is very efficient.

eg:-
```
import java.util.*;
public class LinkedHashMapex {
    public static void main (String [] args) {
        LinkedHashMap <Integer, String> LHM = new LinkedHashMap <> ();
        LHM.put (1,"One");
        LHM.put (2,"Two");
        LHM.put (3, "Three");
        System.out.println (LHM);
    }
}
```

O/P:- 1 = One, 2 = Two, 3 = Three.

TreeMap:-

- It maintains elements in natural order (or) based on a custom compator.
- offers O(logn) time complexity (or) basic operations, this because it is implemented as tree

e.g:-
```
import java.util.*;
public class TreeMapex {
public static void main (String [] args) {
    TreeMap <Integer, String> treeMap = new TreeMap <> ();
        treeMap.put (3, "Three");
        treeMap.put (1, "One");
        treeMap.put (2, "Two");
    System.out.println (treeMap);
    }
}
```

O/P:- 1=One, 2=Two, 3=Three.

Q8. What is Functional Interface? How do you use Lambda Expression with Functional Interfaces?

→

- A Functional Interface is an interface that has 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 numbers of default method.
- Functional Interfaces is additionally recognised as an 'Single Abstract Method Interface'
- Functional Interfaces in Java are the new features that provides users the approach of fundamental programming.
- Functional Interface can also have any number of default, static method but can only contain one abstract method
- It can also declare methods of object class.

e.g:-

```
class Test{
public static void main (String[] args){
new Thread (new Runnable (){
        public void run (){
          System.out.println ("New Thread Created");
    }).start();
  }
}
```

O/P:- New Thread Created.

* Usage of Lambda Expression with functional Interface:

```
//definition of functional Interface.
    @FunctionalInterface
     Interface myfuncInterface{
            void myMethod();
     }
//creation of lambda expression.
myfuncInterface myLambda = ()->{
   System.out.println("Lambda Expression");};
};
```

Usage:- We can use 'myLambda' variable like an object of our functional Interface. When we call the interface method, it will execute the code defined in the lambda expression.

```
myLambda.myMethod();
```

Q.9 Which of these is a valid Lambda Expression? Explain.

```
(int x, int y) -> x+y;
or
(x,y) -> x+y;
```

→

- Both of the expressions are valid Lambda expression.
- But the point of difference is their parameter declaration.
  (int x, int y) → x+y;
- This lambda expression explicitly specifies the parameter types (int) for x & y.
-
- This is important when types are not clear from the context (or) when you want to enforce a specific type

- (x,y) → x+y
  This lambda expression uses implicit types inference, allowing the compiler to deduce the type of 'x' & 'y' based on the content.
- In this case, it would typically infer that 'x' & 'y' are types 'int' because of the addition operation

- Both expressions achieve the samething i.e addition of 'x' & 'y'.

- The choice between these depends on one's coding style & whether one wants to explicitly mention the type.

- In many cases, the second expression (implicit) is preferred.

Q10. State the advantages of using Lambda Expressions.

→

The lambda expression is an inline code that implements a functional interface without creating a concrete (or) anonymous class

Advantages:-

1. Higher efficiency:- • By using lambda expression, we can achieve higher efficiency in case of bulk operations on collections.
   • Also, it helps in achieving the internal iteration of collection rather than external iteration.

2. Fewer lines of code:- • Most useful & neat benefit of lambda expression, is that reduces lines of code.

3. Sequential & Parallel Execution:-
   • Sequential & Parallel execution support by passing behaviour as an argument in methods.
   • By using stream & APIs in Java 8, the function are passed to collection methods.
   • Now it is the responsibility of collection for processing the elements either in a sequential (or) parallel manner.

Q11. WAP using lambda expression to calculate the area of triangle.

→

```
import java.util.Scanner;

interface TriangleArea{
    double calculator (double base, double height);
}
class AreaofTriangle{
    public static void main (String[] args){
        Scanner sc= new Scanner (System.in);
        System.out.println ("Enter base: ");
        double base = sc.nextDouble();
        System.out.println ("Enter height");
        double height = sc.nextDouble();
        TriangleArea calculate (b,h) -> 0.5*b*h;
        double area = calculate.calculator (base, height);
        System.out.println ("The area of triangle is: "+area);
    }
}
```

O/P:- Enter base: 12
Enter height: 12
The area of height is: 72.0

Q12. Can Lambda Expression be passed to different target types? Justify.

→

- In java, lambda expression can be passed to different target types.
- This is due to java's functional interface & type interface
- This feature of java is part of functional programming.
- Java lambda expressions can be passed to different target types because they are closely tied to functional interfaces, benefits from type interfaces, allowing for greater flexibility & code reuse.
- This feature promotes cleaner (or) more concize code & better support for functional programming in java.

① Compatibility:- This feature enhances code reusability & makes it more expressive.
- We can use the same lambda expression in different parts of a code block, as long as parameter type & return types are compatible.

② Functional Interface:- ~~Java~~ Lambda expressions are primarly used with functional interfaces, which are interfaces with a single abstract method.
We can pass a lambda expression wherever a functional interface is expected.

```
Interface myfunction {
        int apply (int x, int y);
}

public void dooperation (myfunction operation) {
        int result = operation.apply (5, 3);
        System.out.println (result);
}
```

```
myClass myClass = new myClass();
myClass.doOperation (x,y) -> x*y;
        // Another lambda expression
}
```

Q13. What is lambda expression? Explain syntax & use of lambda expression with a suitable program.

→

- A lambda expression is a short block of code which takes in parameter & returns a value.

- Lambda expression are similar to methods but they do need a name & they can be implemented right in a body of method

- The lambda expression provides a clear & concize way to represent ~~data~~ one method interface using an expression. It is very useful in collections library.

- It saves a lot of code, we don't have define the method again for providing implementation.

- Syntax:- (argument-list) → {body}

argument-list: can be empty (or) non-empty.
arrow-token: used to link argument list to the body.
body: contains expressions & statements for lambda expression.

Program:-

```
import java.util.ArrayList;
public class Main {
    public static void main (String[] args) {
        ArrayList<Integer> numbers = new ArrayList
                                     <Integer> ();
        numbers.add (3);
        numbers.add (7);
        numbers.add (9);
```

```
numbers.forEach (n) -> { System.out.println(n); }
        );
    }
}
```

O/P:-

```
3
7
9
```