6

Q-21

IT-23032

With Java 8, interfaces can have default methods, which provide a method body, allowing behaviour to be defined within the interface itself. This makes interfaces more powerful compared to abstract class.

Default methods in Interfaces: Interfaces can now have concrete method ~~app~~ implementations, reducing the need for abstract classes when only adding behaviour is required.

Abstract class: Abstract classes can have both abstract and concrete methods, as well as instance variables and constructors. They are used when more complex implementations or share state is needed.

Multiple inheritance: Interfaces with default methods allow multiple inheritance of behaviour, while abstract classes only support single inheritance. A class can implement multiple interfaces with default methods but can only extend one abstract class.

Yes, an abstract class can implement an interface if both the abstract class and interface provide conflicting method implementations, the subclass or concrete class must resolve the conflict by overriding the method.

Method resolution: The subclass must explicitly override the conflicting method to ~~override~~ avoid ambiguity. If not overridden, the default method from the interface is used.

Code:

IT-23082

```
interface MyInterface {
    default void method() {
        system.out.println("Interface method");
    }
}

abstract class MyAbstractClass implements MyInterface {
    public void method() {
        System.out.println("Abstract class method");
    }
}

class myclass extends MyAbstractClass {
@override
    public void method() {
            System.out.println("Concrete class method");
    }
}
```

In this case, Myclass provides its own method implementation resolving conflict.

Q-22

Hashmap uses a hash table for storage and doesn't guarantee any specific order of elements. It provides fast operations (O(1) on average) for insertion, lookup and deletion, making it ideal for scenarios where performance is crucial and order doesn't matter.

Treemap uses a Red-Black tree which stores elements in sorted order based on the natural ordering of the keys or a custom comparator. It provides O(log n) time complexity for common operations and is useful when maintaining sorted

keys is required. IT-23032

LinkedHashMap is similar to Hashmap but also maintains insertion or access order using a doubly linked list. It offers fast performance (O(1) for common operations) while ensuring elements are iterated in the order they were inserted or accessed, making it ideal for cases where the order matters.

The key difference between Treemap and Hashmap is the ordering of elements. Treemap stores elements in a sorted order based on the natural ordering of the keys or a custom comparator, ensuring the keys are always in a defined sequence. In contrast, Hashmap doesn't guarantee any specific order of elements, it stores them based on the hash codes of the keys, so the order can vary, especially across different executions of the program.

Q-23

Static binding: is resolved at compile time and is used for methods that are private, final or static. The method to be invoked is determined by the reference type. Dynamic binding, on the other hand, occurs at runtime and is used for overridden methods. In polymorphism, dynamic binding allows

the JVM to determine which method to call based on the actual object type, not the reference type. This is a core concept of polymorphism, as it enables method overriding, where the method executed is based on the actual object not the type of the reference variable.

in the following example, static binding occurs for the makeSound() method, while dynamic binding occurs for the move() method.

Code:

```
class Animal {
    public static void makeSound() {
        System.out.println("Animal sound");
    }
    public void move() {
        System.out.println("Animal moves");
    }
}
class Dog extends Animal {
    public static void makeSound() {
        System.out.println("Dog Barks");
    }
    @Override
    public void move() {
        System.out.println("Dog runs");
    }
}
```

IT-23032

```
public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.makesound();
        animal.move();
    }
}
```

Static Binding (animal.makesound()): The method is resolved at compile time based on the reference type.

Dynamic Binding: (animal.move): The method is resolved at runtime based on the actual object type

performance and method Resolution

Static: More efficient, as the method resolved at compile time. It happens for methods that are static, final or private.

Dynamic: Slightly slower because the JVM must determine the actual object type at runtime. It is essential for polymorphism to work, allowing the correct method to be called based on the object's actual type.

Q-26 IT-23032

We need to consider these facts when we are handling exceptions,

1) Identify the exception type (e.g., NullPointerException)
2) Ensure the program doesn't crash and provides meaningful feedback.
3) Use finally or try with resource to release resources like files or connections.
4) Log exceptions for debugging and monitoring.
5) Display clear error message to users.

Code Calculate Area of a Circle:

```
import java.util.Scanner;
Class Circle {
    private doble radius;
public void setRadius (double radius){
if (radius < 0) {
    throw new IllegalArgumentException ("Radius can't be negative");
    }
} this.radius = radius;
public double getArea () { return Math.PI * radius * radius; } }
public class CircleMain {
    public static void main (String[] args) {
```

IT-23032

```
Circle cc = new Circle();
try {
    Scanner sc = new Scanner(System.in);
    double r = sc.nextDouble();
    cc.setRadius(r);
    System.out.println("Area = " + cc.getAreal());
    }
    Catch (IllegalArgumentException e) {
        System.out.println(e);
    }
}
```

Q-26

In Java, there are two ways to create a thread

1) By extending the thread class
2) By implementing the Runnable interface

Code for extending thread class

```
class myThread extends Thread {
    @Override
    public void run() {
        for (int i=1 ; i<=5 ; i++)
        System.out.println(i);
    }
}
```

IT-23032

```
public class main{
    public static void main(String[] args) }
        MyThread thn = new MyThread();
        thn.start();
    }
}
```

Code for implementing the Runnable interface

```
Class MyThread2 implements Runnable{
    @Override
        public void run() {
        for(int i=1; i<=6;i++)
            System.out.println(i)
        }}
public class ThreadMain2 {
    public static void main(String [] args){
        MyThread2 obj = new MyThread2();
        Thread thn = new Thread(obj);
        thn.start();
        }
}
```

8-27

IT-23032

```
interface Edible {
    String howToEat();
}
abstract class Animal {
    abstract String sound();
}
class Tiger extends Animal {
    @Override
    public String sound() {
        return "Tiger roars";
    }
}
class Chicken extends Animal implements Edible {
    @Override
    public String sound() {
        return "Chicken clucks";
    }
    @Override
    public String howToEat() {
        return "Fry it or make chicken curry";
    }
}
abstract class Fruit implements Edible {}
class Orange extends Fruit {
    @Override
    public String howToEat() {
        return "Peel and eat";
    }
}
```

IT-23082

```
class Apple extends Fruit {
    @Override
    public String howToEat() {
        return "Make apple juice or eat raw"; }
}

public class Main {
    public static void main(String[] args) {
        Animal tiger = new Tiger();
        Chicken chicken = new Chicken();
        System.out.println(tiger.sound());
        System.out.println(chicken.sound());
        System.out.println(chicken.howToEat());
        Orange orange = new Orange();
        Apple apple = new Apple();
        System.out.println(orange.howToEat());
        System.out.println(apple.howToEat());
    }
}
```

Q-28 (IT-23032)

Java's Garbage Collection (GC) is a feature that automatically manages memory by cleaning up objects that are no longer in use. This helps developers avoid memory leaks and makes programming easier since we don't have to manually free memory.

Java provides different types of garbage collectors each with it's pros and cons are given below:

1) Serial GC: This garbage collector uses a single thread to perform garbage collection, making it suitable for small applications or single core systems. However it pauses the entire program during cleanup, which can cause delays in execution.

2) Parallel GC: Designed for multicore systems, this GC uses multiple threads to speed up garbage collection making it ideal for applications requiring high throughput. Despite it's efficiency, it still pauses the application, though for shorter durations compared to SGC.

3) CMS (Concurrent Mark-Sweep) GC: This collector runs alongside the application to minimize pause times, making it beneficial for low-latency applications like real time systems. It consumes more CPU resource

(IT-23032)

4) G1 (Garbage-First) GC : it divides the heap into regions and prioritizes collecting the ones with most garbage first, allowing for predictable pause times. It is well suited for large applications, but it comes with increased complexity.

How GC decides what to clean :

Garbage collection works by identifying objects that are no longer reachable, meaning no active references point to them. Common algorithms include mark-sweep, where unused objects are marked and removed, and mark-compact, which not only removes garbage but also defragments memory to optimize allocation.

-problems in Large Application :

Garbage collection can introduce challenges in large scale applications, including pauses that temporarily freeze execution, making it unsuitable for real time systems. Memory fragmentation can make it difficult to allocate large objects and a high CPU usage from collectors like CMS can degrade overall performance. Additionally, tuning GC settings for optimal efficiency can be complex and application dependent.

Finalize Method

IT-23032

The finalize() method is invoked by the GC before an object is removed, allowing resource cleanup such as closing files or releasing connections. However, it's execution is not guaranteed to happen immediately, which can introduce performance issues. Due to its unpredictability, finalize() is generally discouraged in favor of using try-with-resources or explicit cleanup methods.

Q-20

```
import java.io.*;
import java.util.*;
Public class FP{
    public static void main(String[] args){
    String inputF = "input.txt";
    String outputF = "output.txt";

    try{ File inputFile = new File(inputF);
         Scanner sc = new Scanner(inputFile);
         int maxi = -1;
```

IT-23032

```
    int num = 0;
while (sc.hasNextInt()){
    int number = sc.nextInt();
     sum += number;
   if (number > maxi) maxi = number;
}
 sc.close();
printWriter writer = new PrintWriter (outputF);
writer.println(" Highest value = " + maxi);
writer.println(" sum = " + sum);
 writer.close();
}
 catch (FileNotFoundException e) {
     System.out.println ("An error occured");
        }
   }
}
```

Q-30

IT-23032

```
import java.util.*;
public class Q-30-Array {
    public static void main (String [] args){
        Scanner sc = new Scanner (System.in);
System.out.println("Enter the size of the first array \n");
        int n = sc.nextInt();
    while (n<=20){
        System.out.println("please Enter a value greater than 20 ");
            n = scanner.nextInt();
        }
    int[] array1 = new Int [n];
    int[] array2 = new int[(int)Math.ceil(n/10.0)];
System.out.println("Enter the elements of the first array:");
    for (int i=0; i<n; i++){
        array1 [i] = sc.nextInt(); }
    System.out.println("Enter the elements of 2nd array");
    for(int i=0; i<n; i++ ) { array2[i]= sc.nextInt(); }
    int [] divisors = new int [array1.length];
    int[] remainders = new int [array1.length];
    for (int i=0; i<n; i++){
    int divIdx = i % array2.length;
    divisor[i] =(int) Math.ceil((double) array1[i]/array2[divIdx]);
```

```
    remainders[i] = array1[i] % array2[divIdx];
  }
  System.out.println("Divisors: " + Arrays.toString(divisors));
  System.out.println("Remainders " + Arrays.toString(remainders));
  sc.close();
 }
}
```

IT-23032

Q-31

```
import java.time.*;
public class CurrentDateTime {
  public static void main(String[] args) {
    LocalDateTime CDT = LocalDateTime.now();
    DateTimeFormatter ff = DateTimeFormatter.ofPattern("yyyy-mm-dd HH:mm:ss");
    String fDT = CDT.format(ff);
    System.out.println("Current Date and Time " + fDT);
  }
}
```

Q-32

IT-23032

```
public class Counterclass {
    private static int instanceCount=0;
    public Counterclass(){
        instanceCount++;
        if (instanceCount>50) instanceCount=0;
    }
    public static int getInstanceCount(){
        return instanceCount;
    }
    public static void main(String[] args){
        for(int i=0; i<100; i++){
            new counterclass();
            System.out.println("Instance count:"+Counterclass.
                                getInstanceCount());
        }
    }
}
```

Q-33

```
Import java.util.*;
public class main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        String s= sc.next();
    List<Integer> v = new ArrayList<X>;
```

```
while (sc.hasNextInt()) {
    v.add(sc.nextInt());
}
Collections.sort(v);
if (s.equals("smallest")) {
    system.out.println(v.get(0)); }
else {system.out.println(v.get(v.size()-1)); }
sc.close();
}
}
```

IT-23032

Q-34

Output:

true // s1.equals(s2): Content is the same.

false // s1 == s2: different objects

true // s1 == s3 : same object in memory

Q-36, 37

IT-23032

```
interface Alpha {
    void methodA();
    void methodB();
}

interface Beta {
    void methodC();
    void methodD();
}

abstract class AbstractBase implements Alpha {
    public abstract void methodE(); }

class FinalClass extends AbstractBase implements Beta {
    @Override
    public void methodA() {
        System.out.println("Implementing A"); }

    @Override
    public void methodB() {
        System.out.println("Implementing B"); }

    @Override
    public void methodC() {
        System.out.println("Implementing C"); }

    @Override
    public void methodD() {
        System.out.println("Implementing D"); }

    @Override
    public void methodE() {
        System.out.println("Implementing E"); }
```

```
public static void main (String[] args) {
    FinalClass fc = new FinalClass();
    fc.methodA(); fc.methodB(); fc.methodC();
    fc.methodD(); fc.methodE();
    }
}
```

IT-23032

Q-38

```
public class SingletonExample {
    private static SingletonExample instance;
    private SingletonExample() { }
    public static synchronized SingletonExample getInstance() {
        if (instance == null) { instance = new SingletonExample(); }
        return instance;
    }
    public void showMessage() {
        System.out.println("I am from SingletonExample!"); }
    public static void main(String[] args) {
        SingletonExample ss = SingletonExample.getInstance();
        ss.showmessage();
        SingletonExample anss = SingletonExample.getInstance();
        System.out.println("Both are same: " + (ss == anss));
    }
}
```

Q-39&40 II-23032

```
public class ArithmeticOperations {
    public static void main(String[] args) {
    try { if (args.length<2) {
        throw new IllegalArgumentException("Two integer
        arguments are required");
        }
    int num1 = Integer.parseInt(args[0]);
    int num2 = Integer.parseInt(args[1]);
    int sum = num1 + num2;
    int diff = num1 - num2;
    Int product = num1 * num2;
    int quotient = num1/num2;
System.out.println("sum: "+sum);
System.out.println("Difference:" + difference);
system.out.println("product "+product);
System.out.println("Quotien" + quotient);
}
 catch (NumberFormatException e) {
    System.out.println("Invalid input"); }
catch (ArithmeticException e) {
    System.out.println("Arithemetic error"); }
catch (IllegalArgumentException e) {
    system.out.println("Error");
}
}
```