IT-23046

1.
```
import java.io.file;
import Java.io.Printwriter;
import Java.util.Scanner;
Public Class SumofNatualNumber{
Public stadic void main(String[]args){
try{
file inputfile = newfile ("input.txt");
Scanner scanner = new Scanner(import.file);
Scanner.useDelimiter(",")
int maxNumber=Integer.MINVALVE;
while (scanner.hasNextInt()){
int num = scanner.nextint();
if (num> maxNumber){
maxNumber=num;
}
scanner.close();
int sum=(maxNumber* (maxNumber+1)/2;
Printwriter write =new printwriter ("Output.txt);
writer.println(Sum);
writer close();
system.out.println("Result written to output.txt);
} catch (Exption c){
}}
```

2. Difference between Static and Final fields and methods in Java.

Static: Belongs to the class rather than instances
Memory Allocation: Stored in the class memory.
Modification: Can be changed.
Method: Can be called using the class name.
fields: Shared across all object of the class.

Final:
Definition: Used to declare constant or prevent method avoiding.
Memory Allocation: Stored separately for each instance.
Modification: Cannot be modified after initialization.
Method: Prevents method avoiding when used with methods.
Fields: Cannot be reassigned ones initialized.

3. Here is a Java program to find all factorion numbers within a given range

```
import java.util.Scanner;
public class factorionNumbers{
private static final int[] FACTORIALS=new int[]
static{
FACTORIALS[0]=1;
for(int i=1; i<10; i++){
FACTORIALS[i]=i* FACTORIALS[i-1];
}
}
private static boolean isFactorion(int num){
  int sum=0, temp=num;
  while(temp>0){
   int digit = temp %. 10;
    sum+=factorials[digit];
    temp /= 10;
  }
  return sum == num;
}
```

```
public static void main (String[] args){
 Scanner scanner = new scanner (system.in);
system.out.print(" Enter the lower bound of the range:");
int lower = scanner. nextInt();
system.out.print ("Enter the upper bound of the range:");
int upper = scanner.nextInt ();
system. out.println("Factorion number in the range:");
boolean found=false;
for (int i=lower; i<=upper; i++){
   if (isFactorion(i)){
     system. out. print (i + " ");
      found=true;
     }
  }
  if (!found ){
     system.out.println ("None found in the given range. ");
     }
    scanner.close();
   }
}
```

4. Difference Among class, local and instance variables.

Class variable:

Defination: Shared across all instances of a class.
Scope: class level (shared by all instances)
Lifetime: Exists as long as the class is in memory.
Access method: Accessed using class Name.variable-name.

Instance Variable:

Defination: Unique to each instance of a class.
Scope: Instance level (Specific to an object)
Lifetime: Exist as long as the instance exists.
Access method: Accessed using self.variable-name.

Local Variable:

Defination: Defined inside a method or function.
Scope: Limited to the function which its declared.
Lifetime: Exists only during function execution.
Access method: Directly used within the function.

5. A Java program that defines a method to calculate the sum of all elements in an integer array.

```
public class ArraySumCalculator {
public static int calculateSum(int[] numbers){
  int sum = 0;
  for (int num : numbers){
   sum += num;
   }
  return sum;
}
public static void main(String[] args){
  int[] mArray = {10, 20, 30, 40, 50};
  int result = calculateSum(mArray);
  System.out.println("Sum of array elements: " + result);
  }
}
```

6. Access Modifiers in Java define the visibility and accessibility of classes, methods and variables. They control which part of a program can access certain elements, ensuring data encapsulation and security.

Java provides four access modifiers:

1. (Public) - Accesible from anywhere
2. (Private) - Accesible only within the same class.
3. (Protected) - Accesible within the same package and subclasses.
4. (Default) - Accessible only within the same package.

Types of variables of in Java:

Java has three types of variable:

1. Local variables: Declared insid a method can stolen or block.
2. Instance variables: Declared inside a class but outside methods, each object has its own copy
3. Class variables: Declared using static shared among all instances of a class.

7. Here is a Java program to find the smallest positive root of a quadratic equation of the form $ax^2+bx+c=0$:

```
import java.util.Scanner;
public class QuadraticSolver{
public static void main(String[] args){
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a, b, and c: ");
int a = scanner.nextInt();
int b = scanner.nextInt();
int c = scanner.nextInt();
double d= b*b - 4*a*c
if(d<0){
   System.out.println("No real roots.");
} else{
double r1 = (-b+Math.sqrt(d))/(2*a);
double r2 = (-b-Math.sqrt(d))/2*a);
if (r1>0 && r2>0){
System.out.println("Smallest positive root: "+Math.min(r1, r2))
```

```
} else if (r1>0){
    system.out.println ("Smallest positive root: "+r1);
} else if (r2>0){
    system.out.println ("Smallest positive root: "+r2);
} else{
    system.out.println("No positive roots.");
    }
}
scanner.close();
}
}
```

5. Here is a Java program that can determines if each character in a given string is a letter, a white-space or a digit.

```
public class characterClassifier{
public static void classifycharacters(char[] characters){
for (char c: characters){
if (character.isletter(c)) {
  system.out.println(" " + ct " ' is a letter. " );
}else if (character.iswhitespace(c)){
  system.out.println (" ' " + ct " ' is a whitespace. ");
} else if (character.isDigit(c)) {
  system.out.println(" ' " + ct " ' is a digit. ");
} else{
      system.out.println(" ' " + ct " ' is something else ");
    }
  }
}
```

```
public static void main(String[] args){
    char[] inputArray = {'a', '.', '1', 'b', '7', ' '};
    classifyCharacters(inputArray);
  }
}
```

Passing an Array to a Function in Java:

In Java we can pass an array to a method by specifying the array type as the parameter (e.g; char[] characters). The array is passed by reference, meaning any changes to the array within the function will affect the original array.

9. Method overriding in Java:

Defination: Method overriding allows a subclass to provide its own implementation of a method already defined in the superclass.

How it works: The subclass method must have the same signature as the superclass method. The subclass version replaces the superclass method when called on an object of the subclass.

Example:

```
class Animal {
    void makesound() {
        system.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @override
    void makesound() {
        system.out.println("Dog barks");
    }
}
```

super keyword :-

super is used to call superclass method or constructor

Example:

```
class Dog extends Animal{
  @Override
  void makesound (){
    super.makesound ();
    system.out.println("Dog barks");
  }
}
```

Issue with Overriding:

1. Syntax mismatch : The method signature must be the same.
2. The subclass method can have more restrictive access than the superclass method.
3. A final method cannot be overridden.
4. Constructor are not inherited and cannot be overridden though the subclass can call superclass constructor using super.

10. Difference between static and Non-static members;

Static Members	Non-static Members
Belongs to the class, shared by all instances	Belongs to an instance of the class
Allocated memory once for the class, not for each object	Allocated memory for each object.
Can be accessed without creating an object	Must be accessed through an object of the class.
Used for values or methods common to all instances	Used for instance-specific values or methods.

Example:

```
class Example {
  static int staticVar = 10;
  int nonStaticVar = 20;
  static void staticMethod () {
   System.out.println("Static method");
  }
   void nonStaticMethod() {
   System.out.println ("NonStaticMethod");
}
```

```
public static void main (String[] args){
 System.out.println(Example.staticVar);
 Example.staticMethod();
 Example obj = new Example();
 System.out.println(obj.nonStaticVar);
 obj.nonStaticMethod();
 }
}
```

Here is a Palindrome checker program

```
public class PalindromeChecker{
   static boolean isPalindrome(String str) {
     return str.equals(new StringBuilder(str).reverse().toString());
   }
 static boolean isPalindrome(int num){
     int original = num, reversed = 0;
       while(num != 0){
        reversed = reversed * 10 + num % 10;
        num /= 10;
       }
       return original == reversed;
     }
```

```
public static void main (String [] args){
 system.out.println("madam is palindrome:" +is palindrome("madam")
 system.out.println("121 is palindrome:" +isPalindrome(121));
 }
}
```

Output:

madam is palindrome; true

121 is palindrome; true

11. Class Abstraction & Encapsulation

1. Abstraction: Hiding implementation details and exposing only necessary functionality. Achieved using abstract classes and interfaces.
2. Encapsulation: Wrapping data (variables) and method into a single unit (class) and restricting direct access using private modifiers.

Example:

```
// Abstraction using abstract class
abstract class Animal{
  abstract void makeSound();
class Dog extends Animal{
private String name;
  Dog(String name){
  this.name=name;
  }
  void makeSound(){
  System.out.println(name+"barks");
  }
    public String getName(){
    return name;
  }
```

```
public class main{
    public static void main(String[] args){
        Dog d = new Dog("Buddy");
        d.makeSound();
        System.out.println("Dog's name: " + d.getName());
    }
}

// Abstract class
abstract class Vehicle{
    abstract void start();
}
class Car extends Vehicle{
    void start(){
        System.out.println("Car starts with key");
    }
}
interface Flyable{
    void fly();
}
class Plane implements Flyable{
    public void fly(){
        System.out.println("plane flies in the sky");
    }
}
```

12. Here is a Java program implementing the given requirements using inheritance.

```
class Baseclass{
  void printResult(String result){
    system.out.println(result);
  }
}
class sumClass extends BaseClass{
double computeSum(){
  double sum=0;
  for(double i=1.0; i>=0.1; i=0.1) sum+=i;
   return sum;
  }
}
class DivisorMultipleClass extends BaseClass{
  int gcd(int a, int b){ return b==0? a: gcd(b, a%b);}
    int lcm(int a, int b){return (a*b)/gcd(a,b);}
}
class NumberConversionClass extends Baseclass{
  String convert(int num, int base){ return Integer.toString
(num, base).toUpperCase();}
```

```
class customprintclass extends Baseclass{
 void pr (String label, String value){
  system.out.printfln ("%.s : %.s%.n", label, value);}
}
public class Mainclass{
 public static void main(String[] args){
    Sumclass sumobj = new sumclass();
    system.out.println("Sum of series: "+sumobj.computeSum());
 DivisorMultipleclass divobj = new DivisorMultipleClass();
  system.out.println("GCD : "+divobj.gcd(12,18)+", LCM : "+divObj.lcm
                                                    (12,18));
 NumberConversion class numObj = new NumberConversionClass();
 system.out.println("Binary: "+numObj.convert(10,2)+", Hex: "+num
```

13. Here is the Java program that works including the class contains the main method given the UML scenario below:

```
import java.util.Data;
class GeometricObject{
 private string color;
 private boolean filled;
 private Data dataCreated;
 public GeometricObject(){
   this.dataCreated = new Date();
 }
 public GeometricObject(String color, boolean filled){
   this();
   this.color = color;
   this.filled = filled;
 }
 public string getcolor(){return color;}
 public void setcolor(String color){this.color=color;}
 public boolean isFilled(){return filled;}
 public void setFilled(boolean filled){this.filled=filled;}
 public Date getDateCreated(){return dateCreated;}
```

```
@override
public string toString (){
return "color: ' + color+ ", filled: "+filled+ ", created on:
                                        +data created;
  }
}

class Circle extends GeometricObject {
 private double radius;
 public Circle (){}
 public Circle (double radius){
   this.radius = radius;
 }
 public Circle(double radius, string color, boolean filled)
 super (color, filled);
 this.radius = radius;
 }
 public double getRadius(){return radius;}
 public void setRadius (double radius){ this.radius=radi
 public double getArea(){return Math.PI * radius.radius;}
 public double getPerimeter (){return 2 * Math.PI *radius;}
 public double get Plameter(){ return 2*radius;}
 public void printCircle(){ system.out.println("Circle Radius=" +
}       radius);}
```

```
class Rectangle extends GeometricObject{
private double width, height;
public Rectangle(){}
public Rectangle(double width, double height){
this.width = width;
 this.height = height;
}
public Rectangle(double width, double height, String color, boolean
 filled){
super(color, filled);
this.width = width;
this.height = height;
}
public double getWidth(){return width;}
public void setWidth(double width){this.width = width;}
public double getHeight(){return height;}
public void setHeight(double height){this.height = height;}
public double getArea(){return width*height;}
public double getPerimeter(){return 2*(width+height);}
}
```

```
// Main class
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle(5, "Red", true);
        System.out.println("Circle Area: " + c.getArea());
        Rectangle r = new Rectangle(4, 6, "Blue", false);
        System.out.println("Rectangle Area: " + r.getArea());
    }
}
```

14. The BigInteger class in Java is used for handling very large integers that exceed the range of primitive data types like int or long. It provides operations for modular arithmetic GCD calculation, and more.

Here is a program to calculate the factorial of any integer using BigInteger.

```
import java.math.BigInteger;
public class Factorial {
 public static BigInteger factorial(int n){
    BigInteger result = BigInteger.ONE;
   for(int i=2; i<=n; i++){
     result=result.multiply(BigInteger.valueOf(i));
    }
    return result;
 }
 public static void main (String[]args){
  int number = 50;
  System.out.println(factorial(number));
  }
}
```