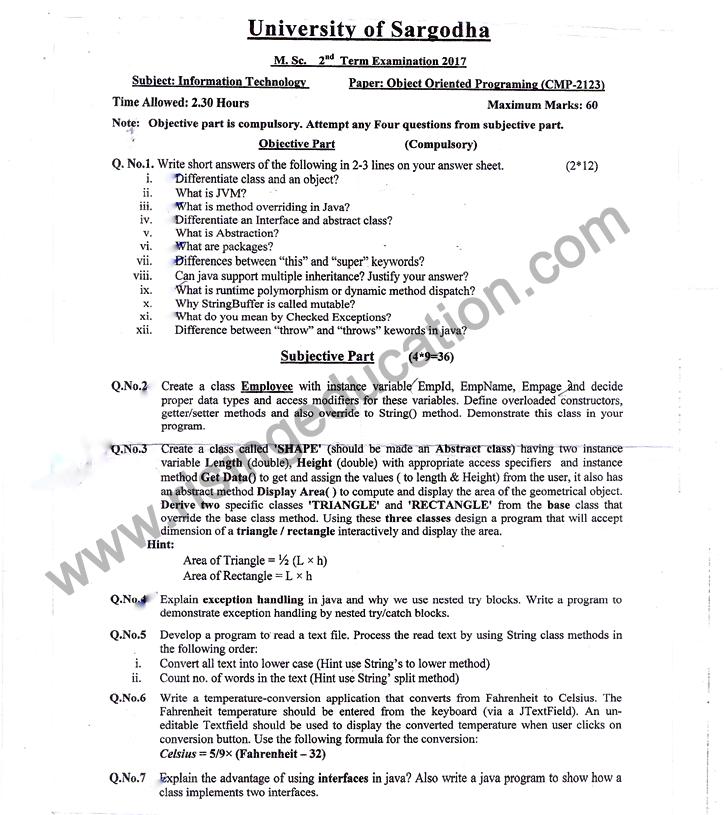
******

***Question No.1:***

A class is a non-primitive or user-defined data type in Java, while **an object is an instance of a class**. A class is a basis upon which the entire Java is built because class defines the nature of an object.

***Question No.2:***

A Java virtual machine is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation.

***Question No.3:***

In any object-oriented programming language, Overriding is a **feature that allows a subclass or child class to provide a specific implementation of a method** that is already provided by one of its super-classes or parent classes.

***Question No.4:***

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. Since Java 8, it can have default and static methods also. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |

***Question No.5:***

In Java, Data Abstraction is defined as **the process of reducing the object to its essence** so that only the necessary characteristics are exposed to the users. Abstraction defines an object in terms of its properties (attributes), behavior (methods), and interfaces (means of communicating with other objects).

***Question No.6:***

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for: Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college. **staff**

***Question No.7:***

The Java programming language supports **multiple inheritance of type**, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements.

***Question No.8:***

The Java programming language supports **multiple inheritance of type**, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements.***Question No.9:***

Runtime polymorphism or Dynamic Method Dispatch is **a process in which a call to an overridden method is resolved at runtime rather than** compile-time. ... The determination of the method to be called is based on the object being referred to by the reference variable.

***Question No.10:***

Answer: **The String class is considered as immutable**, so that once it is created a String object cannot be changed. If there is a necessity to make alot of modifications to Strings of characters then StringBuffer should be used.

***Question No.11:***

The exceptions that are checked during the compile-time are termed as Checked exceptions in Java. The Java compiler checks the checked exceptions during compilation to verify that **a method that is throwing an exception contains the code to handle the exception with** the try-catch block or not.

5: The compiler will give an error if the code does not handle the checked exceptions

1: Checked exceptions occur at compile time

10: Examples of Checked exceptions:File Not Found ExceptionNo Such Field Excep

***Question No.12:***

The throw keyword is used to throw an exception explicitly. It can throw only one exception at a time. The throws keyword can be used to declare **multiple exceptions**, separated by a comma. ... Syntax of throws keyword includes the class names of the Exceptions to be thrown

***Long Questions:***

***Q.4***

# **Java Nested try block**

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

### **Why use nested try block**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### **Syntax:**

1. ....
2. //main try block
3. **try**
4. {
5. statement 1;
6. statement 2;
7. //try catch block within another try block
8. **try**
9. {
10. statement 3;
11. statement 4;
12. //try catch block within nested try block
13. **try**
14. {
15. statement 5;
16. statement 6;
17. }
18. **catch**(Exception e2)
19. {
20. //exception message
21. }
23. }
24. **catch**(Exception e1)
25. {
26. //exception message
27. }
28. }
29. //catch block of parent (outer) try block
30. **catch**(Exception e3)
31. {
32. //exception message
33. }
34. ....

## **Java Nested try Example**

### **Example 1**

Let's see an example where we place a try block within another try block for two different exceptions.

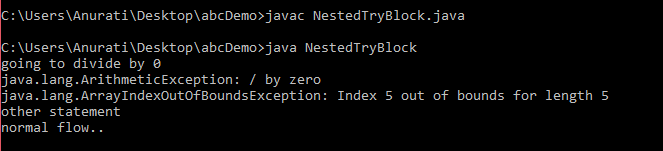
**NestedTryBlock.java**

1. **public** **class** NestedTryBlock{
2. **public** **static** **void** main(String args[]){
3. //outer try block
4. **try**{
5. //inner try block 1
6. **try**{
7. System.out.println("going to divide by 0");
8. **int** b =39/0;
9. }
10. //catch block of inner try block 1
11. **catch**(ArithmeticException e)
12. {
13. System.out.println(e);
14. }

17. //inner try block 2
18. **try**{
19. **int** a[]=**new** **int**[5];
21. //assigning the value out of array bounds
22. a[5]=4;
23. }
25. //catch block of inner try block 2
26. **catch**(ArrayIndexOutOfBoundsException e)
27. {
28. System.out.println(e);
29. }

32. System.out.println("other statement");
33. }
34. //catch block of outer try block
35. **catch**(Exception e)
36. {
37. System.out.println("handled the exception (outer catch)");
38. }
40. System.out.println("normal flow..");
41. }
42. }

**Output:**



When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

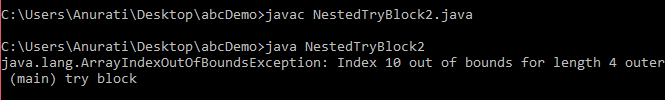
### **Example 2**

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

**NestedTryBlock.java**

1. **public** **class** NestedTryBlock2 {
3. **public** **static** **void** main(String args[])
4. {
5. // outer (main) try block
6. **try** {
8. //inner try block 1
9. **try** {
11. // inner try block 2
12. **try** {
13. **int** arr[] = { 1, 2, 3, 4 };
15. //printing the array element out of its bounds
16. System.out.println(arr[10]);
17. }
19. // to handles ArithmeticException
20. **catch** (ArithmeticException e) {
21. System.out.println("Arithmetic exception");
22. System.out.println(" inner try block 2");
23. }
24. }
26. // to handle ArithmeticException
27. **catch** (ArithmeticException e) {
28. System.out.println("Arithmetic exception");
29. System.out.println("inner try block 1");
30. }
31. }
33. // to handle ArrayIndexOutOfBoundsException
34. **catch** (ArrayIndexOutOfBoundsException e4) {
35. System.out.print(e4);
36. System.out.println(" outer (main) try block");
37. }
38. **catch** (Exception e5) {
39. System.out.print("Exception");
40. System.out.println(" handled in main try-block");
41. }
42. }
43. }

**Output:**



# Question No.7

# **Interface in Java**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

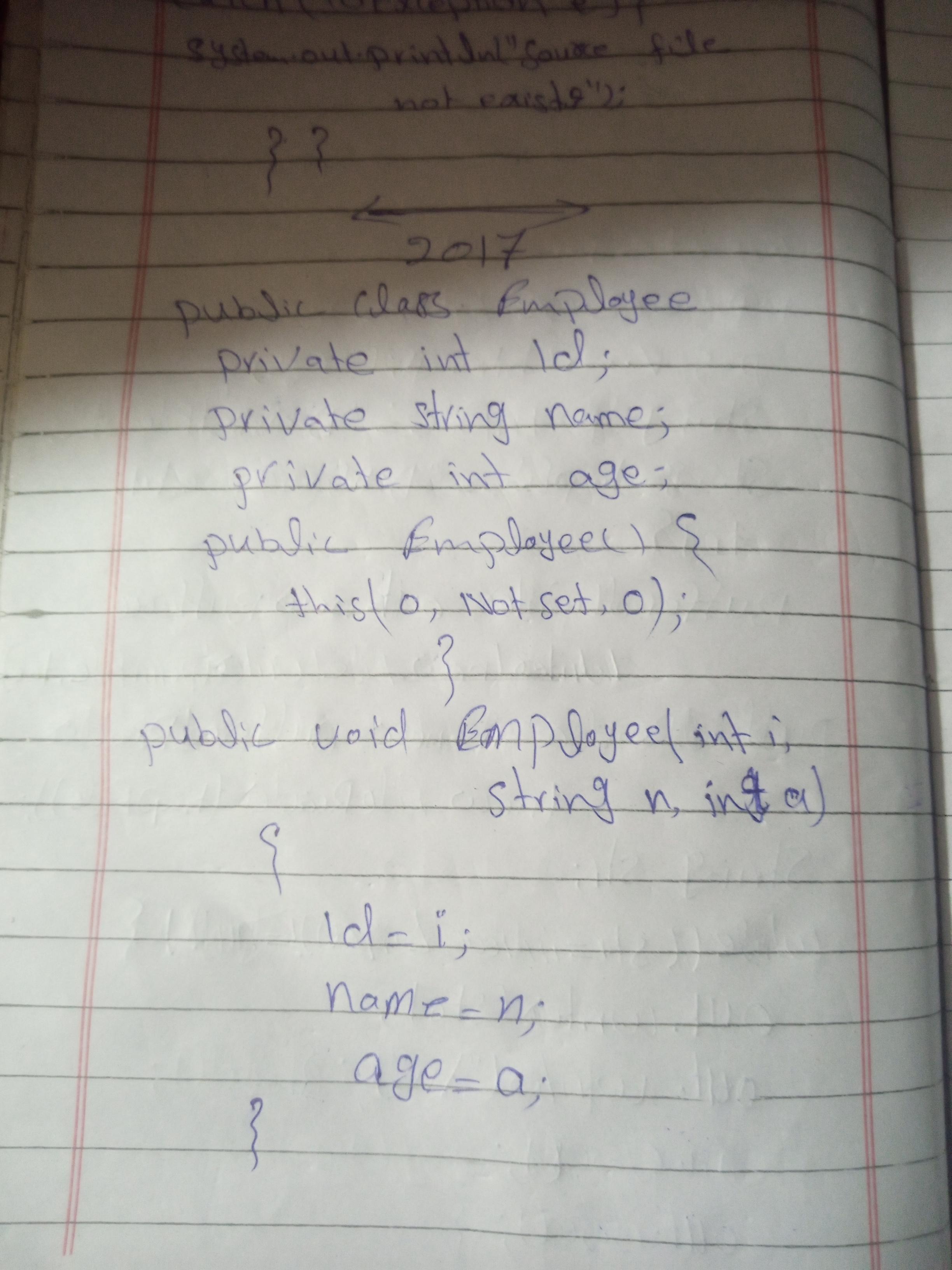
## **Why use Java interface?**

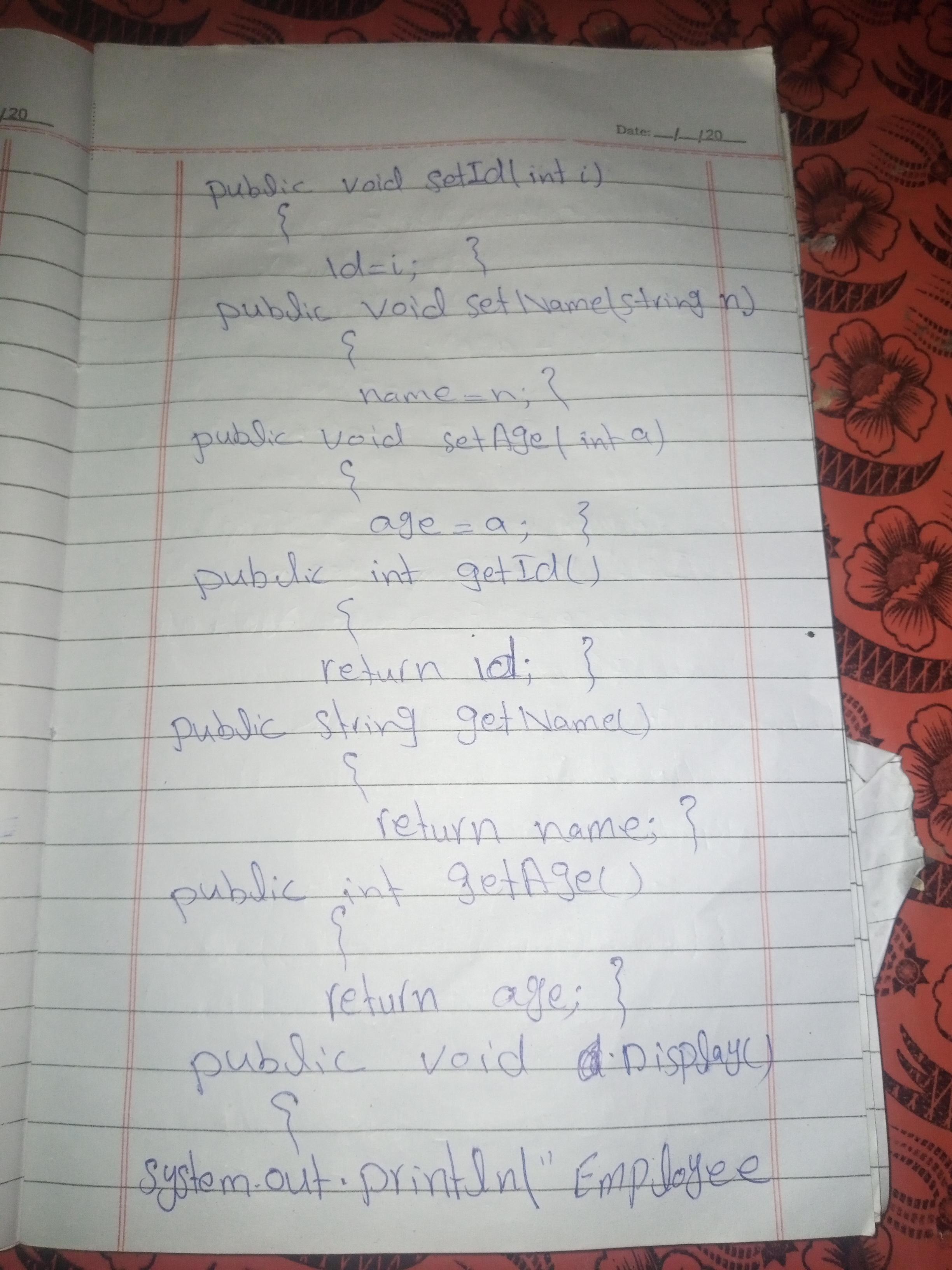
There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.



1. **interface** Drawable{
2. **void** draw();
3. }
4. //Implementation: by second user
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle **implements** Drawable{
9. **public** **void** draw(){System.out.println("drawing circle");}
10. }
11. //Using interface: by third user
12. **class** TestInterface1{
13. **public** **static** **void** main(String args[]){
14. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
15. d.draw();
16. }}





1. **interface** Drawable{
2. **void** draw();
3. }
4. //Implementation: by second user
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle **implements** Drawable{
9. **public** **void** draw(){System.out.println("drawing circle");}
10. }
11. //Using interface: by third user
12. **class** TestInterface1{
13. **public** **static** **void** main(String args[]){
14. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
15. d.draw();
16. }}

Interface Drawable{

Void draw();

}

//Implementation: by second user

Class Rectangle implements Drawable{

Public void draw(){System.out.println(“drawing rectangle”);}

}

Class Circle implements Drawable{

Public void draw(){System.out.println(“drawing circle”);}

}

//Using interface: by third user

Class TestInterface1{

Public static void main(String args[]){

Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()

d.draw();

}}

