1. Compare and Contrast Java With Python

Java and Python, while both versatile, high-level languages, differ significantly in their design philosophies and use cases. Java is a compiled, statically-typed language, known for its performance and suitability for enterprise applications, while Python is an interpreted, dynamically-typed language, favored for its simplicity, readability, and use in data science and scripting.

1. Programming Paradigm:

- **Java:** Primarily object-oriented, emphasizing encapsulation, inheritance, and polymorphism.
- **Python:** Supports multiple paradigms, including object-oriented, functional, and procedural, with a focus on simplicity and readability.

2. Typing:

- **Java:** Statically typed, meaning variable types are checked at compile time, requiring explicit type declarations.
- **Python:** Dynamically typed, allowing type checking to occur at runtime, with no need for explicit type declarations.

3. Compilation/Interpretation:

- **Java:** Compiled language, meaning the code is translated into bytecode by the compiler, which is then executed by the Java Virtual Machine (JVM).
- **Python:** Interpreted language, meaning the code is executed line by line by an interpreter.

4. Performance:

- **Java:** Generally faster than Python due to its compiled nature and optimized execution environment.
- **Python:** Slower than Java, as it requires an interpreter to execute the code.

5. Syntax and Readability:

- **Java:** Known for its more verbose and complex syntax, requiring semicolons and curly braces.
- **Python:** Emphasizes readability with its simple and concise syntax, using indentation to define code blocks.

6. Use Cases:

- **Java:** Commonly used for enterprise applications, Android app development, and large-scale systems where performance and scalability are crucial.
- **Python:** Popular for data science, machine learning, web development, scripting, and rapid prototyping.

7. Libraries and Frameworks:

- **Java:** Offers a rich ecosystem of libraries and frameworks, including Spring for web development and Apache for big data processing.
- **Python:** Boasts powerful libraries for data analysis (Pandas, NumPy), machine learning (Scikit-learn, Tensor Flow), and web development (Django, Flask).

2. Characteristics of clean coding

Clean code in Java emphasizes readability, maintainability, and test ability through practices like meaningful names, clear structure, and adherence to coding standards.

Readability & Clarity:

- **Meaningful Names:** Use descriptive names for variables, methods, and classes to convey their purpose.
- **Clear Structure:** Organize code logically with proper indentation, spacing, and comments to enhance readability.
- **Code Formatting:** Adhere to consistent formatting and indentation conventions.

Maintainability:

- **DRY Principle:**

  Avoid code duplication by extracting common logic into reusable functions or methods.

- **SOLID Principles:**

  Apply SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to create modular and flexible code.

- **Error Handling:**

  Implement robust error handling using try-catch blocks and informative error messages.

- **Comments:**
  Provide concise and relevant comments to explain complex logic or non-obvious decisions.

Testability:

- **Small, Focused Functions:**

  Design functions that perform a single, well-defined task to facilitate unit testing.

- **Testable Code:**
  Structure code in a way that makes it easy to write unit tests and verify its behaviour.

Other Important Aspects:

- **Avoid Unnecessary Complexity:** Keep code simple and avoid over-engineering.
- **Follow Conventions:** Adhere to established coding standards and best practices.
- **Refactoring:** Regularly refactor code to improve its structure and maintainability.
- **Code Reviews:** Participate in code reviews to identify potential issues and improve code quality.

3. Study and present-Type casting in Java and Java Keywords and usage

1. Type casting in Java

Type casting in Java is the process of converting one data type to another data type using the casting operator. When you assign a value from one primitive data type to another type, this is known as type casting. To enable the use of a variable in a specific manner, this method requires explicitly instructing the Java compiler to treat a variable of one data type as a variable of another data type.

Syntax:

```
<datatype> variableName = (<datatype>) value;
```

Types of Type Casting

There are two types of Type Casting in java:

- Widening Type Casting
- Narrow Type Casting

Widening Type Casting

A lower data type is transformed into a higher one by a process known as widening type casting. Implic it type casting and casting down are some names for it. It occurs naturally. Since there is no chance of data loss, it is secure. Widening Type casting occurs when:

- The target type must be larger than the source type.
- Both data types must be compatible with each other.

Syntax:

```
larger_data_type variable_name = smaller_data_type_variable;
```

Narrow Type Casting

The process of downsizing a bigger data type into a smaller one is known as narrowing type casting. Casting up or explicit type casting are other names for it. It doesn‟t just happen by itself. If we don‟t explicitly do that, a compile-time error will occur. Narrowing type casting is unsafe because data loss might happen due to the lower data type‟s sma**l**er range of permitted values. A cast operator assists in the process of explicit casting.

Syntax:

```
smaller_data_type variable_name = (smaller_data_type) larger_data_type_variable;
```

Types of Explicit Casting

Mainly there are two types of Explicit Casting:

- Explicit Up casting
- Explicit Down casting

Explicit Up casting

Up casting is the process of casting a subtype to a super type in the inheritance tree‟s upward direction. When a sub-class object is referenced by a superclass reference variable, an automatic process is triggered without any further effort.

Explicit Down casting

When a subclass type refers to an object of the parent class, the process is referred to as down casting. If it is done manually, the compiler issues a runtime Class Cast Exception error. It can only be done by using the instance of operator. Only the downcast of an object that has already been up cast is possible.

2. Java keywords and their usage

In Java, Keywords are the Reserved words in a programming language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as **variable** names or objects.

Java Keywords List

Java contains a list of keywords or reserved words which are also highlighted with different colours be it an IDE or editor in order to segregate the differences between flexible words and reserved words. They are listed below in the table with the primary action associated with them.

1. abstract: Using the abstract keyword in java we can create abstract classes and methods. Abstract keywords are essential to implement abstraction into a program.

2. assert: Using the assert keyword we can implement assertion in a program. Using it we can check the correctness of any assumptions made in a program. The assert keyword was added in JDK 1.4.

3. boolean: Using the boolean keyword we can declare a boolean variable. A boolean variable is a variable that has two values, true and false.

4. break: The break keyword is a jump statement using which we can break out of a loop or switch statement.

5. byte: Using the byte keyword in java we can declare a variable that can hold a value of 1 byte or 8 bit.

6. char: Using the char keyword we can declare a character variable.

7. class: Using the class keyword we can declare a class in java.

8. continue: Continue is also a jump statement in java. Using it we can terminate the current iteration and continue from the next iteration inside the loop.

9. default: default is the keyword using which we can declare the default statement inside a switch case. It is executed when none of the cases match.

10. do: Using the do keyword we can declare a do-while loop. It is an exit controlled loop, so it doesn‟t have any entry condition.

11. double: Using the double keyword we can declare a double variable. A double variable can hold a 64bit long floating point number.

12. else: Using the else keyword we can write statements that wi**l** be executed when the if block doesn‟t execute successfully.

13. extends: extend keyword is used in inheritance. Using it we can inherit one class into another.

4. Compare and contrast methods and constructors and Public and private access modifiers

I. Compare and contrast methods and constructors

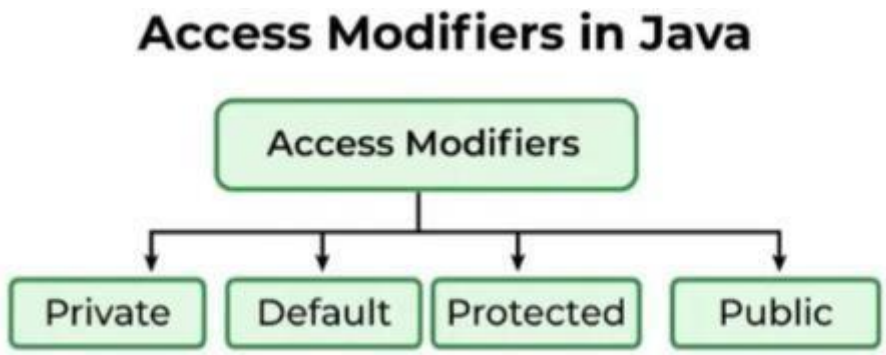
In Java, both constructors and methods are blocks of code within a class, but they serve distinct purposes: constructors initialize objects, while methods perform actions on objects. Constructors have the same name as the class and no return type, whereas methods have arbitrary names and can have any return type.

Similarities:

- **Both are blocks of code within a class:** They both contain instructions that define what the class can do.
- **Can have parameters:** Both constructors and methods can take arguments to customize their behavior.
- **Can have access modifiers:** Both constructors and methods can be declared with access modifiers (e.g., public, private, protected) to control their visibility.

II. Public and private access modifiers

In programming, public and protected are access modifiers that control the visibility and accessibility of classes, methods, and variables. public grants access from anywhere, while protected restricts access to the class itself, its subclasses, and classes within the same package.



❖ **public:**

- A member declared as public can be accessed from any other class, regardless of its package or inheritance relationship.
- It provides the broadest level of access, allowing any code to interact with the member.

❖ **protected:**

- A member declared as protected can be accessed by:
  - ✓ The class itself.
  - ✓ Its subclasses, even if they are in a different package.
  - ✓ Any class within the same package.
- It offers a more restricted access level than public, limiting access to the class and its descendants.

When to Use Each Access Modifier in Real-World Projects

- ✓ **Protected:** This is ideal for methods and fields that should be accessible within the same package and subclasses, commonly used in inheritance-based designs like framework extensions.
- ✓ **Public**: This is used for classes, methods, or fields meant to be accessible from anywhere, such as API endpoints, service classes, or utility methods shared across different parts of an application.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

5. Study and present how byte code works in Java

What is Bytecode in Java?

Byte code in Java is the reason java is platform-independent, as soon as a Java program is compiled bytecode is generated. To be more precise a Java bytecode is the machine code in the form of a .class file.

A bytecode in Java is the instruction set for Java Virtual Machine and acts similar to an assembler.

How does Bytecode Work

When a Java program is executed, the compiler compiles that piece of code and a Bytecode is generated for each method in that program in the form of a .class file.

We can run this bytecode on any other platform as well. But the bytecode is a non-runnable code that requires or relies on an interpreter. This is where JVM plays an important part.

The byte code generated after the compilation is run by the Java virtual machine. Resources required for the execution are made available by the Java virtual machine for smooth execution which calls the processor to allocate the resources.

Bytecode vs Machine code

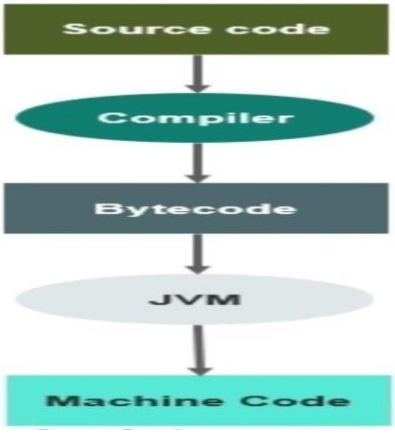
The main difference between the machine code and the bytecode is that the machine code is a set of instructions in machine language or binary which can be directly executed by the CPU.

While the bytecode is a non-runnable code generated by compiling a source code that relies on an interpreter to get executed.

Advantages of Bytecode

Following are a few advantages of Byte code:

- ✓ It helps in achieving platform-independence which is one of the reasons why James Gosling started the formation of Java.
- ✓ The set of instructions for a JVM may differ from system to system but can all interpret Bytecode.
- ✓ Byte codes are non-runnable codes that rely on the availability of an interpreter, this is where JVM comes into play.



- ✓ It is a machine-level language code that runs on the JVM.
- ✓ It adds portability to Java which resonates with the saying, "write once, read anywhere".

6. Present nesting of conditional and iterative statements considering a use case

In Java, nesting conditional (if, else if, else) and iterative (for, while) statements allows for complex logic execution by checking conditions and repeating actions based on those conditions, as demonstrated in the example below.

Use Case: Checking Student Grades

Let's say you want to process a list of student grades and determine which students passed or failed, and then print a message accordingly. Here's how you could use nested conditional and iterative statements:

Java

```
public class StudentGrades {

   public static void main(String[] args) {
      int[] grades = {75, 88, 55, 92, 60, 80}; // Sample student grades

      for (int i = 0; i < grades.length; i++) { // Iterate through each grade
         int grade = grades[i];

         if (grade >= 60) { // Outer conditional: Check if the student passed
            System.out.println("Student " + (i + 1) + " passed with a grade of " + grade);

            if (grade >= 80) { // Inner conditional: Check for higher grades
               System.out.println(" (Excellent performance!)");
            }
         } else { // Else block of outer conditional: Student failed
            System.out.println("Student " + (i + 1) + " failed with a grade of " + grade);
         }
      }
   }
}
```

Explanation:

- **Outer Iteration (for loop):** The for iterates through each grade in the grades array.
- **Outer Conditional (if statement):** The if statement checks if the current grade is greater than or equal to 60 (the passing grade).
- **Inner Conditional (if statement):** If the outer condition is true (student passed), the inner if statement checks if the grade is greater than or equal to 80 (for excellent performance).
- **Output:** Based on the conditions, appropriate messages are printed to the console, indicating whether the student passed or failed and whether their performance was excellent.
- **Else Block:** If the outer condition is false (student failed), the else block executes, printing a message indicating the student failed.

7. Identify Advantage and Disadvantage of Encapsulation and Polymorphism

i. Encapsulation

Encapsulation in Java, a core OOP principle, bundles data and methods that operate on that data within a class, offering advantages like enhanced security, flexibility, and maintainability, but also potentially increasing code complexity.

Advantages of Encapsulation:

- **Data Security and Protection:**

  By restricting direct access to data, encapsulation prevents unauthorized modification and ensures data integrity.

- **Improved Flexibility and Maintainability:**

  Internal changes to the implementation can be made without affecting other parts of the application, promoting easier maintenance and updates.

- **Better Control Over Data:**

  Getters and setters allow controlled access to data, enabling custom validations and ensuring data consistency.

- **Reusability:**

  Encapsulated data and methods can be reused across different parts of a program or in different programs.

- **Modularity:**

  Encapsulation promotes modularity, making code easier to understand, debug, and maintain.

- **Reduced Complexity:**
  By hiding implementation details, encapsulation simplifies the interface, making it easier for developers to understand and work with the code.

Disadvantages of Encapsulation:

- **Increased Code Complexity:**

  While encapsulation improves code organization, it can also lead to more code (getters, setters, validations) for basic data access, potentially increasing complexity.

- **Potential Performance Issues:**

  The increased invocation of getters and setters can have a small impact on performance, especially if not implemented efficiently.

- **Breaking Abstraction:**
  Exposing implementation details through the public interface can break encapsulation and lead to tight coupling and reduced maintainability.

ii. Polymorphism

Polymorphism in Java, while offering benefits like code reusability and flexibility, also presents potential downsides like increased complexity and runtime overhead. Here's a breakdown:

Advantages of Polymorphism:

- **Code Reusability:**

  Polymorphism allows you to write code that can be used with different types of objects, reducing redundancy and promoting efficient development.

- **Flexibility and Dynamism:**

  It enables objects to behave differently based on their specific type, making code adaptable to various situations.

- **Reduced Complexity:**

  Polymorphism can simplify code by allowing the same method name to be used for related functionality across different classes.

- **Improved Readability and Maintainability:**

  By using a common interface or superclass, polymorphism makes code easier to understand and maintain.

- **Extensibility:**

  Polymorphism makes it easier to add new functionalities to existing code by creating subclasses that inherit and override methods.

- **Simplified Coding:**
  Polymorphism simplifies coding by reducing the number of methods and constructors that need to be written.

Disadvantages of Polymorphism:

- **Increased Complexity:**

  Overuse of polymorphism can lead to complex code that is difficult to debug and maintain.

- **Runtime Overhead:**

  Run-time polymorphism (method overriding) can introduce a performance overhead as the method call is resolved at runtime.

- **Potential for Errors:**

  If not implemented carefully, polymorphism can lead to unexpected behaviour or errors.

- **Reduced Readability:**

  While polymorphism can improve readability, it can also make it harder to understand the code if not used correctly.

- **Tight Coupling:**
  Over-reliance on polymorphism can lead to tight coupling between classes, making them difficult to modify or reuse independently.

8. Study and report arrays concept in Java

Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable. They are useful for storing and managing collections of data. Arrays in Java are objects, which makes them work differently from arrays in C/C++ in terms of memory management.

For **primitive arrays**, **elements** are stored in a contiguous memory location. For **non-primitive arrays**, **references** are stored at contiguous locations, but the actual objects may be at different locations in memory.

Example:

```
public class Main {
   public static void main(String[] args)
   {

      // initializing array
      int[] arr = { 1, 2, 3, 4, 5 };

      // size of array
      int n = arr.length;

      // traversing array
      for (int i = 0; i < n; i++)
         System.out.println(arr[i] + " ");
   }
}
```

Basics of Arrays in Java

There are some basic operations we can start with as mentioned below:

1) **Array Declaration:** To declare an array in Java, use the following syntax:

   type[] array Name;

   - **type**: The data type of the array elements (e.g., int, String).
   - **arrayName**: The name of the array.

2) **Create an Array:** To create an array, you need to allocate memory for it using the new keyword:

   // Creating an array of 5 integers
   int[] numbers = new int[5];

This statement initializes the numbers array to hold 5 integers. The default value for each element is 0.

3) **Access an Element of an Array:** We can access array elements using their index, which starts from 0:

   // Setting the first element of the array
   numbers[0] = 10;
   Accessing the first element
   int firstElement = numbers[0];

The first line sets the value of the first element to 10. The second line retrieves the value of the first element.

4) **Change an Array Element:** To change an element, assign a new value to a specific index:

   // Changing the first element to 20
   numbers[0] = 20;

5) **Array Length:** We can get the length of an array using the length property:

```
// Getting the length of the array
int length = numbers.length;
```

Array Properties

- In Java, all arrays are dynamically allocated.
- Arrays may be stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++, where we find length using size of.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class, depending on the definition of the array. In the case of primitive data types, the actual values might be stored in contiguous memory locations (JVM does not guarantee this behaviour). In the case of class objects, the actual objects are stored in a heap segment.

Types of Array in Java

There are two types of arrays in Java:

1. Single-dimensional array in Java
2. Multi-dimensional array in Java

1. Single-Dimensional Array in Java

- A 1D Array in Java is a linear array that allows the storage of multiple values of the same data type.
- It's a collection of data that stores elements of the same type in a sequentially allocated space in memory.
- Single-dimensional arrays can be utilized to store both simple and complex data types, anything from strings, integers, and Booleans to custom-made classes depending on the user's requirements.
- In memory, a one or single-dimensional array holds the data in a linear list. The index of the memory runs from the array size of 0 to -1.
- Single-Dimensional Arrays are initialized just like regular variables, but instead of assigning a single value the programmers can pass several values separated by commas or define an array with static size.
- 1D Arrays are one of the handy tools contained within Java. They are useful for organizing data quickly and efficiently as well as giving us access to stored data using looping structures.

2. Multi-Dimensional Array in Java

- Multi Dimensional Array in Java is an array of arrays, i.e. it is an array object that has multiple dimensions.
- Multi-dimensional arrays are useful when dealing with a large amount of data since they give the ability to store and access data from a single variable but with multiple levels of hierarchy.
- This multi-dimensional array can be expanded to a certain number of dimensions such as two dimensions, three dimensions, etc.

- Multi-dimensional arrays in Java can hold any type of item, i.e. two-dimensional arrays can hold int values or objects such as strings and objects.
- Multi-dimensional arrays also have several methods available to help search and arrange the data within the array, making them very flexible and efficient when dealing with complex tasks.

9. Identify and document how below principles help in coding- SRP and ISP

a. SRP

To effectively code with the Single Responsibility Principle (SRP), focus on creating classes, modules, or functions that have one, and only one, well-defined responsibility, making them easier to understand, maintain, and test.

```
// User class (responsible for holding user data)
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
}

// User Data Manager class (responsible for saving user data)
class UserDataManager {
    public void saveUser(User user) {
        // Logic to save user data (e.g., to a file, database)
        System.out.println("Saving user data: Name - " + user.getName() + ", Email - " + user.getEmail());
    }
}

// Main class to demonstrate the usage
public class Main {
    public static void main(String[] args) {
        User user = new User("John Doe", "john.doe@example.com");
        UserDataManager userDataManager = new UserDataManager();
        userDataManager.saveUser(user);
    }
}
```

A. Single Responsibility Principle (SRP):

- **How it helps:**
- **Improved Maintainability:** When a class has a single, focused responsibility, it's easier to understand, modify, and debug.

- **Enhanced Reusability:** A class with a clear purpose is more likely to be reusable in other parts of the system or even in different applications.
- **Easier Testing:** It's easier to write focused unit tests for a class with a single responsibility, leading to more reliable code.

B. Modularity:

❖ **How it helps:**

- **Encapsulation:** Modules can encapsulate their own data and logic, hiding implementation details from other parts of the system.
- **Reduced Coupling:** Modularity helps minimize dependencies between different parts of the system, making it easier to change one module without affecting others.
- **Improved Maintainability:** Changes to one module are less likely to have unintended consequences in other parts of the system.

C. Abstraction:

❖ **How it helps:**

- **Simplified Interfaces:** Abstraction allows developers to interact with modules at a higher level, without needing to understand the underlying implementation.
- **Improved Reusability:** Abstracted modules can be reused in different contexts, as their interfaces are independent of their implementation.
- **Reduced Complexity:** Abstraction simplifies the overall system architecture, making it easier to understand and maintain.

D. Cohesion:

❖ **How it helps:**

- **High Cohesion:** A module with high cohesion has a clear, focused purpose, making it easier to understand and maintain.
- **Low Cohesion:** A module with low cohesion has many unrelated responsibilities, making it difficult to understand and maintain.
- **SRP and Cohesion:** The SRP promotes high cohesion, as each module should have only one, well-defined responsibility.

b. ISP

The Interface Segregation Principle (ISP) in Java promotes modularity and flexibility by breaking down large interfaces into smaller, more specific ones, ensuring clients only depend on methods they need, leading to more maintainable and adaptable code.

Core Concept:

- **Avoid forcing clients to depend on methods they don't use:**

  ISP states that a client should not be forced to depend on an interface that it doesn't use.
- **Split large interfaces into smaller, more specific ones:**

Instead of having one large interface, create multiple interfaces, each with a specific set of methods relevant to a particular client.

Benefits of ISP:

- **Reduced Coupling:** By breaking down interfaces, you reduce the dependencies between classes, making the system more flexible and easier to maintain.
- **Improved Modularity:** Smaller, focused interfaces make the code easier to understand, test, and modify.
- **Enhanced Reusability:** Specific interfaces can be reused by different clients that need the same functionality, without being burdened by unnecessary methods.
- **Better Maintainability:** Changes to one interface are less likely to impact other parts of the system, leading to more stable and maintainable code.

Example:

```
interface Printer {
   void print(String document);
   void fax(String document);
}

interface Print {
   void print(String document);
}

interface Fax {
   void fax(String document);
}
```

Benefits in Java:

- **Clearer Responsibilities:** Smaller interfaces make it easier to understand what each class is responsible for.
- **Easier Testing:** You can test individual interfaces and their implementations in isolation.
- **Flexibility for Future Changes:** Adding or removing functionality is easier because you're not impacting the entire system.

10. Write a note on multiple inheritance in Java

What is Multiple Inheritance?

Multiple Inheritance is the process in which a subclass inherits more than one superclass. Many real-world examples of Multiple Inheritance also exist.

For example, consider a newly born baby, inheriting eyes from mother, nose from father.

Kindly note that **Java does not support** Multiple Inheritance, but we can use **Interfaces** to achieve the same purpose. Now we will be discussing an example to see what happens when we try to implement Multiple Inheritance in Java.

Example of Multiple Inheritance in Java

Definitely, there must be a reason why Java doesn't support Multiple Inheritance. Before we discuss the actual reason behind this, Let's see what happens if we try to implement Multiple Inheritance in Java.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
public class Main{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Output:

```
weeping...
barking...
eating...
```

Why multiple inheritance is not supported in Java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Suppose there are three classes A, B, and C. The C class inherits A and B classes. If A and B classes have the same method and we call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now  which msg() method would be invoked?
 }
}
```

Compile Time Error

While Java doesn't directly support multiple class inheritance to avoid ambiguity and the "diamond problem", you can achieve a similar effect by using interfaces to implement multiple behaviours in a class. Here's a breakdown:

Why no direct multiple inheritance?

Java's single inheritance model (a class can only extend one parent class) prevents the complexities that arise from multiple inheritance, such as the "diamond problem" where a class inherits conflicting methods from multiple parent classes.

Interfaces to the rescue:

Interfaces define a contract (methods without implementations) that a class can choose to implement.

How it works:

A class can implement multiple interfaces, inheriting the methods declared in those interfaces.

The class then provides its own implementations for those methods.

Example:

Java

```
interface Flyable {
   void fly();
}

interface Walkable {
   void walk();
}

class Bird implements Flyable, Walkable {
   @Override
   public void fly() {
      System.out.println("Bird is flying");
   }

   @Override
   public void walk() {
      System.out.println("Bird is walking");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.fly(); // Output: Bird is flying
        bird.walk(); // Output: Bird is walking
    }
}
```

Benefits of using interfaces:

- ✓ **Flexibility:** A class can implement any number of interfaces, allowing it to inherit behaviours from multiple sources.
- ✓ **Reduced Complexity:** Interfaces avoid the complexities of multiple class inheritance.
- ✓ **Loose Coupling:** Interfaces promote loose coupling between classes, making code more maintainable and reusable.

11. Identify and document system exception

Exception handling in Java allows developers to manage runtime errors effectively by using mechanisms like try-catch block**,** finally block**,** throwing Exceptions**,** Custom Exception handling, etc.
An Exception is an unwanted or unexpected event that occurs during the execution of a program (i.e., at runtime) and disrupts the normal flow of the program‟s instructions. It occurs when something unexpected things happen, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.
Exception in Java is an error condition that occurs when something wrong happens during the program execution.

Exception Handling in Java
Exception handling in Java is an effective mechanism for managing runtime errors to ensure the application‟s regular flow is maintained. Some Common examples of exceptions include Class Not Found Exception, IO Exception, SQL Exception, Remote Exception, etc. By handling these exceptions, Java enables developers to create robust and fault-tolerant applications.

Example: The below Java program modifies the previous example to handle an ArithmeticException using try-catch, and finally blocks and keep the program running.

```
// Java program to demonstrates handling
// the exception using try-catch block
import java.io.*;

class Geeks {
    public static void main(String[] args)
    {
        int n = 10;
        int m = 0;

        try {

            // Code that may throw an exception
            int ans = n / m;
            System.out.println("Answer: " + ans);
        }
        catch (ArithmeticException e) {

            // Handling the exception
            System.out.println(
                "Error: Division by zero is not allowed!");
        }
        finally {
            System.out.println(
                "Program continues after handling the exception.");
        }
    }
}
```

Output

```
Error: Division by zero is not allowed!

Program continues after handling the exception.
```

Major Reasons Why an Exception Occurs

Exceptions can occur due several reasons, such as:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Out of bound
- Null reference
- Type mismatch
- Opening an unavailable file
- Database errors
- Arithmetic errors

12. Study and report how JDBC works

Java Debugger (JDB) is a command-line tool used by Java developers to debug Java code efficiently. It is a part of the Java Development Kit (JDK) and allows developers to set breakpoints, examine variables, and step through the code. Jdb can attach itself to a running Java process and support both local and remote debugging. Its user-friendly command-line interface, combined with its powerful debugging capabilities, makes it an essential tool for any Java developer.

Why Do We Need JDB?

Debugging is an essential part of software development, as it helps developers identify and fix errors in their code. Jdb provides developers with a powerful command-line tool to debug Java code efficiently. It allows developers to set breakpoints, examine variables, and step through the code, helping them identify and fix issues in their code quickly. Jdb can also attach itself to a running Java process, making it suitable for both standalone and server-side Java applications. Overall, jdb is a necessary tool for any Java developer to ensure their applications are reliable and robust.

Advantages of JDBC

- ✓ **Powerful debugging tool:** Jdb is a powerful command-line tool that provides developers with a range of debugging features, such as setting breakpoints and examining variables, helping them identify and fix errors in their Java code efficiently.
- ✓ **Platform independence:** As part of the Java Development Kit (JDK), Jdb can run on any platform that supports Java, making it a highly portable tool.
- ✓ **Remote debugging:** Jdb supports remote debugging, allowing developers to debug Java code running on a remote machine.

Disadvantages of JDBC

- ✓ **Command-line interface:** Jdb has a command-line interface, which can be intimidating for novice users and may require some effort to learn.
- ✓ **Limited GUI debugging:** Jdb does not have a graphical user interface (GUI) for debugging, which some developers may find less intuitive compared to other debugging tools with GUIs.
- ✓ **Learning curve:** Jdb‟s extensive range of features may require some time to master, which can be a disadvantage for developers who need to debug code quickly.

JDBC Architecture

The architecture of Jdb consists of the following components:

- ➢ **Java Debug Interface (JDI):** JDI is a set of APIs that provides a standard interface for interacting with the JVM. Jdb uses JDI to communicate with the JVM and perform debugging tasks.
- ➢ **Debugger:** Debugger is the Java program being debugged by Jdb. Jdb attaches itself to the debugger process and interacts with it through JDI.

- **Jdbc client:** Jdbc client is the user interface that allows developers to interact with Jdbc. It provides a command-line interface for entering debugging commands.
- **Java Virtual Machine (JVM):** JVM is the runtime environment in which Java code is executed. Jdb uses the JVM to run the debugger process and perform debugging tasks.

The Jdb architecture allows developers to debug Java code efficiently by providing a standard interface for interacting with the JVM. The JDI APIs abstract the low-level details of the JVM and provide a simple, standardized way to debug Java code. The Jdb client provides a user-friendly interface for developers to enter debugging commands and interact with the debugger process. Overall, the Jdb architecture is a powerful tool for debugging Java code that helps developers identify and fix errors quickly and efficiently.

JDBC Syntax

Suppose you have a Java program called "HelloWorld.java" which prints "Hello, World!" to the console. Here is the code for this program:

```
public class Java {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

13. Study and report a note on Java annotation

Annotations are used to provide supplemental information about a program.

- ✓ Annotations start with „**@**".
- ✓ Annotations do not change the action of a compiled program.
- ✓ Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- ✓ Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
- ✓ Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.

Categories of Annotations

There are broadly 5 categories of annotations as listed:

1) Marker Annotations

2) Single value Annotations

3) Full Annotations

4) Type Annotations

5) Repeating Annotations

- ➢ **Marker Annotations**

The only purpose is to mark a declaration. These annotations contain no members and do not consist of any data. Thus, its presence as an annotation is sufficient. **@Override** is an example of Marker Annotation.

- ➢ **Single value Annotations**

These annotations contain only one member and allow a shorthand form of specifying the value of the member.

- ➢ **Full Annotations**

These annotations consist of multiple data members, names, values, pairs.

- ➢ **Type Annotations**

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with @Target annotation.

- ➢ **Repeating Annotations**

These are the annotations that can be applied to a single item more than once. For an annotation to be repeatable it must be annotated with the @Repeatable annotation, which is defined in the java.lang.annotation package.

Predefined/ Standard Annotations

Four are imported from java.lang.annotation: @Retention, @Documented, @Target, and @Inherited. Three are included in java. Lang: @Deprecated, @Override and @SuppressWarnings

1) @Deprecated

- ➢ It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

- The Javadoc@deprecated tag should be used when an element has been deprecated.

2) **@Override:** It is a marker annotation that can be used only on methods. A method annotated with @Override must override a method from a superclass.

3) **@SuppressWarnings:** It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

4) **@Documented:** It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included in „Javadoc" comments.

5) **@Target:** It is designed to be used only as an annotation to another annotation. @Target takes one argument, which must be constant from the Element Type enumeration.

6) **@Inherited: @**Inherited is a marker annotation that can be used only annotation declaration. It affects only annotations that will be used on class declarations.

7) **User-defined (Custom):** User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc.

Do keep these certain points as rules for custom annotations before implementing user-defined annotations.

1. **AnnotationName** is an interface.
2. The parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
3. Parameters will not have a null value but can have a default value.
4. default value is optional.
5. The return type of method should be either primitive, enum, string, class name, or array of primitive, enum, string, or class name type.