Page No.
Date 23 01 24

OOPs Interview Questions

Q) What is OOPs & why it is important?

-> OOPs is programming technique that employs Objects rather than just functions & procedures.

-> All objects are grouped into classes in Object Oriented Programming.

-> OOPs integrates real-world concepts into programming such as inheritance, Polymorphism, & abstraction.

Why OOPs is Important

-> OOPs allows more clarity in programming there by allowing simplicity in solving complex problems.

-> OOPs reduces Redundancy

-> OOPs provides ability to bind both data & code together.

-> Allows in keeping Sensitive data confidential.

-> OOPs improves code-readability

-> Polymorphism gives flexibility to the programs by allowing the entities to have multiple forms.

Q] What Are classess & Objects

* Class

1] A class is an Object's blueprint or template it is a data type that a User specifies.

2] Inside class we can define Variables, Constants, member functions, etc.

3] A class does not Consume memory at the Run-Time.

- Objects are basically instances of class. They can access variables or methods declared inside the class.

Q] What is Pure OOP? why Java is Not Pure Object Oriented Programming?

Encapsulation, Abstraction, Polymorphism, inheritance, class & Object

Pure Object Oriented Programming Language support or have features that treat everything within a program as an Object.

int y;

∴ Variable Declaration with Primitive data Types.

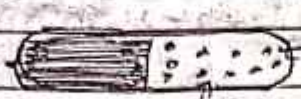
Variables can be stored in Java using primitive data types. In addition, the static keyword in Java allows us to use classes without the use of objects.

Q] What Are the Differences Between Inheritance And Polymorphism?

-> Inheritance	Polymorphism
-> with inheritance, a derived class inherits the already existing class's features (base class)	Polymorphism allows class methods to exist in multiple forms.
-> Inheritance refers to using the structure & behavior of a parent class in a subclass	Polymorphism intends on changing the behavior of parent class's method
-> Inheritance can be of single, hybrid, multiple, hierarchical, multipath & multilevel types	Polymorphism has two types: compile time & run time polymorphism.
-> Inheritance supports code reusability & reduces lines of code	Polymorphism decides which form of the function to be invoked.

Q) What is Encapsulation?

Encapsulation is the process of binding data members & methods of a program together to do a specific job, without revealing unnecessary details.



class Binding variables & methods.

Q) what is Abstraction?

→ Abstraction is the method of hiding unnecessary details from the necessary ones. it is one of the main features of OOPs.

Q) Can we Instantiate an Abstract class?

→ we cannot instantiate an abstact class because it is abstract, & it is not complete.

when we add { } parenthesis while creating an object, the compiler considers it as an anonymous class, where { } denotes the body of an anonymous class.

Q) Can a class Inherit the Constructor of it's base class ?

→ ② public Parent()

parent class	Inheritance	child class

Whenever a child class extends parent class, the subclass inherits state & behavior in the form of variables & methods from its superclass but it does not inherit Constructor of supper class.

② If Constructors are inherited then child class would Contain a parent class Constructor which is against the Constructor Constraint.

Q) How is an Abstract class Different from an ~~iter~~ interface ?

→

Both Contains only the method declaration & not their implementation.

→ when an interface is implemented, the subclass must define all it's methods & provide it's implementation.

→ when an abstract class is inherited. the subclass does not need to provide the definition of its abstract method, Until & Unless the subclass is using it.

Q) what is Constructor chaining ?

-> Constructor chaining is a sequence of invoking constructors of the same class. Upon initializing an Object.

```
new demo(8, 10); // invokes parameters
                 // constructor 3.

demo(int x, int y)
{
    this(5);
}
```

Q) what is Singleton class ?

-> Singleton class is a class that can have can have only one object at a time. After the first instantiation, the new object also points to the first instance

-> Singleton class is capable of implementing interfaces, inheriting from other classes & allowing inheritance

-> Static class on the other hand, cannot inherit its instance members which makes singleton classes more adaptable

Q] What is the fundamental Difference Between Abstraction & Encapsulation?

~~Abstraction~~ ~~Encapsulation~~

Hiding Details of Television Component. Circuitry: Encapsulation

Hiding Everything Except TV screen & Control panel : Abstraction

-> Abstraction is about express external simplicity & Encapsulation is about hiding internal complexity.

Q] What is Constructor and Destructor? Can we Overload them?

-> Constructor: It is a block of code similar to the method. A constructor will be invoked when an instance of the class is created. At the time of calling the Constructor, memory for an object is allocated in the memory space.

-> Destructor: It is the last function that is called before an object is destroyed.

Lamda

① public class Welcom {
    public void greet (Greeter greeter)
    {
        greeter.perform();
    }
}

② public interface Greeter {
    void perform();
}

③ public class GuestUser Greeter
        implements Greeter {
    @Override
    public void perform() {
    System.out.println("welcome to application");
    }
}

④ public class logged implements Greeter{
    @Override
    public void perform() {
        System.out.println("welcome back");
    }
}

(5)

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

Welcom wel = new Welcom();
Scanner sc = new Scanner (System.in);
String user = sc.nextLine();

Greeter greet;
if ("guess".equals(user)){
    greet = () -> System.out.println("
        welcome to application");
} else if ("loggedIn".equals(user)){
    greet = () -> System.out.println("
        welcome back");
} else {
    greet = () -> System.out.println("
        Unknown");
}

wel.greet(greet);

}
```

16_01_2024

Page No. 10
Date

* Lambda Predicate.

```
public static void main (String [] args)
{
    Predicate <String> letter =
            text -> text.length() > 5;
Predicate <String> start = text ->
            text.startsWith ("welcome");

boolean isMoreThan5 = letter.test ("welcome");
    // isMoreThan5 => true.
boolean isStart = start.test ("welcome");
    // isStart => true.

boolean isless = letter.negate().test ("
            welcome");
    // isless => false

boolean isComand = letter.and (
            start).test ("welcome");
    // isComand => true.

boolean isComOR = letter.or (start)
            .test ("welcom");
    // isComOr => true.
}
```

```
public class lamda {
    public stutic Void main (String [] args)
    {
        examineNumber (15, num -> num > 7);
    List<Integer> num = List.of (1, 2, 3, 4,
                5, 6, 7);
    List<Integer> odd = num.stream()
            .filter (number -> number
                % 2 == 0)
            .collect (Collectors.toList());
    System.out.println ("odd" + odd);
}

Private stutic Void examineNumber (
    int num, Predicate<Integer> era) {
        if (era.test (num)) {
    System.out.println (" The result of
        erum " + num + " is true.");
}}}

Supplier<Integer> randomNum = ()
        → {
    Random ran = new Random();
    return ran.nextInt (1000) + 1;
};

System.out.println (" random.get());
```

Page No.
Date

Lambda Unary

```
public static void main (String[] args) {
List<Integer> num = Arrays.asList(
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
System.out.println (operateOnList (num, numb
        -> numb * 3));
List<String> lett = Arrays.asList(
    "a", "b", "c", "d", "e");
System.out.println (operateOnList(
        lett, letter -> letter + letter
}
private static <T> List<T>
    operateOnList (List<T> list,
        UnaryOperator<T> ope) {
List<T> result = new ArrayList<>(
for (T listElem : list) {
    result.add(ope.apply (listElem));
}
    return result;
}
```

Expression

① List<Integer> input = List.of(1, 2, 3, 4, 5);

NumberOperation numOpe = new NumberOperation();

Operation ope = num -> num * num;
List<Integer> out = numOpe.execute(input, ope);

System.out.println(out);

② public class NumberOperation {

public List<Integer> execute(List<Integer> input, Operation ope) {

List<Integer> output = new ArrayList<>();

for (int number : input) {
output.add(ope.operate(number));
}
return output;
}
}

③ Public interface Operation {
int operate(int number);
}

⇒ Stream.

```
public class StreamCreationApp {
    public static void main (String [] args)
    {
        List <String> music = List.of ("Rock",
            "Trance", "POP", "Blue", "classic music");
        music.Stream()
            .sorted()
            .forEach (style -> System.out.println(
                style + "Extra text"));

        System.out.println (Stream.of ("Rock",
            "POP", "Blue", "classical music")
            .Collect (Collectors.toList()));

        System.out.println ("The number of
            elements in the array :" +
            Arrays.stream (new double [] {
            3.4, 5.7, 1.3, 54.3 }).count());

        IntStream.range (0, music.size())
            .forEach (num -> System.out.print
            ((num + 1) + ". " + music.get (num)
    } }
```

⇒

```
List <Integer> temp = List.of (16, 16, 16,
18, 18, 17, 19, 19, 16);

System.out.println (temp.Stream()
    .filter (temp2 -> temp2 > 16)
    .filter (temp3 -> temp3 < 19)
    .count ());          // => 3
```

→ Method Reference -> Comparator, Collection.

① public class PersonComparisonProvider {
public static int compareByNameandAge
(Person per1, Person per2) {

~~return comparing~~

return Comparator.comparing (
Person :: getName)
.thenComparing (Person :: getAge)
.compare (per1, per2);
}
}

② Public class Person implements Comparable {
private String name;
private int age;

// build Constructor & setter getter method
// write toString;

@Override
public int compareTo (Object o) {
Person per = (Person) o;
return this.getAge().compareTo (Person.
getAge());
}
}

Page No.
Date

③
```
public class MethodReference {
    public static void main(String[] args) {
        List<Person> per = List.of(
            new Person("steve", 40),
            new Person("Alex", 18));

        per.stream()
           .sorted(Person.ComparisonProvider::
                   compareByNameAndAge)
           .forEach(System.out::println);

        List<Integer> num = List.of(12, 23, 45, 45,
                67, 12, 34, 87, 102);
        Set<Integer> numSet = CollectionTransformer
                .transform(num, HashSet::new);
        System.out.println(numSet);

        Set<Person> perSet = CollectionTransformer
                .transform(people, TreeSet::new);
        System.out.println(perSet);
    }
```

④
```
Public class CollectionTransformer {

public static <T, DEST extends Collection<T>
        , SOURCE extends Collection<T>>
DEST transform(SOURCE sourceCollection,
  Supplier<DEST> Creator) {

DEST dest = Creator.get();
dest.addAll(sourceCollection);
return dest;
}
}
```

Distinct / filter

```
List <String> available = data.stream()
        .distinct().sorted()
        .collect(Collection.toList());

data.stream()
    .filter(k -> k.getPrice() < 8 && k.getName().equals("ABC"))
    .forEach(System.out::println);
```

forEach

```
List <Product> cer = List.of(
new Product("Cap", "ABC", 9.99, 4.7),
new Product("Fra", "old", 6.99, 4.2),
new Product("Gold", "ABC", 9.99, 4.8));

List <Product> che = new ArrayList<>();
    cer.stream()
    .filter(cen -> cen.getPrice() < 8)
.forEach(che::add);
System.out.println(che);
```

Page No.
Date

forEach for Map

① Map<Integer, Product> smart = new HashMap<>();

smart.put(1, new product("Fitness", "UXG", 199.9, 4.7));
smart.put(2, new Product("OK", "BYE", 54.99, 4.6));

smart.entrySet().stream()
.filter(prod -> prod.getValue() != Null && prod.getValue().getBrand().equals("BYE"))
.forEach(prodE -> {
Product pValue = prodE.getValue();
System.out.println(
pValue.getName() + " - " +
pValue.getBrand() + " - $" +
pValue.getPrice());
});

② List<String> names = List.of("aladdin", "dara", "ceers", "polly");

List<String> capNames = names.stream()
.map(na -> na.substring(0,1).toUpperCase() + na.substring(1))
.collect(Collection.toList());

① List<String> cex = data.stream()
.map(Product::getName)
.collect(Collection.toList());

② List<String> incre = data.stream()
.map(prod ->
new Product(
prod.getName(),
prod.getBrand(),
prod.getPrice() * 1.2,
prod.getRating()
)).collect(Collection.toList());

③ public class Product {
private final String name;
private final String brand;
private final string rating;
// setter & getter & Constructer.

@Override
public boolean equals(Object o) {
if (this == o) return true;
if (o == null || getClass() != o.getClass())
return false;
Product pro = (Product) o;
return Double.compare(pro.price, price)==0
&& Double.compare(pro.rating, rating)==0
&& Objects.equals(name, pro.name)
&& Objects.equals(brand, pro.brand);
}
@Override
public int hashCode() {
return Object.hash(name, brand, price,
rating); }

Page No.
Date

Sorted App.

①
```
List<Integer> lo = List.of(34, 34, 2, 64, 11);
lo.stream()
  .sorted(Comparator.reverseOrder())
  .collect(Collectors.toList());
```

②
```
data.stream()
  .sorted(Comparator.comparing(
    Product::getPrice)
  .thenComparing(Product::getName)
  .reversed())
  .forEach(System.out::println);
```

17.01.24. Collect.
17.01.24. Flatmap

Collect Grouping

```
Public class CollectGrouping {
    public static void main (String[] arg) {
List<Item> fruitBox = List.of(
        new Item("straw", 10, 9.99),
        new Item("blue", 20, 29.99),
        new Item("pear", 10, 29.99),
        new Item("apple", 30, 19.99));

Map<Boolean, List<Item>> partti =
        fruitBox.stream().collect(Collectors.
        .partitioningBy(fr -> fr.getPrice()
        >10));
System.out.println("more than $10:");
System.out.println(partti);

Map<String, List<Item>> fruits =
        fruitBox.stream().collect(
        Collectors.groupingBy(Item::
        getName));
System.out.println("fruit boxes");
    System.out.println(fruits);

Map<String, Long> fruitCount =
        FruitBox.stream().collect(
        Collectors.groupingBy(Item::getName
        , Collectors.counting()));
System.out.println("fruit by type");
System.out.println(fruitCount);
```

Page No.
Date

```
Map<String, Long> ordered =
    new LinkedHashMap<>();
ordered.entrySet
```

```
fruitCount.entrySet().stream()
    .sorted(Map.Entry.comparingByKey())
    .forEach(Box -> Box.
        ordered.put(Box.getKey(),
        Box.getValue()));
System.out.println("ordered by type:");
System.out.println(Ordered);
}}
```

(2)

```
public class Item {
    private String name;
    private int quantity;
    private double price;

// setter & getter & Constructor
// toString()
@Override
public boolean equals(Object o)
{
    if(this == o) return true;
    if(o == null || getClass() != o.getClass())
        return false;
    Item item = (Item) o;
    return quantity == item.quantity &&
        Double.compare(item.price, price) == 0
        && Objects.equals(name, item.name);
}
```

```
@Override
public int hashCode() {
    return Objects.hash(name, quantity, price);
}
}
```

4 Find by Optional

```
List<Item> fruit = List.of(
    new Item("sh", 10, 7.99),
    new Item("ABS", 20, 26.99),
    new Item("PVC", 30, 15.99));

Optional<Item> first =
    fruit.stream().findFirst();
System.out.println(first.get());

Optional<Item> any = fruit.stream()
    .findAny();

System.out.println(any.get());
```