WHAT IS JAVA
==========
Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.
James Gosling is known as the father of Java. Before Java, its name was Oak.
Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform.
Since Java has a runtime environment (JRE) and API, it is called a platform.

Applications :

1.Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

JVM , JDK AND JRE IN JAVA
==========

JVM, JDK, and JRE are important components in the Java programming and runtime environment. Here's what each of them represents:

1. **JVM (Java Virtual Machine)**:
   - JVM stands for Java Virtual Machine. It is a virtualized execution environment that enables Java applications to run on various hardware and operating systems without modification.
   - JVM is responsible for executing Java bytecode, which is the compiled form of Java source code. It interprets or compiles bytecode into native machine code that can run on the host operating system.
   - JVM provides various services, including memory management, garbage collection, and platform-independent execution.

2. **JDK (Java Development Kit)**:

- JDK stands for Java Development Kit. It is a software package that provides the tools and libraries required for Java development, including writing, compiling, and running Java applications.
- JDK includes the Java Compiler (`javac`), the Java Runtime Environment (`JRE`), and a variety of development tools and libraries.
- It is typically used by developers to create Java applications. JDK is platform-specific, meaning there are different versions of the JDK for different operating systems.

3. **JRE (Java Runtime Environment)**:
- JRE stands for Java Runtime Environment. It is a subset of the JDK and is used for running Java applications.
- JRE includes the JVM, class libraries, and other necessary runtime components that are required for executing Java applications.
- Unlike the JDK, the JRE is only needed on systems where Java applications are to be run, not for development.
- It provides the necessary runtime support for Java applications to execute but does not include development tools.

In summary, the JVM is the runtime engine responsible for executing Java bytecode, the JDK is used for Java development and includes the JDK, and the JRE is used for running Java applications and includes the necessary runtime environment.
Typically, developers use the JDK for writing and compiling Java code, while end-users need the JRE to run Java applications on their systems.

JAVA OOPS:
==========

Object-Oriented Programming (OOP) is a programming paradigm that is heavily used in Java. In OOP, software is organized into objects, which are instances of classes, and it is designed to mimic the way objects work and interact in the real world. Java is a class-based and object-oriented language, so understanding OOP principles is crucial when working with Java. Here are the core principles of OOP in Java:

1. **Classes and Objects**:
- In Java, everything is an object, and objects are created from classes.
- A class is a blueprint or template for creating objects. It defines the attributes (data) and methods (behavior) that objects of the class will have.
- An object is an instance of a class.

- Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

2. **Encapsulation**:
   - Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit (a class).
   - Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.
   - Encapsulation in java : We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
   - The Java Bean class is the example of a fully encapsulated class.
   - It helps in data hiding, where the internal details of a class are hidden from the outside and can only be accessed through well-defined interfaces (methods).
   - Access modifiers like `private`, `protected`, and `public` control the visibility and accessibility of class members.
   - It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

3. **Inheritance**:
   - Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
   - It is an important part of OOPs (Object Oriented programming system).
   - The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
   - Inheritance represents the IS-A relationship which is also known as a parent-child relationship.
   - The `extends` keyword is used in Java to create subclasses that inherit from a superclass.
   - The `implements` keyword is used in java to create subclass that inherit from a interface.
   - Uses : Runtime polymorphism can be achieved(Method Overriding) and Code Reusablity.
   - Types i. Single Inheritance 2. Mutlilevel Inheritance 3. Hierarchical Inheritance
   - Note : To reduce the complexity and simplify the language, multiple inheritance is not supported in java. It is possiable with interfaces.

4. **Polymorphism**:
   - Polymorphism means the ability to take on multiple forms. In Java, it allows objects of different classes to be treated as objects of a common superclass.
   - Polymorphism in Java is a concept by which we can perform a single action in different ways
   - Polymorphism is achieved through method overriding and method overloading.
   - Method overriding involves creating a method in a subclass with the same name as a method in the superclass, allowing the subclass to provide a specific implementation.
   - Method overloading involves defining multiple methods with the same name in the same class, differing by the number or type of parameters.
   - There are two types of polymorphism in Java: compile-time polymorphism and runtime

polymorphism. We can perform polymorphism in java by method overloading and method overriding.

5. **Abstraction**:
   - Abstraction is a process of hiding the implementation details and showing only functionality to the user.
   - Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
   - Abstraction lets you focus on what the object does instead of how it does it.
   - Ways to achieve Abstraction
        There are two ways to achieve abstraction in java:
        i. Abstract class (0 to 100%)
        ii.Interface (100%)

Abstract Class:

i. **Definition**:
   - An abstract class is a class that can have both abstract (i.e., method without a body) and concrete (i.e., method with a body) methods.
   - You can define fields (variables) and constructors in an abstract class.

ii. **Usage**:
   - Abstract classes are used to provide a common base for multiple related classes.
   - They can serve as a blueprint for subclasses, allowing code reuse.

iii. **Inheritance**:
   - An abstract class can be extended (i.e., a subclass can inherit from it).
   - A subclass must implement (override) all the abstract methods of its abstract superclass unless the subclass itself is declared as abstract.

iv. **Access Modifiers**:
   - Abstract classes can have access modifiers for their fields and methods.

v. **Multiple Inheritance**:
   - A Java class can extend only one abstract class, so Java supports single inheritance with abstract classes.

vi. **Constructor**:
   - Abstract classes can have constructors, and these constructors can be called by the constructors of their subclasses.

Interface:

i. **Definition**:

- An interface is a completely abstract class, which means it can only contain abstract methods and constants (public static final fields).
- All methods declared in an interface are implicitly public and abstract, and all fields are implicitly public, static, and final.

ii. **Usage**:
- Interfaces are used to define contracts that classes must adhere to.
- They allow multiple unrelated classes to implement the same set of methods, promoting code interoperability.

iii. **Inheritance**:
- A class can implement multiple interfaces (i.e., a single class can adhere to multiple contracts), enabling multiple inheritance of types.

iv. **Access Modifiers**:
- All members (methods and constants) of an interface are implicitly public.

v. **Multiple Inheritance**:
- Java allows multiple inheritance through interfaces because a single class can implement multiple interfaces.

vi. **Constructor**:
- Interfaces do not have constructors. You cannot create an instance of an interface.

METHOD OVERLOADING AND METHOD OVERRIING IN JAVA
==========

Method overloading and method overriding are two important concepts in object-oriented programming, particularly in Java. They both involve defining multiple methods with the same name, but they serve different purposes and have distinct characteristics. Let's explore each concept:

Method Overloading:

Method overloading, also known as compile-time polymorphism or static polymorphism, allows you to define multiple methods in a class with the same name but different parameters. The method signature, including the method name and the number or types of its parameters, must be different for overloaded methods. The compiler determines which method to call based on the number and types of arguments passed during a method invocation.

Key points about method overloading:

1. The return type of the method doesn't play a role in method overloading; it's not sufficient to differentiate overloaded methods.

2. Overloaded methods are invoked at compile time, based on the method's signature and the arguments passed.

3. Method overloading is a way to provide multiple ways to call a method with different argument lists for convenience and flexibility.

Here's an example of method overloading in Java:

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

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

    public String add(String str1, String str2) {
        return str1 + str2;
    }
}
```

In the example above, the `add` method is overloaded with different parameter types.

Method Overriding:

Method overriding, also known as runtime polymorphism or dynamic polymorphism, allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass has the same name, return type, and parameters as the method in the superclass. Method overriding is used to implement the "is-a" relationship and allows you to define behavior specific to the subclass.

Key points about method overriding:

1. The return type, method name, and parameter list must be the same in the overriding method as in the overridden method.

2. The `@Override` annotation is commonly used to indicate that a method is intended to override a method in the superclass. While not strictly required, it helps catch errors at compile time.

3. Overridden methods are invoked at runtime based on the actual object type, allowing polymorphic behavior.

Here's an example of method overriding in Java:

```java
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Polymorphism
        animal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the `makeSound` method in the `Dog` class overrides the method with the same name in the `Animal` class.

In summary, method overloading allows multiple methods with the same name but different parameters within the same class, while method overriding allows a subclass to provide a specific implementation for a method inherited from its superclass.
Method overriding is a key feature of polymorphism in object-oriented programming.

COMPILE-TIME AND RUNTIME POLYMORPHISM

Compile-time polymorphism (also known as static polymorphism) and runtime polymorphism (also known as dynamic polymorphism) are two types of polymorphism in object-oriented

programming languages like Java. They occur when different methods with the same name are invoked but are determined at different stages of the program's lifecycle.

Compile-Time Polymorphism (Static Polymorphism):

1. **Method Overloading**: Compile-time polymorphism is mostly achieved through method overloading. It occurs when there are multiple methods with the same name in a class, but they have different parameter lists (i.e., a different number or types of parameters).

2. **Determined at Compile Time**: The method to be called is determined by the compiler based on the number and types of arguments passed during the method call.

3. **No Need for Inheritance**: Compile-time polymorphism can occur within a single class without the need for inheritance.

4. **Examples**: Method overloading is a common example of compile-time polymorphism. The specific method to be invoked is resolved during compilation.

Here's an example of compile-time polymorphism using method overloading:

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

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

Runtime Polymorphism (Dynamic Polymorphism):

1. **Method Overriding**: Runtime polymorphism is primarily achieved through method overriding. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

2. **Determined at Runtime**: The method to be called is determined at runtime based on the actual object type, allowing for polymorphic behavior.

3. **Involves Inheritance**: Runtime polymorphism typically involves inheritance, where a subclass extends a superclass and overrides one or more of its methods.

4. **Examples**: Method overriding is a common example of runtime polymorphism. The

specific method to be invoked depends on the actual runtime type of the object.

Here's an example of runtime polymorphism using method overriding:

```java
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Polymorphism
        animal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the method to be invoked is determined at runtime based on the actual object type (`Dog`), demonstrating runtime polymorphism.

In summary, compile-time polymorphism is resolved by the compiler based on the method's signature, whereas runtime polymorphism is determined at runtime based on the actual object type, allowing for more dynamic and flexible behavior.

UPCASTING AND DOWNCASTING IN JAVA

Upcasting and downcasting are two important concepts related to polymorphism and inheritance in object-oriented programming languages like Java. They involve the relationships between superclasses (parent classes) and subclasses (child classes). Let's explore each concept:

Upcasting:

Upcasting is the process of casting (converting) a reference from a subclass type to a superclass type. In other words, it involves treating an object of a derived class as an object of its base class. Upcasting is safe and doesn't require an explicit cast operator.

Key points about upcasting:

1. Upcasting is an implicit or automatic type conversion that is done by the compiler. No casting operator is needed.

2. It promotes code reusability and allows you to use a more general reference to an object of a specific subclass.

3. Upcasting allows you to access only the members (fields and methods) of the superclass from the reference.

Here's an example of upcasting in Java:

```java
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("The dog barks.");
    }

    void fetch() {
        System.out.println("The dog fetches a ball.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the `Dog` object is upcast to the `Animal` reference, and you can access the

overridden `makeSound` method.

Downcasting:

Downcasting is the opposite process of upcasting. It involves casting a reference from a superclass type to a subclass type. Downcasting allows you to access specific members of the subclass that are not present in the superclass. However, downcasting requires an explicit cast operator and is not always safe. It can lead to a `ClassCastException` if the actual object type is not compatible with the downcast.

Key points about downcasting:

1. Downcasting requires an explicit type cast operator, and it's subject to runtime checks to ensure type safety.

2. You can access members specific to the subclass after downcasting.

3. If the object is not an instance of the target subclass, a `ClassCastException` may occur.

Here's an example of downcasting in Java:

```java
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal; // Downcasting
            myDog.fetch(); // Calls the fetch method of the Dog class
        }
    }
}
```

In this example, downcasting is performed after checking whether the object is an instance of the `Dog` class. If the check is successful, you can safely downcast and access the `fetch` method.

In summary, upcasting and downcasting are techniques used to manipulate object references in the context of inheritance and polymorphism.
Upcasting allows you to treat a subclass as its superclass, while downcasting allows you to access specific members of the subclass, but it requires type casting and runtime checks to ensure safety.

Java Arrays
==========

Array:

1. Normally, an array is a collection of similar type of elements which has contiguous memory location.
2. Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location.
3. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
4. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

EXCEPTION HANDLING
==========

Exception : Exception is an abnormal condition.In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception-Handling : The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Exception are two types : 1. Checked Exception 2. Unchecked Exception.

1. Checked Exceptions (Compile-Time Exceptions):

- The classes that directly inherit the Throwable class except RuntimeException are known as checked exceptions.
- Checked exceptions are exceptions that are checked at compile time by the Java compiler. This means that if a method can throw a checked exception, the programmer is required to handle it using `try-catch` blocks or declare it using the `throws` clause in the method signature.
- Examples of checked exceptions include:
  - `IOException`: Occurs when there are problems with input and output operations.
  - `FileNotFoundException`: Occurs when an attempt is made to access a file that does not exist.
  - `SQLException`: Occurs when there are database-related errors.
- Checked exceptions are typically related to external factors and resources.

Example of handling a checked exception:

```java
```

```
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try {
            // Code that may throw a checked exception
        } catch (IOException e) {
            // Handle the exception here
        }
    }
}
```

2. Unchecked Exceptions (Runtime Exceptions):

- The classes that inherit the RuntimeException are known as unchecked exceptions.
- Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. They typically represent programming errors, and it's not required to handle them explicitly using `try-catch` blocks.
- Examples of unchecked exceptions include:
  - `NullPointerException`: Occurs when trying to access an object or variable that is `null`.
  - `ArithmeticException`: Occurs when there is an arithmetic error, such as division by zero.
  - `ArrayIndexOutOfBoundsException`: Occurs when trying to access an array element with an index that is out of bounds.
- Unchecked exceptions are usually the result of mistakes in the code, so the focus is on fixing the underlying issue rather than handling the exception.

Example of an unchecked exception:

```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        int result = numbers[4]; // Throws ArrayIndexOutOfBoundsException
    }
}
```

INNER CLASS IN JAVA
==========

In Java, an inner class is a class defined within another class. Inner classes provide a way to logically group classes within another class, and they have several use cases and benefits, such as encapsulation and code organization. There are several types of inner classes in Java:

1. **Non-static Inner Class (Member Inner Class):**
   - A non-static inner class is a class that is defined inside another class without the `static` keyword.
   - It has access to the instance variables and methods of the outer class, including private members.
   - To create an instance of a non-static inner class, you typically need to create an instance of the outer class first.

   ```java
   public class OuterClass {
       private int outerField;

       public class InnerClass {
           public void display() {
               System.out.println("Outer field: " + outerField);
           }
       }
   }
   ```

2. **Static Inner Class (Nested Class):**
   - A static inner class is a class defined inside another class with the `static` keyword.
   - It does not have access to the instance variables and methods of the outer class directly, but it can access static members.
   - You can create an instance of a static inner class without creating an instance of the outer class.

   ```java
   public class OuterClass {
       private static int outerStaticField;

       public static class InnerClass {
           public void display() {
               System.out.println("Outer static field: " + outerStaticField);
           }
       }
   }
   ```

3. **Local Inner Class:**
   - A local inner class is defined inside a method or block (e.g., a code block within a method).

- It has local scope and can access final or effectively final local variables from the enclosing method.

```java
public class OuterClass {
    public void display() {
        final int localVar = 42;

        class LocalInnerClass {
            public void printLocalVar() {
                System.out.println("Local variable: " + localVar);
            }
        }

        LocalInnerClass inner = new LocalInnerClass();
        inner.printLocalVar();
    }
}
```

4. **Anonymous Inner Class:**
   - An anonymous inner class is a class without a name defined inside a method's argument or within an expression.
   - It is often used to implement an interface or extend a class and override its methods.

```java
public class OuterClass {
    public void displayMessage() {
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Anonymous inner class thread is running.");
            }
        }).start();
    }
}
```

Inner classes can be useful for implementing encapsulation, achieving better code organization, and enhancing code readability.
The choice of which type of inner class to use depends on the specific requirements of your program and the level of access needed to the outer class's members.

Functional Interface
==========
A functional interface in Java is an interface that has exactly one abstract method. Functional interfaces are also known as Single Abstract Method (SAM) interfaces. These interfaces are a fundamental concept in Java's support for lambda expressions and the Java Stream API. Functional interfaces are used to represent single behaviors, and they can be implemented as lambda expressions or method references, making code more concise and expressive.

Here are the key characteristics and rules for functional interfaces in Java:

1. **Single Abstract Method (SAM)**:
   - A functional interface should have exactly one abstract (unimplemented) method. This method is referred to as the "functional method" or "SAM method."
   - A functional interface can have additional default or static methods, but there must be only one abstract method.

2. **@FunctionalInterface Annotation**:
   - While it's not strictly required, you can use the `@FunctionalInterface` annotation to explicitly declare an interface as a functional interface. This annotation helps provide clarity and allows the compiler to generate an error if there are multiple abstract methods.

3. **Lambda Expressions**:
   - You can use lambda expressions to create instances of functional interfaces. The lambda expression provides the implementation of the single abstract method defined in the interface.
   - Lambda expressions are a concise way to represent behavior or functionality.

Here's an example of a simple functional interface in Java:

```java
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();
}

public class Main {
    public static void main(String[] args) {
        // Using a lambda expression to implement the functional interface
        MyFunctionalInterface functionalObj = () -> {
            System.out.println("Doing something...");
        };
        functionalObj.doSomething();
    }
}
```

```

In this example, `MyFunctionalInterface` is a functional interface with a single abstract method, `doSomething()`. We use a lambda expression to provide the implementation for the `doSomething()` method.

Functional interfaces are widely used in Java, especially in the context of the Stream API, where they enable concise and expressive code for operations on collections. Java provides several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`, to cover common use cases for lambda expressions and the Stream API.
These functional interfaces simplify the development of functional-style programming in Java.

JDBC IN JAVA
==========

JDBC, which stands for Java Database Connectivity, is a Java-based API (Application Programming Interface) that allows Java applications to interact with relational databases. It provides a standard interface for connecting to databases, executing SQL queries, and retrieving and manipulating data. JDBC is a crucial technology for database-driven Java applications and is part of the Java Standard Library.

Here are some key points about JDBC:

1. **Database Connectivity**: JDBC enables Java applications to establish connections to relational databases like MySQL, Oracle, PostgreSQL, SQL Server, and others. It provides a way for Java code to communicate with databases through drivers.

2. **Driver Types**:
   - JDBC drivers are used to establish a connection between the Java application and the database. There are four types of JDBC drivers:
     - Type-1 (JDBC-ODBC bridge)
     - Type-2 (Native-API driver)
     - Type-3 (Network Protocol driver)
     - Type-4 (Thin driver, also known as a native-protocol driver)
   - The most common and recommended driver type is Type-4 (Thin driver) because it doesn't
```

require a native database client.

3. **Database Operations**: JDBC allows you to perform various database operations, including executing SQL queries, retrieving and updating data, and managing transactions.

4. **Basic Steps**:
   - To use JDBC, you typically follow these basic steps:
     1. Load the JDBC driver (if not using the Type-4 driver, you need to load the appropriate driver class).
     2. Establish a database connection using a connection URL, username, and password.
     3. Create a statement or prepared statement for executing SQL queries.
     4. Execute SQL queries to retrieve, insert, update, or delete data.
     5. Process the results, if any, from the database.
     6. Close the database connection when done.

5. **Exception Handling**: JDBC methods can throw various exceptions, including `SQLException`. Proper exception handling is important to deal with potential errors during database operations.

6. **Batch Processing**: JDBC supports batch processing, which allows you to execute multiple SQL statements in a single batch, reducing the overhead of repeated database communication.

7. **Transaction Management**: You can use JDBC to manage database transactions, including committing or rolling back changes to ensure data consistency.

8. **Connection Pooling**: In real-world applications, connection pooling is often used to efficiently manage database connections and improve performance.

Here's a basic example of how to use JDBC to connect to a database and execute a simple query in Java:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "yourUsername";
        String password = "yourPassword";

        try {
            // Load the JDBC driver
```

```
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
            Connection connection = DriverManager.getConnection(jdbcURL, username, password);

            // Create a statement
            Statement statement = connection.createStatement();

            // Execute a query
            String sqlQuery = "SELECT * FROM employees";
            ResultSet resultSet = statement.executeQuery(sqlQuery);

            // Process the results
            while (resultSet.next()) {
                System.out.println("Name: " + resultSet.getString("name"));
            }

            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JDBC provides a powerful and standardized way for Java applications to interact with databases, making it a fundamental technology for database-driven Java applications and enterprise systems.

MULTITHREADING
==========

Thread : A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other

threads. It uses a shared memory area.

Multithreading : Multithreading in java is process of executing multiple threads simultaneously.
Multiprocessing and multithreading, both are used to achieve multitasking.
However, we use multithreading than multiprocessing because threads use a shared memory area.
They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
Java Multithreading is mostly used in games, animation, etc.

1) Process-based Multitasking (Multiprocessing) :
Each process has an address in memory. In other words, each process allocates a separate memory area.
A process is heavyweight.
Cost of communication between the process is high.
Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading) :
Threads share the same address space.
A thread is lightweight.
Cost of communication between the thread is low.

Java Thread class : Java provides Thread class to achieve thread programming.
Thread class provides constructors and methods to create and perform operations on a thread.
Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)
In Java, a thread always exists in any one of the following states. These states are:
i. New
ii. Active
iii. Blocked / Waiting
iv. Timed Waiting
v. Terminated

Collections in Java:
==========
The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
Java Collection means a single unit of objects.
Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection : is used to represent a single unit with a group of individual objects.

Collections : is used to operate on collection with several utility methods.

Iterable Interface
==========
The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
It contains only one abstract method. i.e.,

Iterator interface
==========
Iterator interface provides the facility of iterating the elements in a forward direction only.
1 public boolean hasNext() It returns true if the iterator has more elements otherwise it returns false.
2 public Object next() It returns the element and moves the cursor pointer to the next element.
3 public void remove() It removes the last elements returned by the iterator. It is less used.

There are various ways to traverse the collection elements:

-> By Iterator interface.
-> By for-each loop.
-> By ListIterator interface.
-> By for loop.
-> By forEach() method.
-> By forEachRemaining() method.

1. List Interface
==========
List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();

i. ArrayList:
==========
1) ArrayList internally uses a dynamic array to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array.
3) An ArrayList class can act as a list only because it implements List only.
4) ArrayList is better for storing and accessing data.
5) The memory location for the elements of an ArrayList is contiguous.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.
7) To be precise, an ArrayList is a resizable array.

--> ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

ii. LinkedList
==========
1) LinkedList internally uses a doubly linked list to store the elements.
2) If any element is removed from the array, all the other elements are shifted in memory. Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) LinkedList is better for manipulating data.
5) The location for the elements of a linked list is not contagious.
6) There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) LinkedList implements the doubly linked list of the list interface.

iii.Vector

==========

1) Vector is like the dynamic array which can grow or shrink its size.
2) Unlike array, we can store n-number of elements in it as there is no size limit.
3) It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.
4) It is recommended to use the Vector class in the thread-safe implementation only.
5) If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.
6) The Iterators returned by the Vector class are fail-fast.
7) In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-
a. Vector is synchronized.
b. Java Vector contains many legacy methods that are not the part of a collections' framework.

iv. Stack
==========
1) The stack is a linear data structure that is used to store the collection of objects.
2) It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects.
3) One of them is the Stack class that provides different operations such as push, pop, search, etc.
4) In this section, we will discuss the Java Stack class, its methods, and implement the stack data structure in a Java program.
5) The stack data structure has the two most important operations that are push and pop.
6) The push operation inserts an element into the stack and pop operation removes an element from the top of the stack.

2. Queue Interface
==========

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.
There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
Queue interface can be instantiated.

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

i. PriorityQueue

==========

1) PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority.

2) It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue.

3) However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

3. Deque Interface

==========

The interface called Deque is present in java.util package.
It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure.
Therefore, a deque can be used as a stack or a queue.
We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue.
As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for "double ended queue".

i. ArrayDeque class

==========

1) We know that it is not possible to create an object of an interface in Java.

2) Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque.

3) It grows and shrinks as per usage. It also inherits the AbstractCollection class.

4) The important points about ArrayDeque class are:

a) Unlike Queue, we can add or remove elements from both sides.

b) Null elements are not allowed in the ArrayDeque.

c) ArrayDeque is not thread safe, in the absence of external synchronization.

d) ArrayDeque has no capacity restrictions.

e) ArrayDeque is faster than LinkedList and Stack.

4. Set Interface

==========

Set Interface in Java is present in java.util package.
It extends the Collection interface.
It represents the unordered set of elements which doesn't allow us to store the duplicate items.
We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

i. HashSet Class

==========

1) Java HashSet class is used to create a collection that uses a hash table for storage.

2) It inherits the AbstractSet class and implements Set interface.

3) The important points about Java HashSet class are:

a) HashSet stores the elements by using a mechanism called hashing.

b) HashSet contains unique elements only.

c) HashSet allows null value.

d) HashSet class is non synchronized.

e) HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.

f) HashSet is the best approach for search operations.

g) The initial default capacity of HashSet is 16, and the load factor is 0.75.

ii. LinkedHashSet Class

==========

1) Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface.

2) It inherits the HashSet class and implements the Set interface.

3) The important points about the Java LinkedHashSet class are:

a) Java LinkedHashSet class contains unique elements only like HashSet.

b) Java LinkedHashSet class provides all optional set operations and permits null elements.

c) Java LinkedHashSet class is non-synchronized.

d) Java LinkedHashSet class maintains insertion order.

iii. TreeSet Class

==========

1) Java TreeSet class implements the Set interface that uses a tree for storage.

2) It inherits AbstractSet class and implements the NavigableSet interface.

3) The objects of the TreeSet class are stored in ascending order.

4) The important points about the Java TreeSet class are:

a) Java TreeSet class contains unique elements only like HashSet.

b) Java TreeSet class access and retrieval times are quiet fast.

c) Java TreeSet class doesn't allow null element.

d) Java TreeSet class is non synchronized.

e) Java TreeSet class maintains ascending order.

f) Java TreeSet class contains unique elements only like HashSet.

g) Java TreeSet class access and retrieval times are quite fast.

h) Java TreeSet class doesn't allow null elements.
i) Java TreeSet class is non-synchronized.
j) Java TreeSet class maintains ascending order.
k) The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

5 Map Interface
==========
A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry.
A Map contains unique keys.
A Map is useful if you have to search, update or delete elements on the basis of a key.
There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Class	Description
HashMap	is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	is the implementation of Map and SortedMap. It maintains ascending order.

i. HashMap
==========
1) Java HashMap class hierarchy
2) Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.
3) If you try to insert the duplicate key, it will replace the element of the corresponding key.
4) It is easy to perform operations using the key index like updation, deletion, etc.
5) HashMap class is found in the java.util package.
6) HashMap in Java is like the legacy Hashtable class, but it is not synchronized.
7) It allows us to store the null elements as well, but there should be only one null key.
8) Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value.
9) It inherits the AbstractMap class and implements the Map interface.
a) Java HashMap contains values based on the key.

b) Java HashMap contains only unique keys.
c) Java HashMap may have one null key and multiple null values.
d) Java HashMap is non synchronized.
e) Java HashMap maintains no order.
f) The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

ii. LinkedHashMap class
==========

1) Java LinkedHashMap class hierarchy
2) Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.
3) Points to remember:
a) Java LinkedHashMap contains values based on the key.
b) Java LinkedHashMap contains unique elements.
c) Java LinkedHashMap may have one null key and multiple null values.
d) Java LinkedHashMap is non synchronized.
e) Java LinkedHashMap maintains insertion order.
f) The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

6) TreeMap class
==========
Java TreeMap class hierarchy: TreeMap class implements SortedMap interface and SortedMap interface extends Map interface.
Java TreeMap class is a red-black tree based implementation.
It provides an efficient means of storing key-value pairs in sorted order.
The important points about Java TreeMap class are:
Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
Java TreeMap contains only unique elements.
Java TreeMap cannot have a null key but can have multiple null values.
Java TreeMap is non synchronized.
Java TreeMap maintains ascending order.