23034

1. import java.io.*;
import java.util.*;
publie statie void main()
publie elass asumNatural Numbers {
publie statie void main (String [] args) throws IOException
scanner se=new Scanner(new File("input.txt")).{
se.useDelimiter(",").
int highest = Integer.MIN_VALUE;
while (se.hasNext()){
int num = se.hasnextInt();
if (num > highest) {
highest = num;
}
}
se.close();
int sum = highest * (highest + 1)/2;
printWriter uniter = new PrintWriter(
new File ("output.txt"));
uniter.printf (sum+",").
uniter.close();
}
}

2. Differences between static and final fields and methods:

Static	Final
1. A member (field or method) marked as a static belongs to the class rather than any specific instance.	1. A member marked as final belongs to instances rather than any specific class.
2. Applies to variables, methods, blocks, nested classes.	2. Applies to variables, methods, classes.
3. Shared by all instances of the class. A single copy exists in memory.	3. Once assigned its value cannot be changed if its an object reference then the reference cannot be changed
4. Can be called without creating an object (class method)	4. cannot be overriden in subclasses but can be inherited
5. Stored in the method	5. Stored in top of heap stack.

Example:

23039

```
class staExample {
    static int count=0;
    static void display(){
     system.out.println("static method called");
    }}
public class main{
 public static void main(String[] args){
  System.out.println(static Example.count);
  stati Example.display(); // works
  static Example obj=new staticExample();
        System.out.println(obj.count);
    }
  }
```

Output:

```
0
static method called
0
```

31. import java.text.SimpleDateFormat (2303 q)
import java.util.Date;
public class CurrentDateTime_31{
public static void main(String[] args){
SimpleDateFormat sdf = new SimpleDateFormat
("yyyy-MM-dd HH:mm:ss");
Date now = new Date();
System.out.println("Current date and
time:" + sdf.format(now));
}
}

32. public class counterClass_32{
private static int instanceCount=0;
public CounterClass_32() {
instanceCount++;
if(instanceCount>50){
instanceCount=0;
System.out.println("Object count
exceeded 50, resetting to 0.");
}
}

```
public static int getInstanceCount(){
    return instanceCount;
}
public static void main (String [] args){
    for(int i=0; i<100; i++){
        new counterClass_32();
        System.out.println("Instance count:" +
                counterClass_32.getInstanceCount()
    }
}
```

11.

Abstractions: Class abstraction is the process of simplifying complex reality by modeling only the essential attributes and behaviours of an object while ~~es~~ hidding unnecessary details. It focuses on "what" an object does rather "how" it does it.

Example: A vehicle class may be define as abstract method start(), but each subclass (car, bike) provides it's own implementation.

Encapsulation: Encapsulation is the building of data (attributes) and methods (behaviors) that operate on that data within a single unit. It also involves controlling the access to the internal data of an object, preventing direct modification from the outside. This is often achieved through access modifiers like private, protected and public.

Example: A BankAccount class has a balance variable marked private and users can access it only through getbalance() and deposite() methods. These methods can include logic to validate the operations and maintain the integrity of the data.

Difference between abstract class and Interface-

Abstract	Interface
1. Abstract class can have both abstract and concrete methods.	1. Interface contains only abstract methods.
2. Can have intance variable with any access modifier.	2. Can have only public, static, final constants (no intance variables).
3. Can have constructors.	3. Cannot have constructor.
4. Example: abstract class Animal{}	4. Example: Interface Animal{}

33. import java.util.*;
public class Main{
public static void main(String[] args){
Scanner scanner = new Scanner(System.in);
String s = scanner.next();
List<Integer> v = new ArrayList<>();
while(scanner.hasNextInt()){
v.add(scanner.nextInt());
}
Collections.sort(v);
if(s.equals("smallest")){
System.out.println(v.get(0));}

```
else { System.out.println(v.get(v.size()-1));
  }
  scanner.close();
  }}
```

23034

36.
```
interface Alpha{
    void method1();
    void method2();}
interface Beta{
    void method3();
    void method4();}
abstract class AbstractBase implements Alpha{
    abstract void method5();}
class FinalClass extends AbstractBase
    implements Beta{
    @override
      public void method(){
    System.out.println("Method 1 from Alpha");
    @override
    public void method2(){
    System.out.println("Method 2 from
          Alpha");
```

23038

```
@override
pubk void ●method 3(){
@System.out.println("Method 3 from Beta");
}
@ovennide
 public void method 4(){
 System.out.println("Method 4 from Beta");}
@ov.ennide
  void method 5(){
System.out.println("Method 5 from Abstract
Base");}}
Public class Maing
      public static void main(string[] args){
       Final Class finalclass=neco FinalClass();
       final Class.method 1();
       final Clas.method 2();
       final Class.method 3();
       final class.method 4();
       final class.method 5();
      }
}
```

28. Java's Garbage Collection is a feature that automatically manages memory by cleaning up objects that are no longer in use. Java provides different types of garbage collectors, each with its pros and cons:

1. Serial GC: Uses a single thread for garbage collection. Good for small applications or single-core systems.
2. Parallel GC: Uses multi-~~core~~ple threads for faster garbage collection. Good for multi-core systems and applications that need high throughput.
3. CMS GC: Runs garbage collection concurrently with the application to minimize pauses.
4. G1 GC: Deivides memory into ~~garbages~~ regions and cleans up the most filled regions first.

23034

29)
```
import java.io.*;
import java.util.*;
publie statie void main(
 public class SumNatural Numbers {
    public static void main (String [] args) throws IOException
      scanner se=new Scanner(new File("input.txt"));{
      se.useDelimiter(",").
      int highest = Integer.MIN_VALUE;
      while (se.hasNext()){
         int num = se.hasnextInt();
         if (num>highest){
             highest= num;
         }
      }
      se.close();
      int sum= highest *(highest +1)/2;
      printWriter writer= new PrintWriter(
                          new File("output.txt"));
      writer.print (sum+",");
      writer.Close();
    }
}
```

23.3.24

16. Polymorphism in Java:

Polymorphism is the ability of an object to take on many forms. In java, it allows a single method or class to operate on objects of different types. It is achieved through method overriding and method overloading.

* Dynamic method dispatch

It is the mechanism by which a call to an overriden method is resolved at runtime rather that compile time. It is the foundation of runtime polymorphism in Java.

Example:

```
class Animal {
    void sound() {
    System.out.println("Animal make a sound");
    }
}

class Dog extends Animal {
  @override
```

23.34

```
void sound(){
  System.out.println("Dog barks");}}

class cat extends Animal{
  @override
  void sound(){
    System.out.println("cat meows");}}

Public class Main{
  public static void main(String[] args){
  Animal myAnimal=new Animal();
  Animal mydog=new dog();
  Animal mycat=new cat();

  myAnimal.sound();
  myDog.sound();
  mycat.sound();
}
}
```

Trade offs:

1. Flexibility
2. Readability
3. Performance.

23034

3.
```
import java.util.scanner;
public class Factorion{
    public static int factorial(int n){
    int result=1;
    for(int i=1;i<=n;i++){
    result*=i; }
  return result;
}
public static boolean isfactorion(int n){
int k=n;
int sum=0;
while(k>0){
 int digit = k%10;
  sum+=factorial(digit);
  k/=10;
 }
 return sum==n; }
public static void main(String[] args){
Scanner scan= new Scanner(system.in);
 int lb= scan.nextInt();
    int upd= scan.nextInt();
System.out.println("Factorion in range:");
```

```
for(int i=lb; i<=up; i++){
  if(isFactorion(i)){
    System.out.println(i);
  }
}
scan.close();
}
```

2303 ✓

9. Differences among class, local, instance variables:

Class variable	Instance	Local
1. Defined within class, shared by all instances.	1. Defined inside methods.	1. Defined inside a method or function.
2. Declared with the class name or directly in the class body.	2. Each object of the class has its own body.	2. Only accessible within that method.
3. All objects of the class share the same value for class variable.	3. Different objects can have different values.	3. Its lifetime is limited to the execution of that method.

5. (2304)

```
publie class SumAnnayd
    publie static int calculatesum (int[] arrn){
        int sum = 0;
        for(int num: arrn){
            sum+= num;}
        return sum;
    }
publie static void main (String[] args){
    int[] numbers = {1,2, 3, 4, 5};
    int result = calculatesum (numbers);
    System.out.println (result);
    }
}
```

6. In java, access modifiers are keywords used to define the scope on visibility of variables, methods and classes.

Accessibility of modifiers:

a3033

	Public	Private	Protected
Accessibility	Any class	Within declared class	same package & subclass

7. import java.util.Scanner;
public class QuadraticRoots{
public static void main(String[] args){
Scanner scan = new Scan(System.in);
System.out.println("Enter a, b, c");
int a = scan.nextInt();
int b = scan.nextInt();
int c = scan.nextInt();
int discriminant = b*b - 4*a*c;
if(discriminant >=0){
double sq = Math.sqrt(discriminant);
double root1 = (-b + sq)/(2*a);
double root = (-b - sq)/(2*a);
if (root1 >0 & root2>0){
double smproot = Math.min(root1, root2);
System.out.println("Smallest positive root:"
+ smproot);

2,3034

```
else if(root1>0){
    system.out.println("Smallest positive root:"+
                        sm root1);}
else if (root2>0){
    System.out.println("Smallest positive
                        root:" + root2);}
else { System.out.println("No real roots");
seamen.close();
}
```

8. import java.util.Seamen;
public class Chartype{
public Static void check(char c){
if (character.isLetter(c)){
System.out.println(c+"is letter");}
else if (character.isDigit(c)){
System.out.println(c+"is a digit");}
else if (character.isWhitespace(c)){
System.out.println(c+"Whitespace");}
else { System.out.println(c+"special char");
}}

```
public static void main(String[] args){
  Scanner scanner=new Scanner(System.in);
    string input = scanner.nextLine();
  for(int i=0; i<input.length(); i++){
    char c=input.charAt(i);
      check(c); }
  }
```

Passing array:

```
public class Array{
  public static void prarray(int[] arr){
  for(int i=0; i<arr.length(); i++){
    System.out.println(arr[i]);
public static void main(String[] args){
  int[] arr={1,2,3,4,5};
   prarray(arr);
}
```

23039

9. Method Overriding: Method Overriding is a feature in Java that allows a subclass to provide a new implementation for a method that is already define in its ~~super~~ class. How it works in inheritance-

i) When a subclass overrides a method, the subclass version of the method gets executed, even if the method called on a superclass.

ii) This process is called runtime polymorphism, because the method call is resolved at run time.

(*) What happens when a subclass overrides a method-

1. Subclass method executes, replacing the superclass method.
2. Runtime polymorphism determines method execution at runtime.
3. Superclass method is hidden unless called using super.

Potential issue when overriding methods:

i) Visibility restriction

ii) Exception Limitation

iii) Final static methods

(2303)

* Issue with constructors:

i) Constructors cannot be overridden because they are not inherited.

ii) super() must be used for superclass initialization.

10. Static members:-

1. Belongs to the class.
2. Can be accessed without creating object.
3. Shared by all objects of the class

Example:

```
class Example{
  static int var=10;
  static void statmethod(){
    System.out.println("static");}}
```

```
public class Test{
    public static void main(String[] args){
    System.out.println(Example.var);
    Example.statmethod();}}
```

Non static:

1. Belongs to objects
2. Must be accessed through objects.
3. Cannot be accessed directly.

Example:

```
class Example{
    int nonstatvar=20;
    void nonstatMethod(){
      System.out.println("This is non static");
    }}

public class Test{
    public static void main(String[] args){
    Example obj=new Example();
    System.out.println(obj.nonstatvar);
     obj.nonstatic Method();}}
```

Pallindrome: (23034)

```
import java.util.Scanner;
public class palchk{
    static boolean pal(String s){
    return s.equalsIgnoreCase(new StringBuilder(s).reverse().toString());}
  static boolean pal(int n){
  int rev=0, temp=n;
while(temp>0){
    rev=rev*10+temp%10;
  temp/=10;}
    return n==rev;}
 public static void main(String[] args){
 Scanner sc = new Scanner(System.in);
 System.out.print("Enter number:");
 int n= sc.nextInt();
 System.out.println(n+pal(n)?"is":"is not"+
                    a pallindrome.");
 System.out.println("Enter a string:");
 String s= sc.next();
System.out.println(s+(pal(s)?"is":"is not")
   +"a pallindrome.");
}
```

12. Base class

```
public class Baseclass{
    public void printresult(String Result){
        System.out.println(result);}
    }
Public class sumclass extends Baseclass{
    public double computesum(){
    double sum=0.0;
    for(double i=1.0; i>=0.1; i-=0.1){
        sumt=i;}
    return sum;
    }
public void displaysum(){
    double sum=computeSum();
    printresult("sum of the series:"+sum);
}
}
public class DivisorMultiClass extends Baseclass{
    public int computeGCD(int a, int b){
        while(b!=0){
            int temp=b;
            b=a%b;
            a=temp;}
    return a;
    }
```

```
public int computeGLCM (int a, int b) {
    return (a*b) / computeGCD(a,b); }
public void display GCD and LCM (int a, int b) {
    int gcd = computeGCD(a,b);
    int lcm = computeLCM(a,b);
    printResult("GCD=" + gcd);
    printResult("LCM=" + LCM); }
}
public class Number con extends Baseclass {
    public string dectoBin (int num) {
        return Integer. toBinaryString(num);
    }
public string dectoHex (int num) {
    return Integer.toHexString (num).toUpperCase();
}
public string decto Octal (int num) {
    return Integer. to Octal string (num); }
public int binaryto dec (string binary) {
    return Integer. parseInt (binary, 2);
}
```

```
Public void display
public class void displaycon(int num, String binary){
    print result ("dec to bin : "+ dectoBin (num));
    print result (" dec to Hex: "+ dectoHex(num));
    print result ("Dec to octal: " + dec to Octal(num));
    print result (" bin to dec:" + binary to dec(binary);
}}
public class Custom print extends Baseclass {
    public void pn(String message){
        System.out.println (message);}
    public void display Formated message(String message)
    {pn (message); }}

public class Main class{
    pubic static void main (String [] args){
        Sumclass sum class = new sumclass ();
        DivisorMulti divm = new Divisor multi();
        Numberconvert numcon = new Numbercon ();
        Custom print cuspn = new Custom print Class();

    sumclass. display sum();
        divm. display GCD and LCM (12, 18);
```

```
numbercon.displaycon(123,"1111011");
customprint.displayformattedmessage("ABC");
}
}
```

34

13.

```
import.java.util.Date;
class GeometricObject{
    private String color;
    private boolean filled;
    private Date date;
    public GeometricObject(){
        this.color = "white";
        this.filled = false;
        this.dateereated = new Date();
    }
public GeometricObject(String color, boolean filled)
{ this:color = color;
    this.filled = filled;
    this.date = new Date();
    }
```

```
public string getcolor(){
    return color;}
public class setcolor (string color){
    this.color= color;
}
class circle exteds Geometric Obvect{
    private double radius;
    public circle (){
        super();
        this. radius = 1.0;
    }
public circle (double radius){
    super();
    this.radius= radius;
}
public double get radius (){
    return. radius;}
public void set radius (double radius){
    this. radius= radius;
public double getArea(){
```

return Math.PI * radius * radius; } } (34)

```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5, "Blue", true);
        circle.printCircle();
        System.out.println("Area: " + circle.Area());
    }
}
```

14.

```
import java.math.BigInteger;

public class Fac {
    public static BigInteger fac(int num) {
        BigInteger result = BigInteger.ONE;
        for (int i = 1; i <= num; i++) {
            result = result.multiply(BigInteger.valueOf(i));
        }
        return result;
    }
```

```
public static void main (String [] args){
    int num =100;
    BigInteger fact = fac(number);
    System.out.println ("Factorial=" + fact);
}
}
```

3034

15. Use abstract class when:

1. One wants to share common code between related classes.
2. One needs to define default behaviour for some methods.
3. One needs to define instance variables or constructors.

Use interface when:

1. One wants to define a common set of methods across different classes, regardless of their class hierarchies.
2. One wants to take advantage of multiple inheritance.

18.
```
public class Customrand{
    private static int[] inArr={3,7,11,5,2,8,4};
public static int [ ] myRand( int n, int mv){
    int[ ] randnum= new int [n];
    long curt = System.currentTimeMillis();
    for(int i = 0; i<n; i++){
    int arrayElement= influenArr[i%inArr.
                                length];
randnum[i]= int((currentTime* arrayElement
                        ) %mavvalue));
    currentTime++;
    }
  return randnum;
}
public static int myRand (int mv){
    return (myRand (1, mav v)[0];
}
public static void main (String[]args){
  int[) randomnum= myRand (5, 100);
  for (int num: randnumbers){
```

```
System.out.println(num); }
System.out.println(myRand(50));
}}
```

19. Difference between Thread class and Runnable interface:

Aspect	Thread class	Runnable interface
Extends	Extends Thread class, so cannot extend other classes	Implements Runnable interface, so can extend other classes
Task definition	Task is defined by overriding the run() method	defined by implementing the run().
Flexibility	Less flexible,	More flexible
Thread creation	Thread object is directly used to start the thread.	A Runnable object is passed to a thread

20. Exception handling in java is a mechanism that allows a program to handle runtime errors, ensuring smooth execution even unexpected conditions arise.

Basic structure:

1. try: A block of code that may throw an exception.
2. catch: A block that handles exceptions of specific type.
3. Finally: A block that runs regardless of whether an exception is thrown or not.
4. throw: Used to explicitly throw an exception.
5. throws: Used in method declaration to indicate that a method can throw certain exceptions.

Example:

```
public class Example{
  public static void main(String[] args){
  try{ int result = 10/0;
  }catch(ArithmeticException e){
  System.out.println("Error: "+e.getMessage());
  }finally{
    System.out.println("Executed");
  }
  }
}
```

22. Comparison between Hashmap, Treemap, Linked Hashmap.

Feature	Hashmap	Treemap	Linked Hashmap
Internal Data Structure	Hash table	Red Black Tree	Hash table with doubly linked list
Time complexity (Insert)	O(1)	O(log n)	O(1)

lookup time	O(1)	O(log n)	O(1)
Deletion	O(1)	O(log n)	O(1)
Ordering of Element	No guarantee order	Sorted by key	Maintains insertion order
When to use	When you need fast access don't care about order	When you need sorted keys	When you need insertion order

Hashmap: It is suitable for high performance seonarios.

Treemap: It is ideal when sorted keys are needed.

Linked hash map: It is best when preservation of insertion order is needed.

23.
```
class Animal{
 void sound(){
 System.out.println("Animal sound");
 }
 static void static sound(){
   System.out.println("static Animal Sound");
 }
}
class Dog extends Animal{
  @override
 void sound(){
  System.out.println("Bark");
 }
 static void static sound(){
  System.out.println("static Bark");
 }
}
public ~~mai~~ class Main{
public static void main(String[] args){
```

```
Animal animal = new Dog();
    animal.sound();
    animal.staticSound();
  }
}
```

Key points:

- Static binding is used for static, final and private methods, ~~two~~ fields.
- Dynamic Binding is used for overridden methods in inheritance hierarchies and enables polymorphism, with method resolution occurring at runtime.

29. Differences between submit() and execute() in ExecutorService:

1. execute() method:
   1. signature: void execute(Runnable command)
   2. usage: Used for fire-and-forget tasks
   3. Return value: It doesn't return anything. You can't track the result.
   4. When to use: When you don't care about the result.

2. Submit() method:
   1. signature: <T> Future<T> submit(Callable<T> task)
   2. Usage: Used for submitting tasks that may return a result.
   3. Return value: Returns a future object, which can be used to get the result

4. When to use: When you need to get a result or handle exception from a task.

(59)

25.

```
class NegativeRadius extends Exception{
    public NegativeRadius(String message){
        super(message);
    }
}
class circle{
    private double radius;
public void setRadius(double radius) throw
    NatatieRadiusException{
        if(radius<0){
    throw new NegatieRadiusException("
            cannot be negative radius");
    }
    this.radius=radius;
}
```

2/3034

```
public double getArea(){
    return Math.PI * radius * radius;
}
}

public class CircleAreaCalculator{
    public static void main (String[] args){
        circle circle= new circle();
    try{ cirPle.setRadius(-5);
    }catch(Nagative Radius e){
        System.out.println("Error: "+e.getMessage());
    }
    try{ circle.setRadius(7);
        System.out.println("Area:" + circle.
                                getArea());
    }catch( Negative Radius e){
        System.out.println("Error:" + e.getMessage
                                ());
    }
}
}
```

23039

27.
```
interface Edible{
    String howtoEat();
}
abstract class Animal{
    abstract String sound();
}
class Tiger extends Animal{
    @overnide
    public String sound(){
    return "Tiger roars";
    }
}
class Chicken extends Animal Implements
        Edible{
    @overnide
public String sound(){
    return "Chicken clucks";
}
@overnide
public howtoEat(){
    return "fry it or make curry!!";
```

}} (23034)

```
abstract class fruit implements Edible{ }

class orange extends fruit{
    @overnide
    public string howto Eat(){
        return "Peal and eat";
    }
}

public calass Interfacemain{
    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.onange = new Onange();
    System.out.println (Orange, howto eat()
}}
```

230324

30. import java.util.Scanner;

Pubiic class Div Rem_30{

public static void main (String[] args){

Scanner san = new Scanner (System.in);

System.out.print ("size of array]:");

int n = scan.nextInt();

~~if~~

int [] array 1 = new int[n];

System.out.println ("Enter array]");

for(i=0; i<n; i++){

array[i] = Scanner.nextInt();

}

int m = (int) Math.ceil((double)n/0);

int[] array2 = new int [m]

for(i=0; i<m; i++){

array2 [i] = scan.nextInt();

}

FT23034

```
for(i=0; i< array1.length ; i++){
    for(j=0 ; j<array2.length ; j++){
     int quo = array1[i] / array2[j];
     int rem = array1[i] % array2[j];
    System.out.println("Element" + array1[i] +
      " divided by " + array2[j] + " gives;
        Quotient (coin) = " + quo + ", remainder
    =   " + remainder);
    }
  }
}
```