IT23013

17| Differences Between Arraylist and Linkedlist in Java.

Arraylist	Linked List
1. O(1) (Direct indexing)	1. O(n) (sequential traversal)
2. Amortized O(1) (Occasional resize)	2. O(1) (Direct append)
3. O(n) (Shifting required)	3. O(1) (If node reference is known, otherwise O(n) for traversal)
4. Less overhead (array only)	4. More overhead (extra references)
5. Search contains O(n)	5. Search contains O(n)

When to use which?

- Use Arraylist when:
  - You need fast random access (O(1) lookup).
  - ~~Interai~~ Insertions/delations infrequent.
  - Memory efficiency is a priority.

IT23013

• Use Linked List when:

- Frequent insertions/deletions in the middle are required.
- Sequential access is preferred over random access.
- Memory overhead is not concern.

18] Random Generator

```
import java.util.Random;
import java.util.Arrays;

public class CustomRandomGenerator {
    private static final int[] SEED_ARRAY={3, 7, 11,
    17, 23};
    public static int myRand(int n){
        long timestamp = System.currentTimeMillis();
        int seed = SEED_ARRAY[(int)(timestamp % SEED_A
                length)];
        return (int)((timestamp*seed) % n);
    }
```

```
public static int[] myRand(int n, int count) {
int[] randomNumbers = new int[count];
for(int i=0; i<count; i++){
    randomNumbers[i] = myRand(n);
}
return randomNumbers;
}

public static void main(String[] args) {
System.out.println("Single random number: "+myRand(
System.out.println("Multiple random numbers: "+Arrays.
(myRand(100,5)));
}
}
```

IT23013

19] Multithreading in Java

Multithreading in Java allows multiple threads to run concurrently, improving performance and responsiveness. Java provides built-in support through the Thread class and Runnable interface.

Thread class vs Runable Interface

1. Thread class - Extending Thread requires overriding the run() method but prevents extending other classes.

2. Runable Interface - Implementing Runnable allows better flexibility, enabling the class to extend other classes.

Example:

```
Class MyThread extends Thread {
 public void run () { system.out. println ("Thread
 using Thread class"); }
} }
```

IT23013

```
class MyRunnable implements Runnable {
  public void run(){System.out.Println("Thread using
  Runnable interface");}
}
 pubblic class ThreadExample {
  public static void main (string[] args) {
   new MyThread().start();
   new Thread (new MyRunnable()).start();
  }
}
```

potential Issue in Multithreading

- Race condition - Multiple threads modifying data - simultaneously, causing inconsistencies.
- Deadlocks - Two or more theads waiting for each oth to release resources.
- Starvation - Low-Priority threads not getting CPU ti
- Livelock - Threads continuously responding to each other without making progress.

IT23013

Using Synchronised for Thread Safety

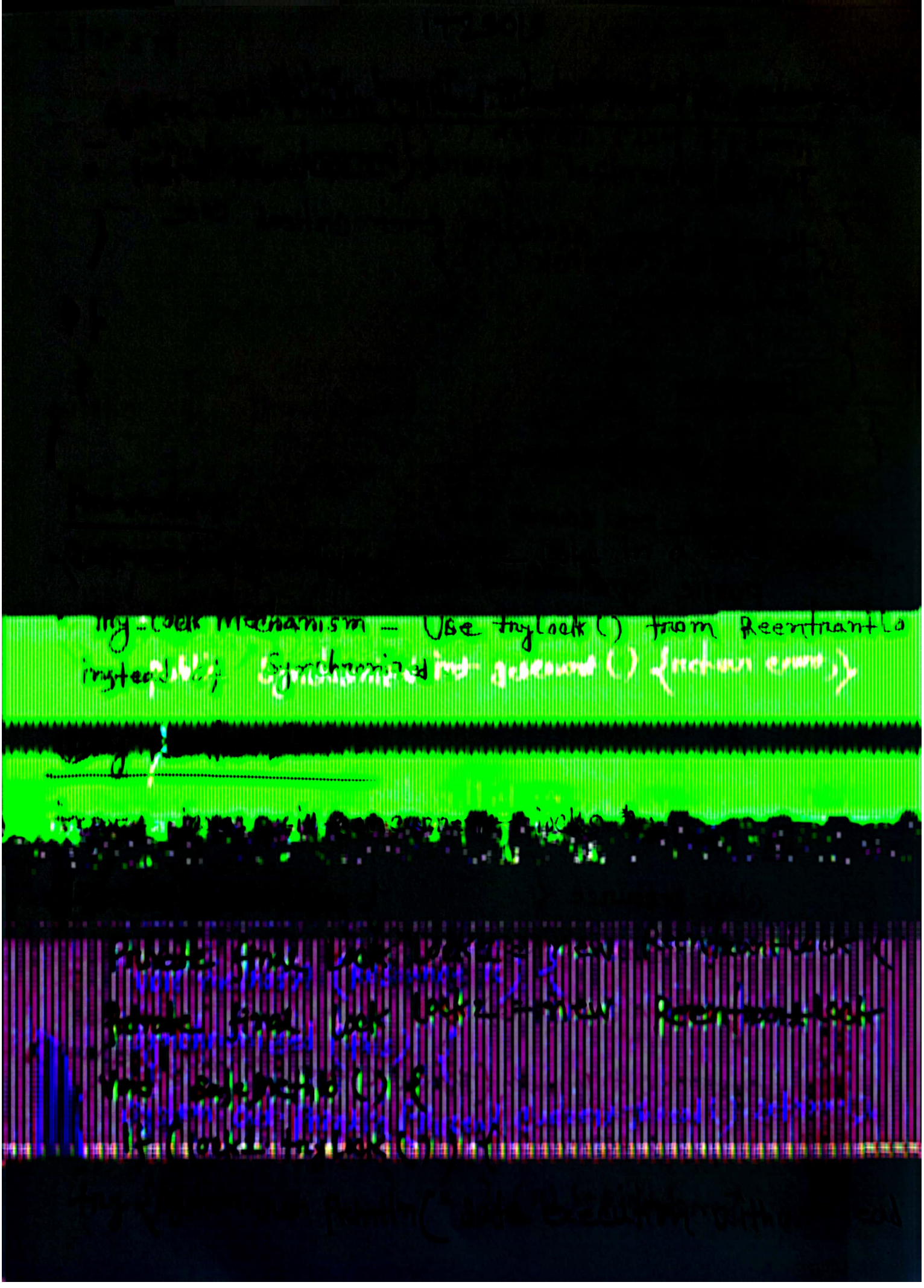
The synchronized keyword prevents multiple threads from accessing ~~erict~~ critical code simultaneously.

Example

```
class SharedResource {
    Private int count = 0;
    public Synchronized void increment() {count++;}
    public Synchronized int getcount() {return count;}
}
```

- Deadlock Scenario:

```
class Resource {
void methodA (Resource r) {
Synchronized (this) {
System.out.Println (Thread.CurrentThread().getName():
" locked Resource 1");
Synchronized (r) {
```



```
        finally { lock2.unlock(); }
    }
    finally {lock1.unlock(); }
  }
}
```

IT23013

20|

Exception Handling in Java (short overview)

Exception handling in Java ensures that runtime errors do not disrupt program flow. It is managed using try, catch, finally, throw, and throws keywords.

Checked vs Unchecked Exceptions

- Checked Exceptions: Checked at compile-time; must be handled using try-catch or declared with throws.
- Unchecked Exceptions: Occur at runtime; typically caused by programming errors.

Creating & Throwing Custom Exceptions

- Extend Exception for checked exceptions or RuntimeException for unchecked exceptions.
- Use throw to explicitly throw an exception.

LT23013

```
class CustomException extends Exception {
    public CustomException (String message)
    {
     super (message); }
}
throw new CustomException ("custom checked
    exception");
```

Role of throw and throws

- throw : used to manually throw an exception.
- throws : Declares exceptions a method might throw.

```
public void method () throws IOException {
    throw new IOException ("Error occurred");
}
```

IT23013

~~Preventing~~

Preventing Resource Leaks

- Use finally Block: Ensures resources are closed.
- Try-with-Resources (AutoCloseable): Java 7+ automatically closes resources.

```
try (FileInputStream file = new FileInputStream("file.txt"))
{
} catch (IOException e) {
    e.printStackTrace();
}
```

IT23013

(21)

Interfaces in Java can have default methods starting from Java 8, and this feature introduces some interesting distinctions when compared to abstract classes's.

Interfaces with default Methods;

- Interfaces can have both abstract methods and default methods.
- Default methods allow you to add functionally to an interface without breaking existing implementations.
- Since interfaces can have no state, they can only provide behavior or constants.

IT23013

Abstract classes:

- Abstract classes can hold both state (instance variable) and behaviour (method).
- They can have constructors, non-static fields, and concrete methods.

Conflicting Method Implementations between abstract classes and Interfaces:

If both an abstract class and interface provide conflicting method implementations, the concrete class must explicitly resolve the conflict by overriding the method.

example code :

```
interface A {
    default void hello() {
        System.out.println("Hellow from A");
    }
}

abstract class B implements A {
    @override
    public void hello() {
        System.out.println("Hellow from B");
    }
}

class C extends B {
    @override
    public void hello() {
        System.out.println("Hellow from C");
    }
}
```

IT23013

(22)

Feature	Hashmap	Tree map	LinkedHashMap
Internal Structure	Hash table	Red black tree	Hash Table + Doubly linked List
order of Elements	NO order guaranteed	sorted	Insertion order
Insertion time	O(1)	O(logn)	O(1)
Deletion time	O(1)	O(logn)	O(1)
When to use	Fast access with no order needed	when you need sorted keys or range queries	When you need insertion/access order

IT23013

How does Tree map differ from HashMap in ordering

- Hashmap does not maintain any order of its keys The order in which you iterate over the keys or entries can be arbitrary.
- Treemap, on the other hand maintains a sorted order of the key based on their natural ordering or a comparator, meaning the elements will be traversed in ascending order by default.

23

Static Binding and Dynamic Binding in Java and their Relation to polymorphism.

Static Binding : Occurs when the method call is resolved at compile-time. This happens when the method being invoked is determined based on the type of the reference. Static methods, private method, and final methods are examples of methods that are statically bound.

Dynamic Binding : Occurs when the method call is resolved at runtime, based on the actual object type, not the reference type. This is a key feature of runtime polymorphism. Methods overridden in subclasses are dynamically bound.

Static binding Example :

a. animal.sound();

b. animal.bark();

24

Advantages of ExecutorService over Manual Thread Management:

1. Thread Pool management.

2. Automatice thread Reuse.

3. Task scheduling.

4. Graceful Shutdown.

5. Error Handling and Monitoring.

IT23013

Benefits of using callable over runnable.

Runnable:

*. runnable tasks are designed to perform an actio but do not return any result. It is deal when you simply want to excute some code without needing any feedback.

*. Runnable cannot throws checked exceptions directly; they must be wrapped in a try-catch block.

Callable:

*. Callable is similar Runnable but can return a result. This is usefull when you need the outcome of the task or need to compute a value.

IT23013

*. Callable can throw checked exeeptions, which means, you can propogate exceptions from the task directly

Differences between submit() and excute() Methods.

⊛ excute method :

*. This method is used to submit a runnable task to the Executorservice. The execute() method doesn't return any result any or feedback. it only signals that the task the task has bee submitted for execution.

*. It doesn't provide a way to haddle excel thrown by the task.

IT23013

submit method:

*. The submit method is more versatile than execute.

It allows you to submit both runnable and callable tasks to the ExecutorService.

- If the task throws an exception, it can be retrieved through the future object

25

CODE:

```
Class NegativeRadiusException extends Exception{
    public NegativeRadiusException(String message){
      super(message);
    }
}
public class circle{
        private double radious;
public void setRadius(double radius)throws NegativeRadi
                                        Exception{
if(radius<0){
throw new NegativeRadiusException("Radius cannot be
                                    negative");
}
  this.radius = radius;
}
public static void main(String[]args){
   circle circle = new circle(){
```

IT23013

```
try{
    circle.setRadius(-5);
    System.out.println("Area of circle: "+ circle.calculate
                                    Area(); }
 catch(NegativeRadiusException e){
   system.out.println("Error", + e.getmessage());
}
 try {
    Circle.setRadious(10);
   System.out.println("Area of circle: " + circle.
                        calculateArea());
}
catch(NegativeRadiousException e){
   System.out.println("Error:+ e.getmesag
   }
}
}
```

26. There are two primary ways to create a thread:

1. By implementing the runnable interface.
2. By extending the thread. class.

Creating a thread using the runnable interface:

```
class MyRunnable implements Runnable {
    @Override
    public void run(){
    System.out.println("Thread is running Runnable
                        Interface!");
    }
}
public class RunnableExample {
public static void main(String[] args){
 MyRunnable myRunnable = new MyRunnabe();
 Thread thread = new Thread(myRunnable);
 thread.start();
  }
}
```

Creating a thread by extending the thread class:

```
class MyThread extends Thread {
@Override
public void run() {
    System.out.println("Thread is running by
        extending Thread class!");
}
}
public class ThreadExample {
public static void main(String[] args) {
    MyThread mythread = new MyThread();
    myThread.start();
}
}
```

28||

Java's garbage collection (GC) is reponsible for automatically managing memory by reclaiming the memory that is no longer in use. Java uses automatic memory management, which helps developers focus on logic without having to manually allocate memory. The goal is to find and remove objects that are no loger referred by the program, freeing up memory to be reused.

phases of Garbage Collection :

1. Mark Phase : The GC identifies all objects that are reachable
2. Sweep phase : The GC removes objects that are not reachable (no references).

IT23013

3. Compact Phase: The GC may move objects in memory to eliminate gaps, allowing more efficient memory usage.

29

Code:

```
import java.io.*;
import java.util.Scanner;

Public class HighestValueAndSum{
    public static void main (String[] args){
        String inputFile = "input.txt";
        String outputFile = "output.txt";
int highestValue = Integer.MIN_VALUE;
    int sum = 0;
 try(Scanner scanner = new Scanner(new F
                        inputFile))){
 while (scaner.hasNextInt()){
 int number = scanner.nextInt();
        sum += number;
 if(number > highestValue){
        highestValue = number;
 }
}
```

IT23013

```
catch(FileNotFoundException e){
System.out.println("The input file was not found.");
 return;
}

try(PrintWriter writer = new PrintWriter(new File(outputFile));
{ writer.println("Highest Value : "+ highestValue);
  writer.println("Sum: "+sum);
system.out.println("Results written to "+outputFile);
}
 catch(FileNotFoundException e){
  System.out.println("Error writing to the
                      output file.");
  }
}
}
```

30.11 IT23013

CODE:

```
import java.utill.Scanner;
import java.utill.Arrays;

Public class ArrayDivision{
public static void main(String[]args){
  Scanner sc = new Scanner(system.in);
   system.out.print("Enter the size of the firs
      array (n>20): ");
  int n= sc.next Int();
if(n<=20){
  System.out.println("n must greater than 20")
  return;
}
 int m = n/10;
 int[] array1 = new int[n];
 int array2 =new int[m];
 Systemout.println("Enter elements of the first
    array(size "+n + " ): ");
 for(int i=0;i<n; i++){
```

```
array1[i] = sc.nextInt();
}
System.out.println("Result of dividing the first array
by the second array:");
for(int i=0; i<m; i++){
double divisionResult = (double) array1[i]/array2[i];
int divisor = (int) Math.ceil(divisionResult);
int remainder = array1[i] % array2[i];
~~int divisor = (int) Math.ceil~~
System.out.println("For index "+i+": ");
System.out.print("Divisor(roundup): "+divisor);
System.out.println("Remainder: "+remainder);
}
sc.close();
}
```

IT23013

31

Current Datetime:

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

IT23013

32

```
Public class CounterClass {
    Privale static int instanceCount = 0;
    Privale static final int MAX_COUNT = 50;
    Public CounterClass() {
     instanceCount++;
     if(instanceCount > MAX_count) {
        instance Count = 0;
     }
    }
public static int getInstanceCount() {
        return instanceCount;
    }
public static void main(String[] args) {
for(int i=0; i<55; i++) {
new CounterClass();
System.out.Println("Instance Count: "+CounterClass.getInstanceCo
                                                    (1)
  }
 }
}
```

IT23013

33

```
Public class ExtremeFinder {

 Public static int findExtreme(String type, int[] numbers) {
    if(numbers.length==0){
   throw new IllegalArgumentException("At least one number
                                       must be provided");
    }

   int extreme=numbers[0];
   for(int num:numbers){
   if(type.equalsIgnoreCase("smallest")){
   if(num<extreme){
   extreme=num;
  }
  else if(type.equalsIgnoreCase("largest")){
   if(num>extreme){
   extreme=num;
   }
  else {
  throw new IllegalArgumentException("Invalid typ. Use 'smallest'
                                      or 'largest.'");
 }}
 return extreme;}
```

IT 23013

```
Public static void main(String[] args) {
int x=findExtreme("smallest",5,2,9,1);
int y=findExtreme("largest",8,3,10,4);
System.out.println("smallest: "+x);
System.out.println("Largest: "+y);
}
}
```

34)

```
/*The .equals() method checks whether the content
of s1 and s2 are the same. since both contain
"This is ICT 2107 Java", it returns true.*/
true
// The == operator checks whether s1 and s2 refer to the same
memory location. They are different objects,
false
//they point to the same memory location in String pool
true
```

IT23013

35, 36

```
interface Alpha {
    void methodA();
    void methodB();
}
interface Beta {
    void methodC();
    void methodD();
}

abstract class AbstractBase implements Alpha {
    public void methodA() {
        System.out.println("Method A implemented in AbstractBase.");
    }
    public abstract void methodE();
}

class FinalClass extends AbstractBase implements Beta {
    public void methodB() {
        System.out.println("method B implemented FinalClass.")
    }
```

IT23013

```
Public void methodC() {
 System.out.Println("method C implemented in Finalclass.");
 }

Public void methodD() {
 System.out.println("Method D implemented in FinalClass.");
 }

Publi'c void methodE(){
 System.out.println("method E implemented in FinalClass.");
 }

Public class Main {
 Public static void main(String[] args) {
FinalClass obj = new FinalClass();
obj.methodA();
obj.methodB();
obj.methodC();
obj.methodD();
obj.methodE();
 }
}
```

IT23013

37

Issue with the code

The class Z implements both interfaces X and Y, but both interfaces contain a default method show()

This results in a "duplicate default method" conflict, leading a compilation error because Java does not know which show() method to inherit

Solution 1: Override the show() method in class Z

```
Public class Z implements X,Y{
@Override
Public void show(){
X.super.show(); //Explicitly calling show() from inherit X
}
Public static void main(String[] args){
  Z obj= new Z();
  Obj.show();
}
}
```

IT23013

Solution 2

```
Public class Z implements X,Y {
@Override
Public Void show() {
System.out.Println("Z's own show method");
}
Public static void main(String[] args){
Z obj = new Z();
obj.show(); //call's overridden show() from Z
}
}
```

IT23013

38)

```
Class SingletonExample{
  Private static SingletonExample instance;
  Private SingletonExample(){
  System.out.Println("SingletonExample instance created.");
  }
 Public Static synchronized SingletonExample getInstance(){
  if(instance == null){
  instance= new SingtonExample();
  }
  Return instance;
}
Public void showMessage(){
   System.out.Println("Hello from SingletonExample!");
   }
 }
Public class SingletonDemo{
 Public Static void main(String[]args){
SingletonExample obj1 = SingletonExample.getInstance();
SingletonExample obj2 = singtonExample.getInstance();
Obj1.showMessage();
if(obj1 == obj2){ System.out.Println("Both are same instance");
else{System.out.Println("Different instance");
```

IT23013

Q1

```
class InvalidAmountException extends exception {
public InvalidAmountException(String message){
super(message);
    }
}
class InsufficientFundsException extends Exception {
public InsufficientFundsException(String message) {
 super(message);
}
}
class wallet {
 Private double balance;
 Public wallet(double initialBalance) {
    this.balance=initialBalance;
 }
Public void AddFunds(double amount) throws InvalidAmountExcep
{
    if(amount<0){
    throw new InvalidAmountException("Amount can't be negat
}
```

IT23013

```
balance += amount;
System.out.println("Added $" + amount + " to wallet. New balance:
                                        $" + balance);
    }
Public void spend(double amount) throws InvalidAmountExcept
InsufficientFundsException {
        if(amount<0) {
        throw new InvalidAmountException("Amount cant be negative
        }
        if(amount>0) {
    throw neo InsufficientFundsException("Insufficient balance");
    }
    balance-= amount;
    System.out.Println("spent $" + amount + "from wallet.Newbalance :
                                        $" + balance);
    }
    Public double getbalance(){
    return balance;
    }
```

IT23013

```
Public class WalletTest {
 Public static void main (String[] args) {
 Wallet mywallet= new wallet(100);
 try {
    mywallet.addfunds(50);
    mywallet.spend(30);
    myWallet.addFunds(-20);
    } catch (InvalidAmountException/InsufficientFundsException e:
      System.out.Println("Exception caugh: "+e.getMessage()
    }
try {
  myWallet.spend(200);
   } catch(InvalidAmountException/InsufficientFundsException e)
    System.out.Println("Exception caught:"+e.getMessage();
   }
}
}
```