1. **What is java?**

Java is a high-level, object-oriented programming language originally developed by Sun Microsystems (now owned by Oracle Corporation) in the mid-1990s. It was designed to be platform-independent, meaning that Java programs can run on any device that has a Java Virtual Machine (JVM) installed, regardless of the underlying hardware and operating system.

Java has become one of the most popular programming languages in the world and is widely used for developing a variety of applications, including:

1. \*\*Web Applications\*\*: Java is commonly used for developing server-side web applications. Frameworks like Spring and Java EE provide powerful tools for building scalable and robust web applications.

2. \*\*Mobile Applications\*\*: Android, one of the most widely used mobile operating systems, is built using Java. Developers use Java to create Android applications that run on millions of devices worldwide.

3. \*\*Desktop Applications\*\*: Java can be used to develop desktop applications with graphical user interfaces (GUIs). Libraries such as Swing and JavaFX provide tools for creating cross-platform desktop applications.

4. \*\*Enterprise Software\*\*: Java is widely used in the enterprise space for building large-scale, mission-critical systems. Its robustness, scalability, and platform independence make it a popular choice for enterprise-level applications.

5. \*\*Embedded Systems\*\*: Java can be used in embedded systems development, providing a platform-independent environment for running applications on embedded devices.

Java's syntax is similar to that of C++, making it easy for programmers to learn if they are already familiar with C++ or other C-style languages. Java applications are typically compiled to bytecode, which is a platform-independent intermediate representation of the program. This bytecode can then be executed by any JVM, making Java programs highly portable.

Java also includes a large standard library that provides a wide range of functionality, including support for networking, database access, and concurrency. Additionally, Java has a strong community of developers and a large ecosystem of third-party libraries and frameworks, making it a versatile and powerful programming language.

1. **What is JDK?**

The JDK (Java Development Kit) is a software development kit used by Java developers to create Java applications. It includes a set of tools and libraries that allow developers to write, compile, and run Java programs. The JDK is provided by Oracle Corporation and is available for free download from the Oracle website.

The JDK includes the following components:

1. Java Compiler (javac): The Java compiler is used to compile Java source code into bytecode, which is the intermediate representation of a Java program. The bytecode can then be executed by the Java Virtual Machine (JVM).

2. Java Virtual Machine (JVM): The JVM is a virtual machine that runs Java bytecode. It is responsible for executing Java programs and managing memory, threads, and other resources.

3. Java Runtime Environment (JRE): The JRE is a subset of the JDK that includes the JVM and other libraries and resources needed to run Java programs. It does not include the Java compiler or other development tools.

4. Java Development Tools: The JDK includes a set of development tools, such as the Java debugger (jdb), the Java documentation generator (javadoc), and the Java Archive (JAR) tool, which is used to create and manage JAR files.

5. Java Standard Library: The JDK includes a standard library of classes and interfaces that provide common functionality, such as input/output, networking, and data structures. This library is used by Java developers to build their applications.

Overall, the JDK provides everything a Java developer needs to write, compile, and run Java programs. It is available for multiple platforms, including Windows, macOS, and Linux.

1. **Draw the JDK diagram?**
2. **How many data types we have ?**

Two types of the data types

1. Primitive data type
2. Non-primitive data type

1. **Ascending order of numeric data types?**

byte – short- int -long-float-double

1. **What is type casting?**

Converting form one data type to another data type.

int x = 10;

double y = x; // Implicit casting from int to double

double x = 10.5;

int y = (int) x; // Explicit casting from double to int

1. **What is JVM architecture?**

The Java Virtual Machine (JVM) is an abstract computing machine that enables a computer to run Java programs. It is a part of the Java Runtime Environment (JRE) and is responsible for executing Java bytecode. The JVM is platform-independent, meaning that it can run on any device that has a compatible JRE installed.

The JVM architecture consists of several components:

1. \*\*Class Loader Subsystem\*\*: This subsystem is responsible for loading Java classes into memory. It has three components:

- \*\*Bootstrap Class Loader\*\*: Loads core Java classes from the bootstrap classpath. It is implemented in native code.

- \*\*Extension Class Loader\*\*: Loads classes from the extension classpath.

- \*\*Application Class Loader\*\*: Loads classes from the application classpath.

2. \*\*Runtime Data Areas\*\*: The JVM uses several runtime data areas to store program data and manage execution. These include:

- \*\*Method Area\*\*: Stores class metadata, including method bytecode, field names, and method signatures.

- \*\*Heap\*\*: Stores objects created by the program.

- \*\*Java Stack\*\*: Stores method call frames, including local variables and operand stacks.

- \*\*PC Register\*\*: Stores the address of the currently executing instruction.

- \*\*Native Method Stack\*\*: Stores information about native methods called by the program.

3. \*\*Execution Engine\*\*: The execution engine is responsible for executing Java bytecode. It includes:

- \*\*Interpreter\*\*: Interprets bytecode instructions and executes them one by one.

- \*\*Just-In-Time (JIT) Compiler\*\*: Compiles frequently executed bytecode into native machine code for faster execution.

- \*\*Garbage Collector\*\*: Manages memory by reclaiming unused objects.

4. \*\*Native Method Interface (JNI)\*\*: Allows Java code to call native code written in languages like C and C++.

5. \*\*Java Native Interface (JNI)\*\*: Allows native code to call Java code.

6. \*\*Java Native Access (JNA)\*\*: Provides a more user-friendly interface for calling native code from Java.

7. \*\*Java Virtual Machine Tool Interface (JVMTI)\*\*: Provides a set of tools for monitoring and managing the JVM.

Overall, the JVM architecture provides a platform-independent environment for executing Java programs, with features for memory management, class loading, and execution optimization.

**In** the Java Virtual Machine (JVM) architecture, methods play a crucial role in executing Java programs. Here's a brief overview of how methods are used in the JVM:

**1. \*\*Compilation and Loading\*\*: When you compile a Java source file, the Java compiler (`javac`) generates bytecode instructions for each method in the class. These bytecode instructions are platform-independent and are designed to be executed by the JVM.**

**2. \*\*Class Loading\*\*: When you run a Java program, the JVM loads the bytecode of the classes it needs into memory. This process is known as class loading. Each class has a method area in memory where the JVM stores information about the methods and fields of the class.**

**3. \*\*Method Area\*\*: The method area is a part of the JVM's memory where the JVM stores information about classes and methods. This includes the bytecode instructions for each method, as well as other metadata such as method signatures, access modifiers, and exception tables.**

**4. \*\*Execution\*\*: When a method is called in a Java program, the JVM looks up the method in the method area and executes the bytecode instructions for that method. The JVM uses a call stack to keep track of the currently executing methods and their local variables.**

**5. \*\*Stack Frames\*\*: Each method call in a Java program creates a new stack frame on the call stack. The stack frame contains information about the method's local variables, parameters, return address, and other information needed to execute the method. When a method returns, its stack frame is removed from the call stack.**

**6. \*\*Garbage Collection\*\*: The JVM also uses methods to manage memory and perform garbage collection. When an object is no longer needed, the JVM uses a method called `finalize()` to clean up the object and free its memory.**

**Overall, methods are a fundamental part of the JVM architecture and are used for compiling, loading, executing, and managing Java programs.**

1. **How many class loaders we have ?**

we have three types of class loaders

bootstrap, Extention , application

1. **How many . dot extensions will create after executing the one code?**

it is depending upon the java file

1. **Can I take multi classes in a single java file?**

yes, we can take multiple classes in a single java class file

1. **How many token we have ?**

there are five types of tokens

keywords

identifiers

operator

literals

separators

1. **What are access modifiers we have?**

private

default

protected

public

1. **How many access** modifiers **tokens we have?**

In Java, there are four access modifiers, also known as access control modifiers, that control the visibility and accessibility of classes, fields, methods, and constructors. These access modifiers are:

1. \*\*public\*\*: The `public` access modifier allows a class, field, method, or constructor to be accessed from any other class in the same package or from any other package. There is no restriction on access.

2. \*\*protected\*\*: The `protected` access modifier allows a class, field, method, or constructor to be accessed from any other class in the same package or from any subclass (even if the subclass is in a different package). It restricts access to classes that are not subclasses of the class that contains the protected member.

3. \*\*default (no modifier)\*\*: The default access modifier (also known as package-private) allows a class, field, method, or constructor to be accessed only from other classes in the same package. It restricts access to classes that are not in the same package.

4. \*\*private\*\*: The `private` access modifier allows a field, method, or constructor to be accessed only from within the same class. It restricts access to other classes, even those in the same package or subclasses.

These access modifiers provide a way to control the visibility and accessibility of classes, fields, methods, and constructors, allowing for better encapsulation and security in Java programs.

1. **Explain class structure?(state and behavior of PEN class).**
2. **Description of main method.**
3. **What is meant by "Write Once, Run Anywhere" in the context of Java?**
4. **Explain how Java achieves platform independence.**
5. **What role does the Java Virtual Machine (JVM) play in making Java platform-independent?**
6. **How do byte code and the Java compiler contribute to platform independence?**
7. **Explain the process of executing a Java program on different platforms.**
8. **What is the difference between platform independence and platform-specific features in Java?**
9. **What is a literal in Java?**
10. **How are integer literals represented in Java?**
11. **Explain the difference between a decimal literal and an octal literal in Java.**
12. **What is a floating-point literal, and how is it written in Java?**
13. **What is an instance variable in Java, and how does it differ from a local variable?**
14. **Explain the concept of a static variable. How is it different from an instance variable?**
15. **What is a parameter variable in the context of methods in Java? How are parameter variables used?**
16. **Define local variable in Java. How is the scope of a local variable determined, and what are its limitations?**
17. **Discuss the scope of instance variables, static variables, parameter variables, and local variables. How does the scope of each type of variable affect its visibility in different parts of a Java program?**
18. **How can you initialize an instance variable in Java with a user-defined value during object creation?**
19. **Explain how constructors can be utilized to set user-defined values for instance variables in a Java class.**
20. **What is the role of setter methods, and how do they facilitate the assignment of user-defined values to instance variables?**
21. **How can user input be processed in Java, and how might this input be used to initialize or modify instance variables within an object?**
22. **What is the purpose of the this keyword in Java, and how is it used?**
23. **In the context of an instance method, explain how and why the this keyword is employed.**
24. **How is the this keyword used within a constructor? Provide an example to illustrate its usage.**
25. **Discuss the concept of method chaining in Java and explain how the this keyword can be involved in this process.**
26. **Explain the concept of data hiding in Java and its significance in object-oriented programming.**
27. **Define abstraction in the context of Java programming. How does it contribute to the design of software?**
28. **What is an abstract class in Java? How does it differ from a regular (concrete) class?**

* **Abstract class :** A class that does not provide complete implementation (partial implementation) is defined as an abstract class.
* Once a class is declared as an abstract class we can't create an object for that class.
* It is used to achieve partial abstraction that means by using abstract classes we can achieve partial abstraction(0-100%)
* In an abstract class we can write all abstract method or all concrete method or combination of both the method.
* An abstract class may or may not have abstract method but an abstract method must have abstract class.

1. **Explain the concept of abstract methods. How are they declared and implemented in Java?**

**An** abstract method in Java is a method that is declared without an implementation. It is meant to be overridden by subclasses to provide a specific implementation. Here's a real-time example of an abstract method in Java**:**

**Real time example :**lets consider a university system as an example. In this system , we have different types of employees such as professors , Staff and Students. Each of these employees has a unique role and responsibility**.**

```java

abstract class Shape {

abstract double area(); // Abstract method

}

class Circle extends Shape {

private double radius;

Circle(double radius) {

this.radius = radius;

}

@Override

double area() {

return Math.PI \* radius \* radius;

}

}

class Rectangle extends Shape {

private double length;

private double width;

Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

double area() {

return length \* width;

}

}

public class Main {

public static void main(String[] args) {

Shape circle = new Circle(5);

Shape rectangle = new Rectangle(4, 6);

System.out.println("Area of Circle: " + circle.area());

System.out.println("Area of Rectangle: " + rectangle.area());

}

}

```

In this example, the `Shape` class has an abstract method `area()` that is meant to be overridden by its subclasses. The `Circle` and `Rectangle` classes extend the `Shape` class and provide their own implementations of the `area()` method. When you create an instance of `Circle` or `Rectangle` and call the `area()` method, the appropriate implementation is executed based on the type of the object.

1. **Compare and contrast abstract classes and interfaces in Java. When would you prefer using one over the other?**
2. **Achieving Abstraction:**
3. **How can you achieve abstraction in Java through the use of abstract classes, interfaces, and abstract methods? Provide examples to illustrate your answer.**
4. **What is encapsulation in Java, and why is it considered a fundamental principle of object-oriented programming?**
5. **Explain the role of access modifiers (e.g., private, public, protected) in achieving encapsulation in Java.**
6. **How do getter and setter methods contribute to encapsulation? Provide an example to illustrate their usage.**
7. **Enumerate the benefits of encapsulation in Java, emphasizing the advantages it brings to code maintainability and security.**
8. **Discuss how encapsulation is applied in real-world scenarios, providing examples from Java libraries or frameworks that demonstrate the use of encapsulation principles.**
9. **What is a constructor in Java? How does it differ from a regular method?**
10. **what are the types of Constructor?**
11. **Explain the concept of a default constructor. When is it automatically provided by the compiler?**
12. **How do you define a parameterized constructor in Java? Provide an example.**
13. **What is constructor overloading? Provide a scenario where constructor overloading would be useful.**
14. **Discuss the role of constructors in initializing objects. How are constructors invoked during object creation?**
15. **Compare and contrast the use of ==, equals(), and hashCode() when comparing objects in Java. Under what circumstances would you override these methods in a class?**
16. **What is the purpose of the equals(Object o) method? How can it be overridden to provide a meaningful comparison between objects of a class?**
17. **Discuss the role of the toString() method in Java. How does it contribute to the readability and debugging of code?**
18. **Explain the significance of the hashCode() method. How is it used in Java, and what considerations should be taken into account when implementing this method?**
19. **In what situations would you commonly use the toString() method to represent an object as a string?**
20. **Describe the difference between reference equality (==) and the equals(Object o) method. When would you override the equals() method to achieve a meaningful comparison of object contents?**
21. **What is the IS-A relation in Java, particularly in the context of inheritance?**
22. **How does inheritance promote code reuse in Java?**
23. **Explain the concept of inheritance and its role in object-oriented programming.**
24. **How is the IS-A relationship established between classes in Java?**
25. **List and briefly explain the different types of inheritance supported in Java.**
26. **What is the difference between single and multiple inheritance?**
27. **How are this() and super() used in Java constructors?**
28. **Explain the significance of calling this() and super() in a constructor.**
29. **Discuss the reasons why Java does not support multiple inheritance.**
30. **Describe the access modifiers in Java and their significance.**
31. **How do access modifiers contribute to encapsulation in Java?**
32. **What is the HAS-A relation in Java, specifically in the context of association between classes?**
33. **Provide an example of a HAS-A relationship in a Java program.**
34. **Define superclass and subclass in the context of inheritance.**
35. **How is the superclass related to the subclass in terms of syntax structure?**
36. **Differentiate between method overriding and method overloading.**

**Method Overload:**

Method overload is a concept in object-oriented programming where multiple methods in a class have the same name but different parameters. This allows you to define multiple methods with the same name, but with different signatures, in the same class. The method that gets called depends on the arguments passed to it.

Real-time example:

Consider a class called `Calculator` that has two methods named `add` - one that takes two integers as arguments and another that takes two floating-point numbers as arguments. These two methods are overloaded because they have the same name but different parameter types.

```java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

}

```

In this example, you can call the `add` method with either two integers or two floating-point numbers, and the appropriate method will be called based on the argument types.

Method Override:

Method override is a concept in object-oriented programming where a subclass provides a specific implementation of a method that is already defined in its superclass. This allows you to provide a different implementation of a method in a subclass, which is useful for achieving polymorphism and code reuse.

Real-time example:

Consider a class called `Shape` that has a method named `area`. Now, let's create a subclass called `Circle` that extends `Shape` and overrides the `area` method to calculate the area of a circle.

```java

public class Shape {

public double area() {

return 0;

}

}

public class Circle extends Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double area() {

return Math.PI \* radius \* radius;

}

}

```

In this example, the `Circle` class overrides the `area` method from the `Shape` class to provide a specific implementation for calculating the area of a circle. When you call the `area` method on a `Circle` object, the overridden method in the `Circle` class will be called instead of the one in the `Shape` class.

1. **Provide examples to illustrate the concepts of method overriding and overloading.**
2. **Explain how polymorphism is achieved through inheritance in Java.**

Polymorphism is a concept in computer science and object-oriented programming (OOP) that refers to the ability of different objects to respond to the same message or method invocation in different ways. This allows for flexibility and extensibility in software design, as it enables the same code to be used with different types of objects.

There are two main types of polymorphism:

1. \*\*Compile-time polymorphism (also known as static polymorphism or early binding)\*\*: This occurs when the method to be invoked is determined at compile time. Examples include function overloading and operator overloading in C++.

2. \*\*Run-time polymorphism (also known as dynamic polymorphism or late binding)\*\*: This occurs when the method to be invoked is determined at runtime. It is typically achieved through inheritance and method overriding. In languages like Java and C#, this is often implemented using interfaces or abstract classes.

Polymorphism is a key principle in OOP and is closely related to other concepts such as inheritance, encapsulation, and abstraction. It allows for code reuse, improved readability, and easier maintenance of software systems.

1. **Discuss the advantages of polymorphism in object-oriented programming.**
2. **what is difference between compile-time polymorphisam and static polymorphism?**
3. **what is difference between runtime polymorphism and dynamic polymorphism?**

Polymorphism is a fundamental concept in object-oriented programming (OOP), including Java. It refers to the ability of different objects to respond to the same message (method call) in different ways. In Java, polymorphism is achieved through method overriding and method overloading.

1. \*\*Method Overriding\*\*: This is the most common form of polymorphism in Java. It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass has the same name, return type, and parameters as the method in the superclass. When an object of the subclass is used, the JVM decides which method to call based on the actual type of the object at runtime. This is also known as runtime polymorphism or dynamic method dispatch.

```java

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal a1 = new Dog();

Animal a2 = new Cat();

a1.sound(); // Output: Dog barks

a2.sound(); // Output: Cat meows

}

}

```

2. \*\*Method Overloading\*\*: This is another form of polymorphism in Java. It occurs when a class has multiple methods with the same name but different parameters. The JVM decides which method to call based on the number and type of arguments passed to the method at compile time. This is also known as compile-time polymorphism or static method dispatch.

```java

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println(calc.add(5, 10)); // Output: 15

System.out.println(calc.add(5.5, 10.5)); // Output: 16.0

}

}

```

In summary, polymorphism in Java allows for flexibility and reusability in code by enabling objects of different classes to be treated as objects of a common superclass, and by allowing methods to be defined with the same name but different implementations. This is a key feature of OOP that promotes code organization, maintainability, and extensibility.

\_

1. **Compare and contrast abstract classes and interfaces in Java.**
2. **When would you choose to use an abstract class over an interface, and vice versa?**
3. **What is covariant?**
4. **Explain the concept of method hiding in Java inheritance.**
5. **When does method hiding occur, and how is it different from method overriding?**
6. **Describe the role of the protected access modifier in inheritance.**
7. **What is Autoboxing in Java?**
8. **Explain the concept of Unboxing in Java.**
9. **How does Autoboxing simplify code in Java?**
10. **What are the primitive data types that are involved in Autoboxing?**
11. **Provide an example of Autoboxing in Java.**
12. **Give an example of Unboxing in Java.**
13. **What is the role of the wrapper classes in Autoboxing and Unboxing?**
14. **Can Autoboxing lead to performance overhead? Explain.**
15. **How does Autoboxing help when working with collections in Java?**
16. **Discuss situations where Autoboxing and Unboxing might lead to NullPointerException.**
17. **What is the purpose of the final keyword in Java?**

In Java, the `final` keyword can be used in different contexts to indicate that a variable, method, or class cannot be changed or overridden. When used with a constructor, it has a specific meaning.

When a constructor is marked as `final`, it means that the constructor cannot be overridden by any subclass. This is useful when you want to ensure that a particular constructor implementation is not changed by subclasses, which can be important for maintaining the integrity of the class's design or ensuring that certain initialization steps are always performed.

Here's an example:

```java

class Base {

final Base() {

// constructor implementation

}

}

class Derived extends Base {

// This will cause a compilation error because the constructor in Base is final

Derived() {

// constructor implementation

}

}

```

In this example, the `Base` class has a `final` constructor, which means that it cannot be overridden by the `Derived` class. If you try to define a constructor in `Derived` with the same signature, you will get a compilation error.

It's important to note that marking a constructor as `final` does not prevent the class from being subclassed. It only prevents the constructor from being overridden.

1. **How does declaring a variable as final affect its value?**
2. **Explain the significance of a final method in Java.**
3. **Can a final variable be reassigned after its initial value is set?**
4. **Describe the role of the final keyword in the context of classes.**
5. **In Java, how does the final keyword relate to constants?**
6. **Can a method be both static and final in Java? Explain.**
7. **What is the difference between a final variable and a final method?**
8. **How does the use of the final keyword contribute to code optimization?**
9. **Explain the behavior of a final class in terms of inheritance.**
10. **What is a nested inner class in Java, and how does it differ from a regular (non-nested) class? Provide an example.**
11. **Explain the concept of a method-local inner class. In what scenarios might you use this type of inner class?**
12. **Define a static nested inner class in Java. How is it different from a non-static nested inner class?**
13. **What is an anonymous inner class, and how is it used in Java? Provide a scenario where anonymous inner classes are particularly useful.**

An anonymous inner class in Java is a class without a name that is defined within a method or a block of code. It is used to create an instance of a class and override its methods at the same time. Anonymous inner classes are particularly useful when you need to create a class that is used only once and does not need to be reused elsewhere in the code.

Here's an example of an anonymous inner class in Java:

```java

public class Main {

public static void main(String[] args) {

// Create an instance of the Runnable interface using an anonymous inner class

Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("Hello, world!");

}

};

// Create a thread using the runnable instance

Thread thread = new Thread(runnable);

// Start the thread

thread.start();

}

}

```

In this example, an anonymous inner class is used to create an instance of the `Runnable` interface and override its `run` method. The `run` method prints "Hello, world!" to the console. The anonymous inner class is then used to create a `Thread` instance, and the `Thread` instance is started.

Anonymous inner classes are particularly useful in scenarios where you need to create a class that is used only once and does not need to be reused elsewhere in the code. For example, you might use an anonymous inner class to create an event handler for a button click in a graphical user interface (GUI) application. The event handler is used only once, and there is no need to create a separate class for it.

1. **Discuss the relationships between All inner classes. How do they interact with their enclosing classes and each other?**
2. **How do you declare an enum in Java, and what is the significance of using enums over other data types?**
3. **Explain the concept of enum constants. How are they defined within an enum, and what type of values can they represent?**
4. **Discuss the role of methods in enums. How can you add custom methods to an enum, and what purpose might they serve?**
5. **Name and briefly explain the main components of the Java Virtual Machine (JVM) architecture.**
6. **What is the role of the class loader in the JVM architecture? How does it contribute to the loading of Java classes?**
7. **Describe the purpose of the execution engine in the JVM. How does it execute Java bytecode?**
8. **Identify and explain the different memory areas in the JVM architecture. How is memory management handled during the execution of a Java program?**
9. What is the byteOrientedStream?
10. what is the CharacterOrientedStream?
11. What is difference between both?
12. What is BufferWriter(); ?
13. What is FileInputStream and FileOutputStream?
14. What is SequenceInputStream?
15. What is BufferedOutputStream and BufferedInputStream?
16. DataOutputStream uses and drawbacks?
17. DataInputStream Uses and Drawbacks?
18. What is the working with Reader and Writer like FileReader, FileWriter, printWriter?

**Programs :**

1. **Find output of this program?**

**public class InfiniteExample { static int methodName( ) {**

**System.out.println("a value is :"+(a=a));**

**a= ++a+(a=++a+--b+c\*(--c+a++)); //2+(3+1+3(2+3)) //2+(3+1+15)// 21**

**return a;**

**}**

**static int sub3values(int a, int b) {**

**return a=--a+(b=--b-(--a+b--)+--a-++b\*(++b+--b));**

**}**

**public static void main(String[] args) { System.out.println(methodName(1, 2, 3));**

**System.out.println(sub3values(10, 5));**

**}**

**}**

1. **Write the code for reversing the string?**
2. **Write the code for count the char’s from given numbers?**
3. **Write the code for count of 1’s and count of 0’s from string?**
4. **Write the code for fibonacc?**
5. **Write the code for some patterns?**
6. **Find Output**

**public class InfiniteExample {**

**char ch=’A’;**

**public static void main(String[] args) { System.out.println(c+1); System.out.println(c++); System.out.println(++c);**

**}**

1. **write code for == and equals() and explain?**
2. **Write the code for constructor overloading ?**
3. **Find the which syntax’s are correct**

# **public static void methodName( ) { }**

**static public void methodName( ) { }**

# **static void public methodName( ) { }**

**void static public methodName( ) { }**

**11.How many Armstrong number 1-1000.ans. 13 1-9,153,370,371,407**

An Armstrong number (also known as a narcissistic number, plenary number, or pluperfect number) is a number that is equal to the sum of its own digits each raised to the power of the number of digits. For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

**12.How many Armstrong number 1-100.ans. 9 1-9**

**122.**

**123.what is the CharacterOrientedStream?**

**124.what is difference between byteOrientedStream and CharacterOrientedStream**

**125.**

# Java Basic Programs

1. [**Java Binary Search (Recursive and Iterative) Program**](https://www.sourcecodeexamples.net/2020/11/java-binary-search-recursive-and.html)
2. [**Java Happy Birthday Program**](https://www.sourcecodeexamples.net/2020/11/java-happy-birthday-program.html)
3. [**Java Program That Calculates the Age of a Person**](https://www.sourcecodeexamples.net/2020/08/java-program-that-calculates-age-of.html)
4. [**Java Logical And Or Xor Examples**](https://www.sourcecodeexamples.net/2020/08/java-logical-and-or-xor-examples.html)
5. [**Java Format Date and Time Examples**](https://www.sourcecodeexamples.net/2020/08/java-format-date-and-time-examples.html)
6. [**Java Convert String to Date and Time**](https://www.sourcecodeexamples.net/2020/08/java-convert-string-to-date-and-time.html)
7. [**Java Convert Roman to Integer Number**](https://www.sourcecodeexamples.net/2020/08/java-convert-roman-to-integer-number.html)
8. [**Java Convert Octal to HexaDecimal**](https://www.sourcecodeexamples.net/2020/08/java-convert-octal-to-hexadecimal.html)
9. [**Java Convert Octal to Decimal**](https://www.sourcecodeexamples.net/2020/08/java-convert-octal-to-decimal.html)
10. [**Java Convert Integer Number to Roman Number**](https://www.sourcecodeexamples.net/2020/08/java-convert-integer-number-to-roman.html)
11. [**Java Convert Hexadecimal to Binary**](https://www.sourcecodeexamples.net/2020/08/java-convert-hexadecimal-to-binary.html)
12. [**Java Convert Decimal to Octal**](https://www.sourcecodeexamples.net/2020/08/java-convert-decimal-to-octal.html)
13. [**Java Convert Decimal to Binary**](https://www.sourcecodeexamples.net/2020/08/java-convert-decimal-to-binary.html)
14. [**Java Convert Binary to Decimal**](https://www.sourcecodeexamples.net/2020/08/java-convert-binary-to-decimal.html)
15. [**Java Convert Binary to an Octal**](https://www.sourcecodeexamples.net/2020/08/java-convert-binary-to-octal.html)
16. [**Java Convert Binary to a Hexadecimal**](https://www.sourcecodeexamples.net/2020/08/java-convert-binary-to-hexadecimal.html)
17. [**Java Convert a Unix Timestamp to Date Time**](https://www.sourcecodeexamples.net/2020/08/java-convert-unix-timestamp-to-date-time.html)
18. [**Java Check If Two Integer Arrays are Equals**](https://www.sourcecodeexamples.net/2020/08/java-check-if-two-integer-arrays-are-equals.html)
19. [**Java program that removes all white spaces from the given string**](https://www.sourcecodeexamples.net/2020/06/java-program-that-removes-all-white-spaces-from-given-string.html)
20. [**Java 8 program that counts the number of vowels and consonants in a given string**](https://www.sourcecodeexamples.net/2020/06/java-8-program-that-counts-number-of-vowels-consonants-in-string.html)
21. [**Java 8 program that reverses words of a given string**](https://www.sourcecodeexamples.net/2020/06/java-8-program-that-reverses-words-of.html)
22. [**Java Program to Count Number of Digits in an Integer**](https://www.sourcecodeexamples.net/2020/05/java-program-to-count-number-of-digits.html)
23. [**Java Program Fibonacci Series Using Recursion**](https://www.sourcecodeexamples.net/2020/05/java-program-fibonacci-series-using.html)
24. [**Java Program to Find Factorial of a Number**](https://www.sourcecodeexamples.net/2020/05/java-program-to-find-factorial-of-number.html)
25. [**Java program to calculate the area of the Triangle**](https://www.sourcecodeexamples.net/2020/05/java-program-to-calculate-area-of.html)
26. [**Java Program to Calculate Area of Square**](https://www.sourcecodeexamples.net/2020/05/java-program-to-calculate-area-of-square.html)
27. [**Java Program to Calculate Area of Rectangle**](https://www.sourcecodeexamples.net/2020/05/java-program-to-calculate-area-of-rectangle.html)
28. [**Java Program to find the Smallest of three numbers using Ternary Operator**](https://www.sourcecodeexamples.net/2020/05/java-program-to-find-smallest-of-three.html)
29. [**Java Program to Find Largest of Three Numbers**](https://www.sourcecodeexamples.net/2020/05/java-program-to-find-largest-of-three-numbers.html)
30. [**Java Program to Find GCD of Two Numbers**](https://www.sourcecodeexamples.net/2020/05/java-program-to-find-gcd-of-two-numbers.html)
31. [**Java Program to Check Armstrong Number**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-armstrong-number.html)
32. [**Java Program to Generate Random Number**](https://www.sourcecodeexamples.net/2020/05/java-program-to-generate-random-number.html)
33. [**Java Program to check if the Number is Positive or Negative**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-if-number-is.html)
34. [**Java program to check prime number**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-prime-number.html)
35. [**Java Program to Calculate Simple Interest**](https://www.sourcecodeexamples.net/2020/05/java-program-to-calculate-simple-interest.html)
36. [**Java Program to Swap Two Numbers Without Using a Temporary Variable**](https://www.sourcecodeexamples.net/2020/05/java-program-to-swap-two-numbers-without-using-a-temp-variable.html)
37. [**Java Program to Swap Two Numbers**](https://www.sourcecodeexamples.net/2020/05/java-program-to-swap-two-numbers.html)
38. [**Java Program to Find ASCII Value of a Character**](https://www.sourcecodeexamples.net/2020/05/java-program-to-find-ascii-value-of-a-character.html)
39. [**Java Program to Check Whether an Alphabet is a Vowel or Consonant**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-whether-alphabet-is-vowel-or-consonant.html)
40. [**Java Program to check Leap Year**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-leap-year.html)
41. [**Java Program to Check Even or Odd Numbers**](https://www.sourcecodeexamples.net/2020/05/java-program-to-check-even-or-odd-number.html)
42. [**Java Program to Add Two Numbers**](https://www.sourcecodeexamples.net/2020/05/java-program-to-add-two-numbers.html)
43. [**Java Program to Swap Two Strings Without Using Third Variable**](https://www.sourcecodeexamples.net/2020/03/java-program-to-swap-two-strings.html)
44. [**Java Program to Swap Two Strings with Third Variable**](https://www.sourcecodeexamples.net/2020/03/java-program-to-swap-two-strings-with-third-variable.html)
45. [**How to Get All Digits from String in Java**](https://www.sourcecodeexamples.net/2019/12/how-to-get-all-digits-from-string-in.html)
46. [**Find Duplicate Number in an Array in Java**](https://www.sourcecodeexamples.net/2019/07/find-duplicate-number-in-array-in-java.html)
47. [**How to Get a Current Working Directory in Java?**](https://www.sourcecodeexamples.net/2019/06/how-to-get-current-working-directory-in.html)
48. [**Check Palindrome String in Java**](https://www.sourcecodeexamples.net/2019/06/check-palindrome-string-in-java.html)
49. [**Java Program to Create Pyramid Of Numbers**](https://www.sourcecodeexamples.net/2019/06/java-program-to-create-pyramid-of-numbers.html)

# Java Programs on Strings

* [**Java Program to Count Number of Duplicate Words in String**](https://www.javaguides.net/2018/08/java-program-to-count-number-of-duplicate-words-in-string.html)
* [**Java Program to Count Number of Words in Given String**](https://www.javaguides.net/2018/08/java-program-to-count-number-of-words-in-given-string.html)
* [**Java Program to Count the Number of Occurrences of Substring in a String**](https://www.javaguides.net/2018/08/java-program-to-count-number-of-occurrences-of-substring-in-string.html)
* [**Java Program to Count the Occurrences of Each Character in String**](https://www.javaguides.net/2018/08/java-program-to-count-occurrences-of-each-character-in-string.html)
* [**Java Program to Merge Two String Arrays**](https://www.javaguides.net/2018/08/java-program-to-merge-two-string-arrays.html)
* [**Java Program to Remove Duplicate Words from String**](https://www.javaguides.net/2018/08/java-program-to-remove-duplicate-words-from-string.html)
* [**Java Program to Reverse a String(5 ways)**](https://www.javaguides.net/2018/08/java-program-to-reverse-string.html)
* [**Java Program to Reverse Each Word of a String**](https://www.javaguides.net/2018/08/java-program-to-reverse-each-word-of-string.html)
* [**Java Program to Swap Two Strings**](https://www.javaguides.net/2018/08/java-program-to-swap-two-strings.html)
* [**How to Check if the String Contains only Digits**](https://www.javaguides.net/2018/08/how-to-check-if-string-contains-only-digits.html)
* [**How to Check if the String Contains only Letters**](https://www.javaguides.net/2018/08/how-to-check-if-string-contains-only-letters.html)
* [**How to Check If the String Contains Only Letters or Digits**](https://www.javaguides.net/2018/08/how-to-check-if-string-contains-only-letters-or-digits.html)
* [**Java Program to Check if Input String is Palindrome**](https://www.javaguides.net/2018/08/java-program-to-check-if-input-string-is-palindrome.html)
* [**Java Program to Find all Permutations of String**](https://www.javaguides.net/2018/08/java-program-to-find-all-permutations.html)
* [**How to Remove or Trim All White Spaces from a String in Java**](https://www.javaguides.net/2019/01/how-to-remove-all-white-spaces-from-string-in-java.html)
* [**How to Remove Leading and Trailing White Space From a String in Java**](https://www.javaguides.net/2019/01/how-to-remove-leading-and-trailing-whitespace-from-string-in-java.html)
* [**Java Program to Count Duplicate Characters in a String**](https://www.javaguides.net/2019/08/java-program-to-count-duplicate-in-a-string.html)
* [**Remove Character from String in Java (Java 8)**](https://www.javaguides.net/2019/08/remove-character-from-string-in-java.html)
* [**Java Program to Count Vowels and Consonants in a String (Java 8)**](https://www.javaguides.net/2019/08/java-program-to-count-vowels-and-consonants-in-a-string-java8.html)
* [**4 Ways to Find First Non-Repeated Character in String in Java**](https://www.javaguides.net/2019/08/4-ways-to-find-first-non-repeated-character-in-string-in-java.html)

# Java Programs on Data Structure

### DS Stack

* [**Stack Data Structure in Java**](https://www.javaguides.net/2018/09/stack-data-structure-in-java.html)
* [**Stack Implementation using Array in Java**](https://www.javaguides.net/2018/09/stack-implementation-using-array-in-java.html)
* [**Dynamic Stack Implementation using Array in Java**](https://www.javaguides.net/2018/09/dynamic-stack-implementation-using-array-in-java.html)
* [**Stack Implementation using Linked List in Java**](https://www.javaguides.net/2018/09/stack-implementation-using-linked-list-in-java.html)
* [**Stack Implementation using Array List**](https://www.javaguides.net/2018/09/stack-implementation-using-array-list.html)
* [**Generic Stack Implementation in Java**](https://www.javaguides.net/2018/09/generic-stack-implementation-in-java.html)
* [**Reverse a String using Stack Data Structure in Java**](https://www.javaguides.net/2018/09/reverse-string-using-stack-data-structure-in-java.html)
* [**How to Reverse a Stack in Java**](https://www.javaguides.net/2018/09/how-to-reverse-stack-in-java.html)

### DS Queue

* [**Queue Data Structure in Java**](https://www.javaguides.net/2018/09/queue-data-structure-in-java.html)
* [**Queue Implementation using Linked List in Java**](https://www.javaguides.net/2018/09/queue-implementation-using-linked-list-in-java.html)
* [**Queue Implementation using Circular Array in Java**](https://www.javaguides.net/2018/09/queue-implementation-using-circular-array-in-java.html)
* [**Dynamic Queue Implementation using Array**](https://www.javaguides.net/2018/09/dynamic-queue-implementation-using-array.html)

### DS Linked List

* [**Linked List Data Structure in Java**](https://www.javaguides.net/2018/09/linked-list-data-structure-in-java.html)
* [**Singly Linked List Implementation in Java**](https://www.javaguides.net/2018/09/singly-linked-list-implementation-in-java.html)

### DS Searching Algorithms

* [**Binary Search Algorithm in Java**](https://www.javaguides.net/2018/09/binary-search-algorithm-in-java.html)
* [**Linear Search Algorithm in Java**](https://www.javaguides.net/2018/09/linear-search-algorithm-in-java.html)
* [**Interpolation Search Algorithm in Java**](https://www.javaguides.net/2018/09/interpolation-search-algorithm-in-java.html)

### DS Sorting Algorithms

* [**Bubble Sort Algorithm in Java**](https://www.javaguides.net/2018/09/bubble-sort-algorithm-in-java.html)
* [**Selection Sort Algorithm in Java**](https://www.javaguides.net/2018/09/selection-sort-algorithm-in-java.html)
* [**Insertion Sort Algorithm in Java**](https://www.javaguides.net/2018/09/insertion-sort-algorithm-in-java.html)
* [**Merge Sort Algorithm in Java**](https://www.javaguides.net/2018/09/merge-sort-algorithm-in-java.html)
* [**Quick Sort Algorithm in Java**](https://www.sourcecodeexamples.net/p/java-programs.html)

# 