PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

Course Code: CCE-121

SUBMITTED TO:

Dr.Md. Sumsuzzaman

Department of Computer & Communication Engineering
Faculty of Computer Science And Engineering

SUBMITTED BY:

Name:

ID: , Registration No:

Faculty of Computer Science and Engineering

Submission Date:

Course Code | **Course Title** | **July December** 2021

CCE-121 | Object Oriented Programming

Final Examination Question Solve

1.[A] Fill in the blanks in each of the following statements:

i) If a class declares constructors, the compiler will not create a default constructor

ii) The public methods of a class are also known as the class's ----- or public interface.

iii) Lists and tables of values can be stored in Arrays and Databases.

iv) The number used to refer to a particular array element is called the elements index (or subscript or position number).

v) A variable known only within the method in which it's declared is called a local variable.

vi) It's possible to have several methods with the same name that each operate on different types or numbers of arguments. This feature is called method overloading.

vii) Typically, for statements are used for counter-controlled repetition and while statements for sentinel-controlled repetition.

viii) Methods that perform common tasks and do not require objects are called Static methods.

[B] Write a Java statement or a set of Java statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99, using a for statement. Assume that the integer variables sum and

count have been declared.

```
public class Main
{
  public static void main(String[] args)
  {
    int sum = 0;
    for (int count = 1 ; count <= 99 ; count += 2) {
      sum += count;
    }


    System.out.println(sum);
  }
}
```

Output: 2500

b) Print the integers from 1 to 20, using a while loop and the counter variable i. Assume that the variable i has been declared, but not initialized. Print only five integers per line.

```
public class ForLoopExample {
    public static void main(String[] args) {
        int count = 0;

        for (int i = 1; i <= 20; i++) {
            System.out.print(i + " ");
            count++;
            if (count == 5) {
                System.out.println();
                count = 0;        }
        }
    }
}
```

<u>Output:</u>

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

[C]

i) Bytecode

ii) the machine executes the program's code

iii) We are allowed to use any name for a filename only when class is not public

iv) There are no restrictions on the number of classes that can be present in one Java program

[D] Write a Java program to create and display unique three-digit number using 1, 2, 3, 4. Also count how many three-digit numbers are there.

```
import java.util.Scanner;

public class Exercise39 {

 public static void main(String[] args) {
    int amount = 0;
    for(int i = 1; i <= 4; i++){
      for(int j = 1; j <= 4; j++){
        for(int k = 1; k <= 4; k++){
          if(k != i && k != j && i != j){
            amount++;
            System.out.println(i + "" + j + "" + k);
          }
        }
      }
    }
    System.out.println("Total number of the three-digit-number is " +
amount);
  }
}
```

Output:

```
123
124
132
134
142
143
213
214
231
234
241
243
312
314
321
324
341
342
412
413
421
423
431
432
Total number of the three-digit-number is 24.
```

2.[A] What are the various access specifiers in Java? Write an example of public access modifier.

⇒ There are four access specifiers in Java, namely: public, protected, private, and default.

Example of public access modifier:
```
public class MyClass {
   public int publicField = 10;

   public void publicMethod() {
      System.out.println("This is a public method.");
   }
}
```
Output:

[B] Write the rules of Constructor. What is the purpose of a default constructor? Explain with example

⇒ In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. It is a special type of method which is used to initialize the object.

- Rules for creating Java constructor
  There are two rules defined for the constructor.
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

- Default Constructor: A constructor is called "Default Constructor" when it doesn't have any parameter.
  Syntax of default constructor: <class_name>(){}
  The purpose of a default constructor is it used to provide the default values to the object like 0, null, etc., depending on the type.
- Example of default constructor:

```
class Student3{
int id;
String name;
void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
```

```
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

Output:

```
0 null
0 null
```

In the above class, we are not creating any constructor so compiler provides us a default constructor. Here 0 and null values are provided by default constructor.

[C] i)

```
public class Test
{
Test(int a, int b)
{
System.out.println("a = "+a+" b = "+b); }
Test(int a, float b)
{ System.out.println("a = "+a+" b = "+b);
}
public static void main (String args[])
{
byte a = 10;
byte b = 15;
Test test = new Test(a,b);
} }
```

Output of the Above Java program:

```
ii) class Test
{
int i; }
public class Main
{
public static void main (String args[])
{
Test test = new Test();
System.out.println(test.i);
} }
iii)
class Test
{
public static void main (String args[])
{
for(int i=0; 0; i++)
{
System.out.println("Hello PSTU CSE");
} } }
```

<u>Output of the Above Java program:</u>

[D] Write a Java program to print a pyramid using star pattern. Number of rows input from keyboard.

```
import java.util.Scanner;

public class StarPyramid {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows for the pyramid: ");
        int numRows = scanner.nextInt();
        for (int i = 1; i <= numRows; i++) {

            for (int j = 1; j <= numRows - i; j++) {
                System.out.print(" ");
            }


            for (int j = 1; j <= 2 * i - 1; j++) {
                System.out.print("*");
            }

            System.out.println();
               }
        }
}
```

Output:

```
Enter the number of rows for the pyramid: 5
    *
   ***
  *****
 *******
*********
```

3 [A] What is the static variable? Explain a java program with and without static variable.

⇒ A variable that is declared as static is called a static variable. It cannot be local. We can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

A Java program with static variable:

```
public class StaticVariableExample {

    // Static variable
    static int staticVariable = 0;
    public static void main(String[] args) {
        // Accessing the static variable
        System.out.println("Static Variable: " + staticVariable);

        // Modifying the static variable
        staticVariable = 10;

        // Accessing the static variable after modification
        System.out.println("Modified Static Variable: " + staticVariable);
    }
}
```

Output:

A Java program without static variable:

```
public class NonStaticVariableExample {

    // Instance variable
    int instanceVariable;

    public NonStaticVariableExample(int value) {
        // Constructor to initialize the instance variable
        instanceVariable = value;
    }

    public void displayValue() {
        // Display the instance variable
        System.out.println("Instance Variable: " + instanceVariable);
    }

    public static void main(String[] args) {
        // Create two objects of NonStaticVariableExample
        NonStaticVariableExample obj1 = new NonStaticVariableExample(5);
        NonStaticVariableExample obj2 = new NonStaticVariableExample(10);

        // Call the displayValue method on both objects
        obj1.displayValue();
        obj2.displayValue();
    }
}
```

Output:

[B] i) What is the difference between static (class) method and instance method?
ii) What are the main uses of this keyword?

⇒ **i)** Static (class) methods and instance methods in Java serve different purposes and have distinct characteristics.
Here are the key differences between them:

- Invocation:

  Static Method: These methods belong to the class itself and are invoked using the class name (e.g., ClassName.staticMethod()). They are not tied to any specific instance of the class.

  Instance Method: These methods are associated with instances (objects) of the class and are invoked on an object of the class (e.g., object.instanceMethod()). They operate on the data specific to that object.

- Access to Variables:

  Static Method: Static methods can only directly access other static members (static variables and static methods) of the class. They cannot access instance variables or instance methods directly.

  Instance Method: Instance methods can access both static and instance members of the class. They have access to the instance's state (instance variables) and can also call static methods.

- Use Cases:

Static Method: These are used for operations that don't depend on the specific state of any object but are related to the class as a whole. Common use cases include utility methods, factory methods, and methods that perform class-level operations.

Instance Method: These are used for operations that depend on the state of a specific object. Instance methods can access and modify instance variables, making them suitable for working with object-specific data.

- Memory Usage:

Static Method: They do not have access to instance-specific memory, as they are not associated with any particular object. They use memory for the class itself.

Instance Method: Each instance of the class has its own copy of instance variables, so instance methods consume memory for both the class and its individual instances.

ii) Usage of Java this keyword

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

[C] Define Object and Class. Write Object and Class Example: main outside the class and main within the class

Object: An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Object and Class Example: main outside the class

```
class Student{
 int id;
 String name;
}
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Object and Class Example: main within the class

```
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference
variable
  System.out.println(s1.name);
 }
}
```

Output:
```
0
null
```

[D] Write a Java program to sort an array of given integers using the Bubble sorting Algorithm
Original Array: [7, -5, 3, 2, 1, 0, 45]
Sorted Array: [-5, 0, 1, 2, 3, 7, 45]

```java
public class BubbleSortExample {
    static void bubbleSort(int[] arr) {
        int n = arr.length;
        int temp = 0;
         for(int i=0; i < n; i++){
                 for(int j=1; j < (n-i); j++){
                          if(arr[j-1] > arr[j]){
                                 //swap elements
                                 temp = arr[j-1];
                                 arr[j-1] = arr[j];
                                 arr[j] = temp;
                         }

                 }
         }

    }
    public static void main(String[] args) {
                int arr[] ={7, -5, 3, 2, 1, 0, 45};

                System.out.println("Array Before Bubble Sort");
                for(int i=0; i < arr.length; i++){
                        System.out.print(arr[i] + " ");
                }
                System.out.println();

                bubbleSort(arr);//sorting array elements using bubble sort

                System.out.println("Array After Bubble Sort");
```

```
            for(int i=0; i < arr.length; i++){
                System.out.print(arr[i] + " ");
            }

        }
}
```

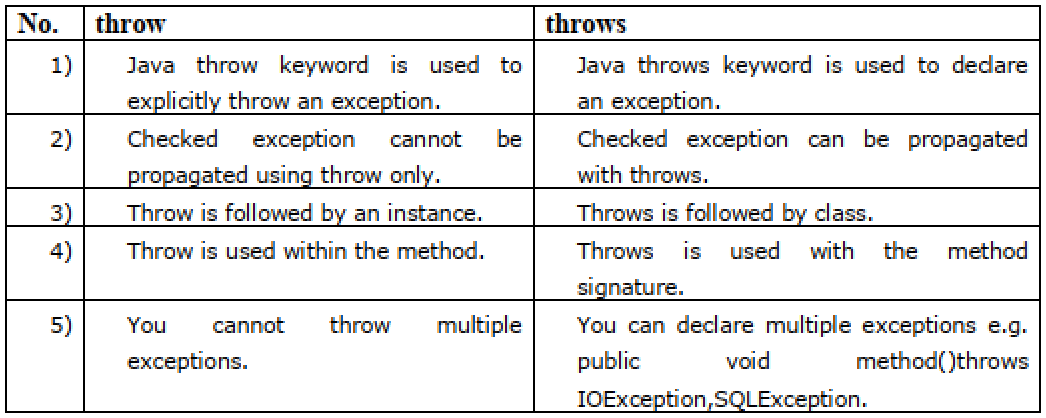
Output:

```
Array Before Bubble Sort
7, -5, 3, 2, 1, 0, 45
Array After Bubble Sort
-5, 0, 1, 2, 3, 7, 45
```

4.(a)

Differentiate between the **throw** and **throws** keyword.

→



No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

(b)

“Aggregation represents HAS-A relationship.”-explain with example.

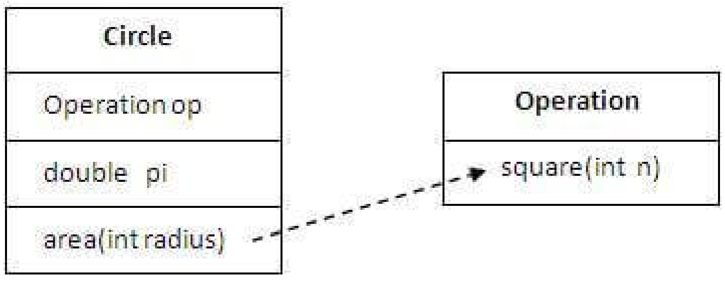
→

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Aggregation is for code reusability.

Example:

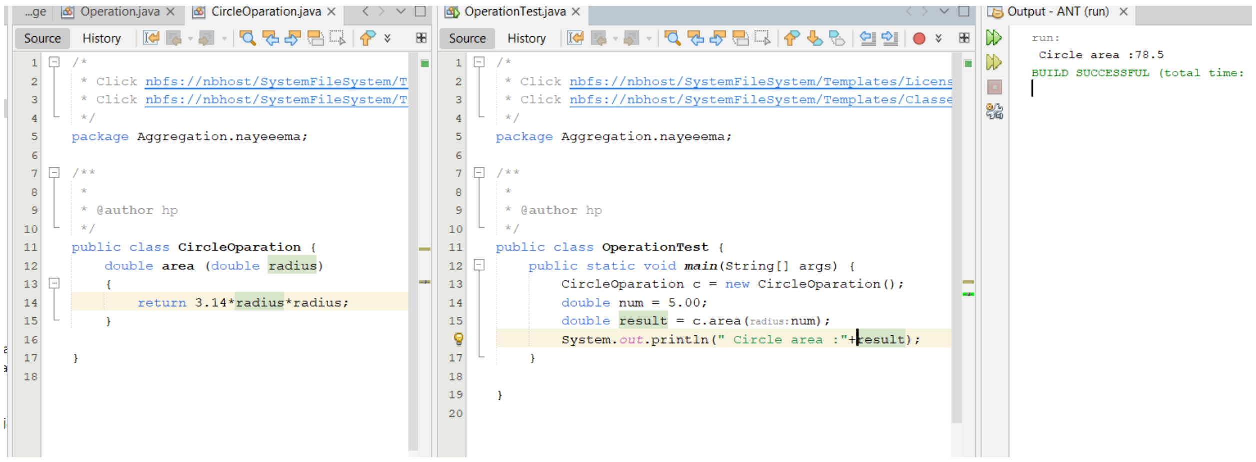


use Aggregation :

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

meaningful example of Aggregation:

In this example, Employee has an object of Address, address object contains its own information such as city, state, country etc. In such case relationship is Employee HAS-A address.



```java
/*
 * Click nbfs://nbhost/SystemFileSystem/T
 * Click nbfs://nbhost/SystemFileSystem/T
 */
package Aggregation.nayeeema;

/**
 *
 * @author hp
 */
public class CircleOparation {
    double area (double radius)
    {
        return 3.14*radius*radius;
    }

}
```

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licens
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classe
 */
package Aggregation.nayeeema;

/**
 *
 * @author hp
 */
public class OperationTest {
    public static void main(String[] args) {
        CircleOparation c = new CircleOparation();
        double num = 5.00;
        double result = c.area(radius:num);
        System.out.println(" Circle area :"+result);
    }

}
```

```
run:
 Circle area :78.5
BUILD SUCCESSFUL (total time:
```

(C)

Is it possible to make any class read-only or write-only in java? How?

→

Yes, It's possible to make read-only class and also make write-only class.

i) For read-only class

We can make a class read-only by making all of the data members private. The Read-only class means, we are talking about "IMMUTABLE" concept. How to make Read-only class and the various steps in given below:

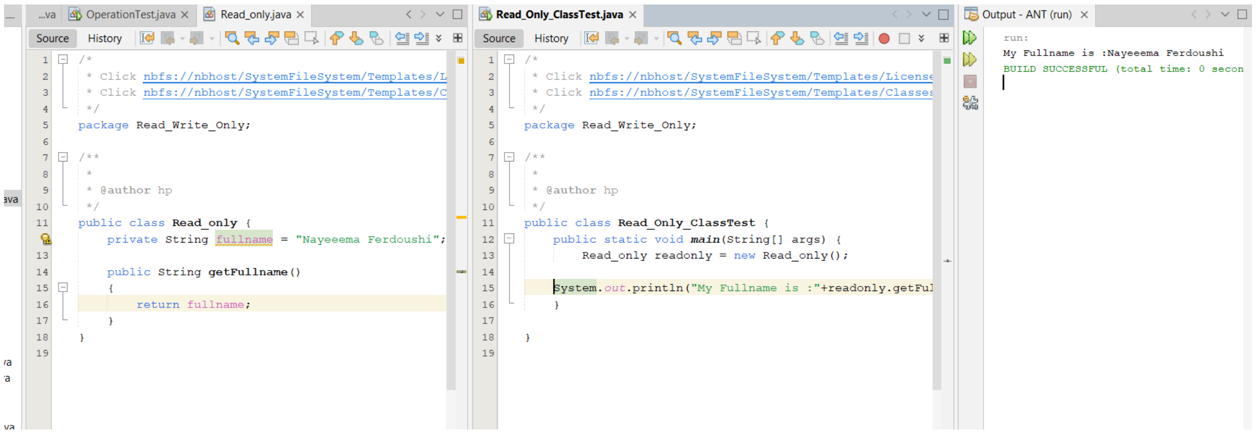
- As we know that "private" data member of the class is accessible in the same class only.

- Let suppose we want to access "private" data member of the class in outside class. So, in that case, we need to declare public "getter" methods.
- The objective of the getter method is used to view the private variable values.

Syntax :

public returntype getmethod();

The following example is :



```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/L
 * Click nbfs://nbhost/SystemFileSystem/Templates/C
 */
package Read_Write_Only;

/**
 *
 * @author hp
 */
public class Read_only {
    private String fullname = "Nayeeema Ferdoushi";

    public String getFullname()
    {
        return fullname;
    }
}
```

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/License
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes
 */
package Read_Write_Only;

/**
 *
 * @author hp
 */
public class Read_Only_ClassTest {
    public static void main(String[] args) {
        Read_only readonly = new Read_only();

    System.out.println("My Fullname is :"+readonly.getFul
    }

}
```

```
run:
My Fullname is :Nayeeema Ferdoushi
BUILD SUCCESSFUL (total time: 0 secon
```

ii) For Write-Only Class

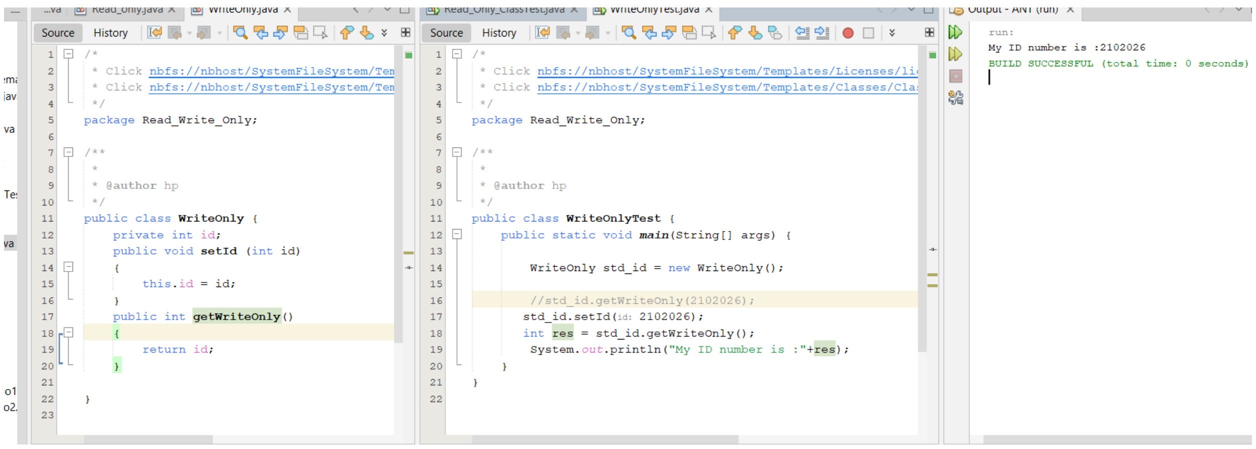
We can make a class Write-only by making all of the data members private. How to make Write-only class and the various steps in given below:

- As we know that "private" data member of the class is accessible in the same class only.
- Let suppose we want to access "private" data member of the class in outside class. So, in that case, we need to declare public "setter" methods.
- The objective of the set method is used to update or set the private variable values.

Syntax:

public return_type setmethod(data_type variable_name);

The following example is:



```
/*
 * Click nbfs://nbhost/SystemFileSystem/Tem
 * Click nbfs://nbhost/SystemFileSystem/Tem
 */
package Read_Write_Only;

/**
 *
 * @author hp
 */
public class WriteOnly {
    private int id;
    public void setId (int id)
    {
        this.id = id;
    }
    public int getWriteOnly()
    {
        return id;
    }

}
```

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/li
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Cla
 */
package Read_Write_Only;

/**
 *
 * @author hp
 */
public class WriteOnlyTest {
    public static void main(String[] args) {

        WriteOnly std_id = new WriteOnly();

        //std_id.getWriteOnly(2102026);
       std_id.setId(id: 2102026);
       int res = std_id.getWriteOnly();
        System.out.println("My ID number is :"+res);
    }
}
```

```
run:
My ID number is :2102026
BUILD SUCCESSFUL (total time: 0 seconds)
```

(D)

What is the use of instance initializer block while we can directly assign a value in instance data member?

→

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

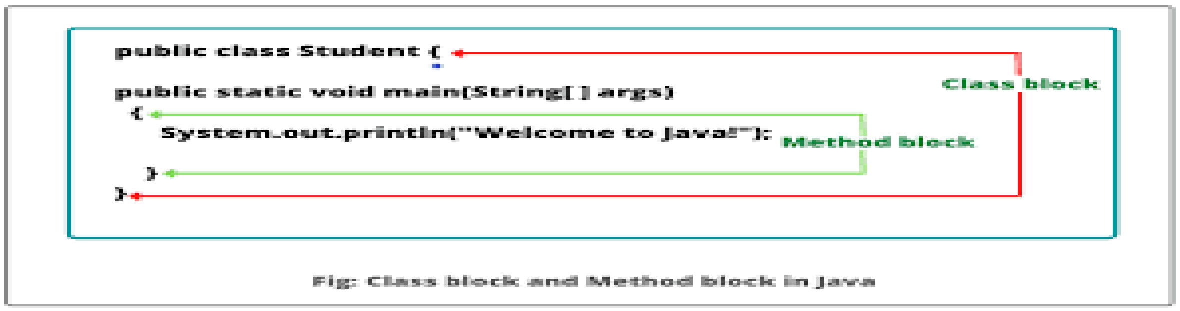

public class Student {
public static void main(String[ ] args)
{
System.out.println("Welcome to Java!");
}
}
Class block
Method block

Fig: Class block and Method block in Java

Example:

// Java Program to Illustrate Initializer Block

```
// Class
class Car {

	// Class member variable
	int speed;

	// Constructor
	Car()
	{

		// Print statement when constructor is called
		System.out.println("Speed of Car: " + speed);
	}
	// Block
	{
		speed = 60;
	}
	// Class member method
	public static void main(String[] args)
	{

		Car obj = new Car();
	}
```

}

Output: Speed of Car: 60

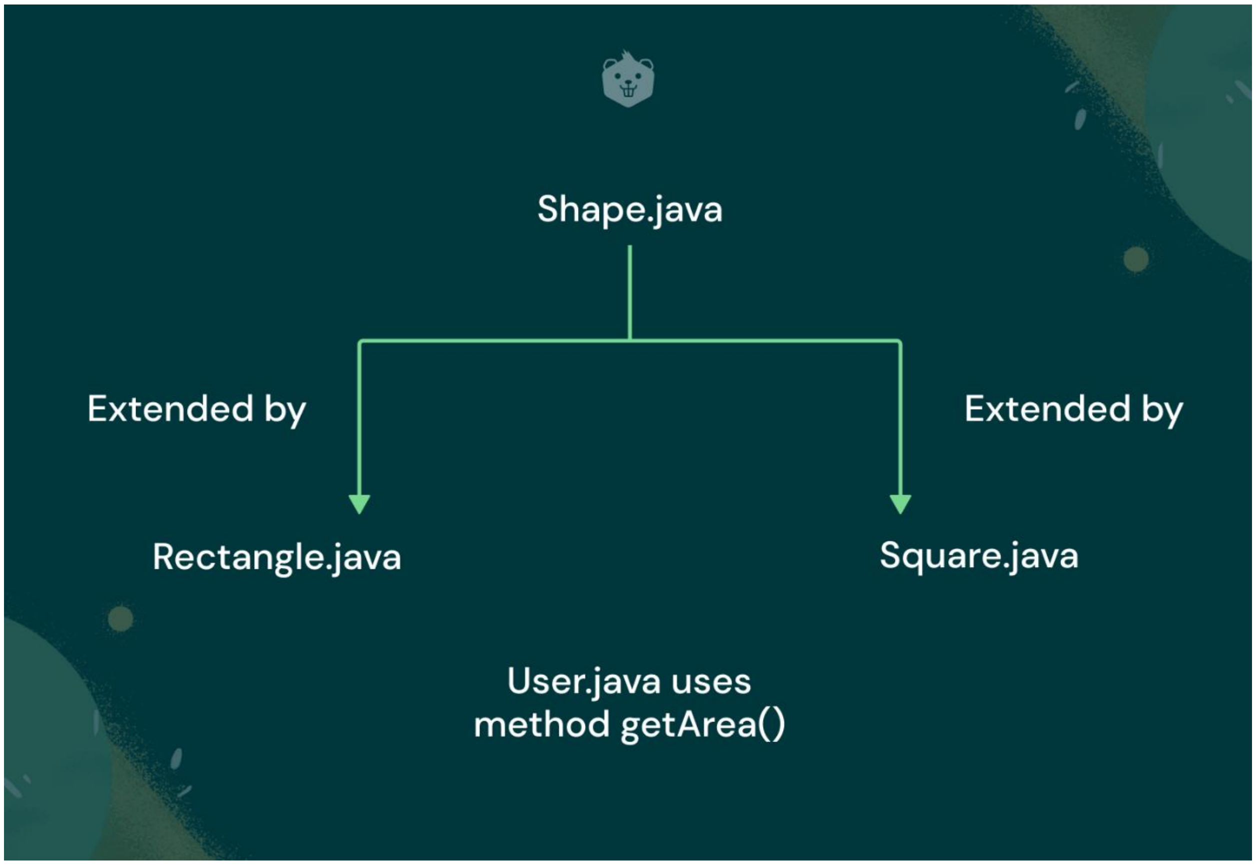
(E) How can you achieve abstraction in java?

→

Abstraction is a technique to identify the useful information that should be visible to a user and ignore the irrelevant details. Abstraction in Java can be achieve in two ways:

1. Using abstract classes

Abstract classes achieve partial abstraction as concrete methods can also be defined in them.

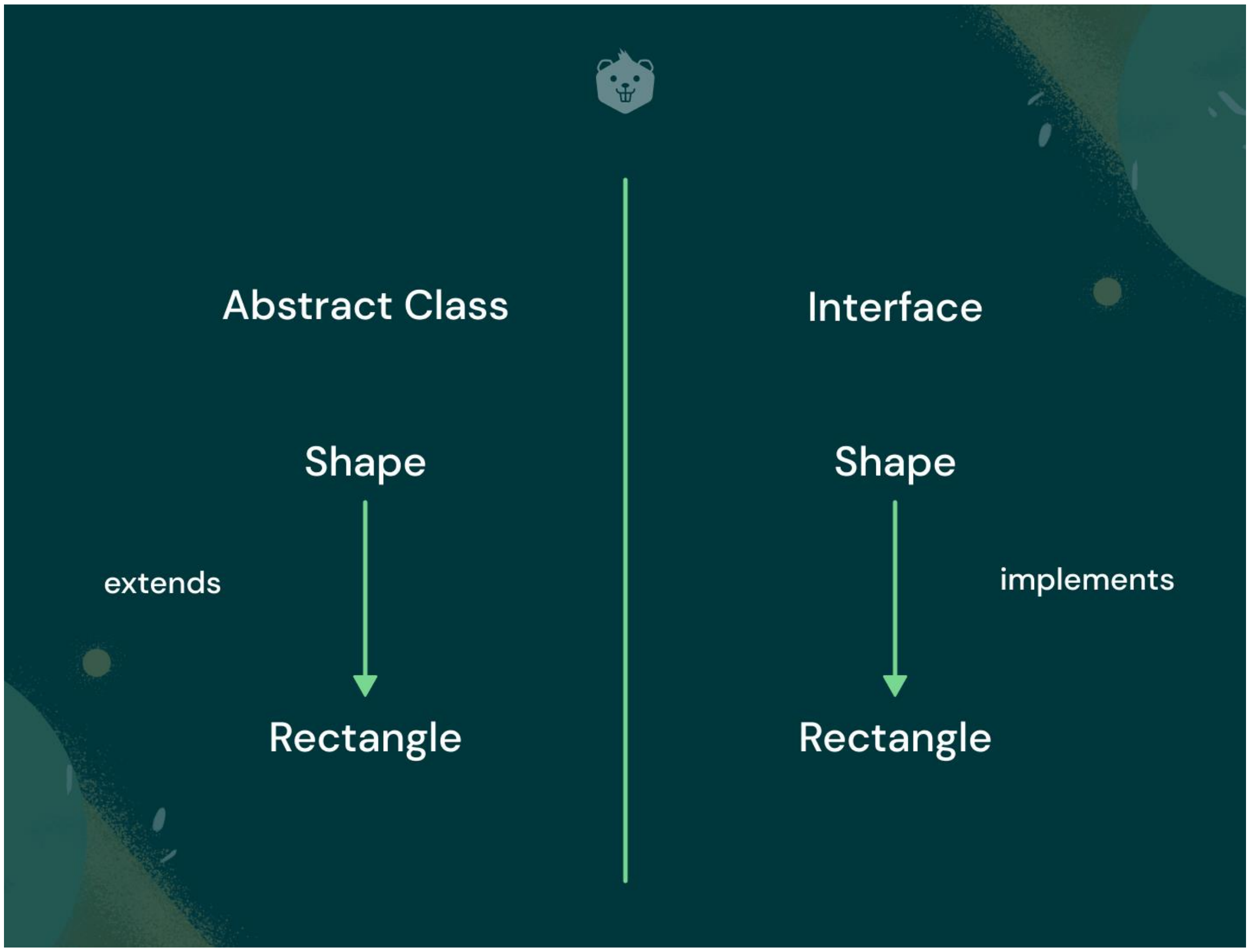


General Syntax:

```
public class abstract class_name{
        public abstract return_type method_name();
}
public class subclass_name extends abstract_class{
        public return_type method_name(){
                //method implementation
        }
}
```

2. Using interfaces

Interfaces achieve complete abstraction as only abstract methods can be defines in them.



General Syntax:

```
public interface interface_name
{
        public abstract return_type method_name();

}

class class_name implements interface_name{

        public return_type method_name(){

        //method implementation

        }

}
```

5(A)

Write a java program for demonstrating several thread states.

→

```
public class ThreadStateDemo {


   public static void main(String[] args) throws InterruptedException {
       // Create a new thread and start it
       Thread newThread = new Thread(new MyRunnable());
       System.out.println("Thread State: " + newThread.getState()); //
NEW
       newThread.start();
       Thread.sleep(100); // Give the thread some time to start


       System.out.println("Thread State: " + newThread.getState()); //
RUNNABLE


       // Create a thread that will block
       Thread blockedThread = new Thread(new
BlockedRunnable(newThread));
       blockedThread.start();
       Thread.sleep(100); // Give the blocked thread some time to start
```

```
        System.out.println("Thread State: " + blockedThread.getState()); //
BLOCKED

        // Create a thread that will wait
        Thread waitingThread = new Thread(new WaitingRunnable());
        waitingThread.start();
        Thread.sleep(100); // Give the waiting thread some time to start

        System.out.println("Thread State: " + waitingThread.getState()); //
WAITING

        // Create a thread that will wait with a timeout
        Thread timedWaitingThread = new Thread(new
TimedWaitingRunnable());
        timedWaitingThread.start();
        Thread.sleep(100); // Give the timed waiting thread some time to
start

        System.out.println("Thread State: " +
timedWaitingThread.getState()); // TIMED_WAITING

        // Wait for the threads to finish
        newThread.join();
        blockedThread.join();
        waitingThread.join();
        timedWaitingThread.join();

        System.out.println("Thread State: " + newThread.getState()); //
TERMINATED
        System.out.println("Thread State: " + blockedThread.getState()); //
TERMINATED
        System.out.println("Thread State: " + waitingThread.getState()); //
TERMINATED
        System.out.println("Thread State: " +
timedWaitingThread.getState()); // TERMINATED
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            // Do some work
        }
    }

    static class BlockedRunnable implements Runnable {
        private final Thread threadToBlock;
        BlockedRunnable(Thread threadToBlock) {
            this.threadToBlock = threadToBlock;
        }

        @Override
        public void run() {
            synchronized (threadToBlock) {
                // Block the thread
            }
        }
    }

    static class WaitingRunnable implements Runnable {
        @Override
        public void run() {
            synchronized (this) {
                try {
                    // Wait indefinitely
                    wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
              }
            }
          }

          static class TimedWaitingRunnable implements Runnable {
            @Override
            public void run() {
              synchronized (this) {
                try {
                  // Wait for 2 seconds
                  wait(2000);
                } catch (InterruptedException e) {
                  Thread.currentThread().interrupt();
                }
              }
            }
          }
        }
```

5.B. The statement "Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading" highlights an interesting aspect of Java and the Java Virtual Machine (JVM). Let's break down this statement and provide an example to justify it.

1. **Java Doesn't Allow Return Type-Based Overloading**:

   In Java, method overloading is a feature that allows you to define multiple methods in a class with the same name but different parameters. However, Java does not consider the return type of a method as a differentiating factor when determining which overloaded method to call. This means that you cannot have two overloaded methods that differ only in their return types. Java considers the method signature, which includes the method name and the parameter types, to determine which method to invoke.

   Here's an example illustrating this limitation

```
public class Example {
    // This is NOT allowed in Java
    // Two methods with the same name and different return types
    public int add(int a, int b) {
        return a + b;
    }

    public double add(int a, int b) {
        return (double)(a + b);
    }
}
```

In this example, attempting to declare two `add` methods with different return types (`int` and `double`) would result in a compilation error in Java because the return type is not considered when overloading methods.

JVM Always Allows Return Type-Based Overloading:

While Java itself does not allow return type-based overloading, the JVM (Java Virtual Machine) does not have this restriction. The JVM distinguishes between methods based on their fully qualified signatures, which include the return type. This means that if you compile a class with methods that differ only in their return types, the JVM will consider them as distinct methods.

Here's an example to illustrate this concept:

```
public class Example {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(int a, int b) {
        return (double)(a + b);
    }

    public static void main(String[] args) {
        Example example = new Example();
        int resultInt = example.add(2, 3);
        double resultDouble = example.add(2, 3);

        System.out.println("Result (int): " + resultInt);
        System.out.println("Result (double): " + resultDouble);
    }
}
```

In this case, even though Java doesn't allow such method overloading, the JVM allows it. When you run the `main` method, it will call the appropriate method based on the return type, and you will get different results for `resultInt` and `resultDouble`.

So, in summary, while Java's language rules do not permit return type-based method overloading, the JVM itself can handle such overloading by considering the return type as part of the method's signature. This can lead to unexpected behavior and should generally be avoided to prevent confusion in code.

5.C. In Java, multiple inheritance is a concept where a class inherits properties and behaviors from more than one class. Java supports multiple inheritance through interfaces but not through classes. This design decision was made to avoid the complexities and ambiguities that can arise from traditional multiple inheritance via classes.

Here's why multiple inheritance is supported through interfaces in Java, but not through classes:

1. **Diamond Problem and Ambiguity**:

   The main issue with multiple inheritance using classes is the "diamond problem." The diamond problem occurs when a class inherits from two classes that have a common ancestor. If both parent classes provide an implementation for the same method, it can be unclear which method should be called in the derived class. This ambiguity leads to code complexity and potential errors.

   Here's a simplified example of the diamond problem:

   Css:

A

/
B C \ / D

csharp

If both `B` and `C` have a method with the same name, say `foo()`, and class `D` inherits from both `B` and `C`, it's unclear which `foo()` method should be called when you invoke it on an instance of `D`.

2. **Java Interfaces**:

Java addresses the diamond problem by allowing multiple inheritance through interfaces. Interfaces in Java are like contracts that define a set of methods that implementing classes must provide. Multiple interfaces can be implemented by a single class, allowing it to inherit multiple sets of method contracts without the ambiguity that arises in multiple inheritance through classes.

Here's an example:

```java

interface A {

   void methodA();

}

interface B {
    void methodB();
}


class MyClass implements A, B {
    public void methodA() {
        // Implementation for methodA
    }


    public void methodB() {
        // Implementation for methodB
    }
}
```

In this example, the `MyClass` class implements both interfaces `A` and `B`, providing implementations for their respective methods without any ambiguity.

By allowing multiple inheritance through interfaces, Java provides a way to inherit behavior from multiple sources while avoiding the complexities and ambiguities associated with traditional multiple inheritance using classes. This design choice helps maintain code clarity and prevents the diamond problem from arising in Java programs.

5.D. In Java, a blank final variable is a final variable that is not assigned a value when it is declared. Instead, it must be initialized later, typically within a constructor. Once initialized, the value of a blank final variable cannot be changed.

Here's how you can initialize a blank final variable in Java:

1.Using a Constructor:

> The most common way to initialize a blank final variable is within a constructor. You can assign a value to the blank final variable in the constructor of the class. Once the blank final variable is assigned a value in the constructor, it cannot be modified.

```
public class MyClass {

    // Blank final variable

    final int myBlankFinalVariable;
    public MyClass(int value) {

        // Initialize the blank final variable in the constructor

        myBlankFinalVariable = value;

    }

}
```

In this example, the `myBlankFinalVariable` is a blank final variable, and it's initialized with a value in the constructor.

2.**Using an Instance Initialization Block**:

You can also initialize a blank final variable using an instance initialization block. An instance initialization block is a block of code enclosed in curly braces that is executed when an instance of the class is created. This block can be used to initialize the blank final variable.

```
public class MyClass {

    // Blank final variable

    final int myBlankFinalVariable;



    // Instance initialization block

    {

        // Initialize the blank final variable in an instance initialization block

        myBlankFinalVariable = 42;

    }

}
```

In this example, the `myBlankFinalVariable` is initialized within an instance initialization block.

It's important to note that once a blank final variable is initialized, it cannot be changed. If you attempt to reassign it in any method other than the constructor or instance initialization block, you will get a compilation error.

In summary, you can initialize a blank final variable in Java by assigning a value to it within a constructor or an instance initialization block. Once initialized, the variable becomes immutable and cannot be modified.

6. [A] Output of a java program

(i)
```
class Dog{

public static void main(String[] args) {

Dog d = null;

System.out.println(d instanceof Dog);

}

}
```

Ans: false

(ii)
```
class A {

protected void msg() {

System.out.println("Hello java");

}

}


public class Simple extends A {

void msg() {

System.out.println("Hello java");

}


public static void main(String[] args) {

Simple obj = new Simple();

obj.msg();

}

}
```

Ans: Compiler error! But it'll print "Hello java" if protected is removed from the method msg of class A.

(iii)
```
public class Sample {

public static void main(String[] args) {

try {
```

```
int data = 100 / 0;

}

catch (ArithmeticException e)

{

System.out.println(e);

}

System.out.println("rest of the code...");

}

}
```

Ans: java.lang.ArithmeticException: / by zero

rest of the code...

(iv)
```
class Animal {

void eat()

{

System.out.println("eating...");

}

}

class Dog extends Animal {

void bark()

{

System.out.println("barking...");

}

}

class Cat extends Animal {

void meow()

{

System.out.println("meowing...");

}

}
```

```
public class TestInheritance3 {

public static void main(String[] args) {

Cat c = new Cat();

c.meow();

c.eat();

c.bark();

}

}
```

Ans: Compiler error (bark method not available).

Without c.bark() it will print meowing… and eating…

6[B] How to access package from another package?

Ans: In java to access package from another one, we use the import command. For example,

```
import packagename.classname;
```

6[C] What is the purpose of join method?

Ans: join method is used when one thread waits for an another thread to finish it's job. For example, if we want to finish the process of ThreadB and then continue, we can use,

```
ThreadB.join();
```

6[D] How to perform two tasks by two threads?

Ans: To perform two task, we can two separate threads and assign each of them two different task and start them. In this way those two will start working simultaneously.

6[E] What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.

- **Preemptive scheduling** allows the scheduler to take a running thread away from the CPU and give it to another thread, even if the first thread has not finished its execution. This is done based on the priority of the threads. A thread with a higher priority will preempt a thread with a lower priority.
- **Time slicing** is a type of preemptive scheduling where each thread is given a certain amount of time to run on the CPU. After the time slice expires, the thread is preempted and another thread is given a chance to run. This process continues until all of the threads have had a chance to run.

The main difference between preemptive scheduling and time slicing is that in preemptive scheduling, the scheduler can preempt a running thread at any time, while in time slicing, the thread is only preempted after its time slice expires.