
🔹 Topic 26: Exception Handling in Java

✅ Definition:

- **Exception Handling** in Java is a mechanism that allows a program to detect and respond to **runtime errors** (exceptions) in a controlled and user-friendly way, rather than crashing abruptly.
- An **exception** is an event that disrupts the normal flow of the program during execution.

✅ Use Case:

Scenario	Application
File not found	File handling
Invalid user input	Web form or CLI input validation
Dividing by zero	Arithmetic operations
Null reference access	Object access in real-world applications
Array index out of bounds	Collection and array operations

✅ Real-Time Usage:

- In banking apps: To handle invalid transactions or server issues.
- In online shopping: To catch cart errors, payment failures.
- In file processing: To handle file not found, access denied, I/O errors.

✅ Architecture Diagram:

```
              ┌─────────────────────────────┐
              │       Java Program          │
              └─────────────┬───────────────┘
                            │
                            ▼
                    Exception Occurs
                            │
        ┌───────────────────┼───────────────────┐
        ▼                                       ▼
    Try Block                             No Try Block
        │                                       │
        ▼                                       ▼
  Catch Exception                          JVM handles
exception
        │                                       │
        ▼                                       ▼
Custom Message / Logging             Program Terminated
Abruptly
```

✅ Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType name) {
    // Code to handle the exception
} finally {
    // Code that will execute always
}
```

✅ Keywords:

`try, catch, finally, throw, throws, Exception, Throwable`

✅ Types of Exceptions:

Category	Description	Examples
Checked	Checked at compile-time	IOException, SQLException
Unchecked	Occurs at runtime	NullPointerException, ArithmeticException
Errors	Not meant to be handled by programs	StackOverflowError, OutOfMemoryError

---

✅ Example 1: Basic Try-Catch

```java
public class Example {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        }
    }
}
```

✅ Example 2: Multiple Catch Blocks

```java
try {
    int[] arr = new int[3];
    arr[4] = 10;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic Exception");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array Index Exception");
} catch (Exception e) {
    System.out.println("General Exception");
}
```

✅ Example 3: Finally Block

```java
try {
    int a = 5 / 1;
} catch (Exception e) {
    System.out.println("Error occurred");
} finally {
    System.out.println("Finally block always executes");
}
```

✅ Example 4: Throw Keyword

```
public class Test {
    static void validate(int age) {
        if (age < 18)
            throw new ArithmeticException("Not eligible to 
vote");
        else
            System.out.println("Eligible to vote");
    }

    public static void main(String[] args) {
        validate(16);
    }
}
```

✅ Example 5: Throws Keyword

```
public class ExceptionEx6 {

     public static int divideNum(int a, int b)
              throws ArithmeticException, 
ArrayIndexOutOfBoundsException  {

         int div = a/b;

         System.out.println(div);

         String[] name = {"Mugilan", "Java", "Info"};

         System.out.println(name[3]);
         return div;
     }

     public static void main(String[] args) {
          //ExceptionEx6 ee6 = new ExceptionEx6();

          try {

               divideNum(36, 6);

          }catch(ArithmeticException e) {

               System.out.println("Number cannot divide by
0");
               e.printStackTrace();

          }catch(ArrayIndexOutOfBoundsException e) {

               System.out.println("Array index is out of
bound..");
               e.printStackTrace();

          }
     }
}
```

✅ Real-Time Example:

```
public class Login {
    public static void main(String[] args) {
        try {
            String username = null;
            if (username.equals("admin")) {
                System.out.println("Login successful");
            }
        } catch (NullPointerException e) {
            System.out.println("Username cannot be null.");
        }
        }
}
```

✅ 15 MCQ Questions with Answers:

1. What is an exception?

    A) Compile error

    B) Runtime error

    ✅ **Answer: B**

2. Which block is always executed?

    A) try

    B) catch

    C) finally

    ✅ **Answer: C**

3. Which keyword is used to throw exception manually?

    A) throws

    B) throw

    ✅ **Answer: B**

4. Can we have multiple catch blocks?

    A) No

    B) Yes

    ✅ **Answer: B**

5. What is the superclass of all exceptions?

   A) Object

   B) Throwable

   ✅ **Answer: B**

6. Which is a checked exception?

   A) ArithmeticException

   B) IOException

   ✅ **Answer: B**

7. Which keyword is used to handle exception declaration?

   A) throw

   B) throws

   ✅ **Answer: B**

8. What happens if exception is not caught?

   A) JVM terminates the program

   ✅ **Answer: A**

9. Can finally block be skipped?

   A) Yes

   B) No

   ✅ **Answer: B**

10. What is the purpose of catch block?

    A) Catch errors

B) Handle exceptions

✅ **Answer: B**

11. What happens if exception is caught?

A) Program terminates

B) Control continues normally

✅ **Answer: B**

12. Which block can be skipped if no exception occurs?

A) try

B) catch

✅ **Answer: B**

13. Can we re-throw exceptions in catch?

A) No

B) Yes

✅ **Answer: B**

14. Which exception is thrown when dividing by 0?

A) NumberFormatException

B) ArithmeticException

✅ **Answer: B**

15. Can we have try without catch or finally?

A) Yes

B) No

✅ **Answer: B**

✅ 20 Interview Q&A:

1. **Q:** What is exception handling in Java?

   A: A mechanism to handle runtime errors gracefully.

2. **Q:** What are checked and unchecked exceptions?

   A: Checked: checked at compile time; Unchecked: runtime errors.

3. **Q:** Can we use multiple catch blocks?

   A: Yes, to handle different types of exceptions.

4. **Q:** What is the use of finally block?

   A: Executes regardless of exception occurrence.

5. **Q:** Can a try block exist without catch or finally?

   A: No, must be followed by either catch or finally.

6. **Q:** What is the difference between throw and throws?

   A: `throw` is used to throw an exception; `throws` declares it.

7. **Q:** What is the base class of all exceptions?

   A: `Throwable`.

8. **Q:** Can we catch multiple exceptions in a single catch block (Java 7+)?

   A: Yes, using pipe `|` operator.

9. **Q:** Can finally block override return statement?

   A: Yes, if return exists in both try and finally.

10. **Q:** Can we throw a custom exception?

    A: Yes, by extending `Exception` or `RuntimeException`.

11. **Q:** What is a custom exception?

    A: User-defined exception for specific errors.

12. **Q:** What is the order of execution in try-catch-finally?

    A: Try → Catch (if exception) → Finally

13. **Q:** What happens if finally block has an exception?

    A: It overrides previous exceptions if not handled.

14. **Q:** What is the difference between Error and Exception?

    A: Errors are not meant to be handled (e.g., OutOfMemoryError).

15. **Q:** Can you catch an Error?

    A: Yes, but not recommended.

16. **Q:** What is exception chaining?

    A: Passing one exception as cause to another.

17. **Q:** What is NullPointerException?

    A: Occurs when trying to use a null object.

18. **Q:** Can we have nested try blocks?

    A: Yes.

19. **Q:** Can a catch block exist without try?

    A: No.

20. **Q:** What is the use of `e.printStackTrace()`?

    A: Prints complete stack trace of the exception.

✅ Topic Outcome:

After learning this topic, students will:

- Understand the flow and need of exception handling.
- Write resilient applications with try-catch-finally blocks.
- Create and use custom exceptions.

- Use `throw` and `throws` properly.

✅ Summary:

- Exception handling in Java is critical for building robust, secure, and stable applications.
- It separates error-handling logic from regular code and prevents the program from crashing due to unexpected errors.

🔹 Compile-Time Exceptions vs Run-Time Exceptions

✅ 1. Compile-Time Exceptions (Checked Exceptions)

Definition:

- These exceptions are **checked by the compiler at compile time**.
- The compiler ensures they are either handled using `try-catch` or declared using `throws`.

🔹 Common Examples:

- `IOException`
- `SQLException`
- `FileNotFoundException`
- `ClassNotFoundException`

✅ Example:

```
import java.io.IOException;

public class ThrowsExample {

    // Method that declares it may throw IOException
    static void checkFile() throws IOException {
        throw new IOException("File not found!");
    }

    public static void main(String[] args) {
        try {
            checkFile();  // calling method that throws 
exception
        } catch (IOException e) {
            System.out.println("Exception caught: " + 
e.getMessage());
        }
    }
}
```

If you don't handle or declare the exception, the compiler throws an error.

✅ Characteristics:

Feature	Compile-Time Exceptions
Checked by compiler	✅ Yes
Must handle or declare	✅ Required
Occurs	Before program runs
Causes	External operations (file, DB)

✅ When to Use:

- File operations
- Database connections
- Network communications

✅ 2. Run-Time Exceptions (Unchecked Exceptions)

Definition:

- These are **not checked by the compiler**.
- They occur **at runtime**, and it's up to the developer to handle them (optional).

🔹 Common Examples:

- `ArithmeticException`
- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`
- `IllegalArgumentException`

✅ Example:

```java
public class Example {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        int result = a / b; // ArithmeticException at
runtime
        System.out.println(result);
    }
}
```

Code compiles fine, but throws exception at runtime.

✅ Characteristics:

Feature	Run-Time Exceptions
Checked by compiler	❌ No
Must handle or declare	❌ Optional
Occurs	During execution
Causes	Logical errors, bad input

✅ When to Handle:

- User inputs
- Object manipulation (null checks)
- Arithmetic logic
- Collections and data access

🔄 Comparison Table:

Aspect	Compile-Time Exception	Run-Time Exception
Also Known As	Checked Exception	Unchecked Exception
Compiler Check	✅ Required	❌ Not required
Handling Requirement	Must handle or declare	Optional

Common Examples	`IOException`, `SQLException`	`NullPointerException`, `ArithmeticException`
Typical Cause	External issues (File, DB)	Logic errors, wrong operations
Occurs	Before execution	During execution
Package	`java.io`, `java.sql`	`java.lang`

✅ Summary:

- **Compile-time exceptions (checked)** must be handled or declared.
- **Run-time exceptions (unchecked)** don’t need to be handled, but should be to avoid crashes.
- Good exception handling strategy involves a mix of compile-time safety and runtime validation.

🔹 Topic 27: Collections in Java

✅ Definition:

- The **Collections Framework** in Java is a unified architecture for storing and manipulating groups of data.
- It includes a set of interfaces and classes that handle **dynamic data structures** like **lists, sets, queues, and maps**, allowing efficient storage, retrieval, and manipulation of data.

✅ Use Case:

- To manage groups of related objects.
- Dynamic data storage with automatic resizing.
- Efficient searching, sorting, and filtering.
- Real-time modeling of data using maps, sets, queues, etc.

✅ Real-Time Usage:

Real-Time Scenario	**Collection Usage**
E-commerce Cart	Use `List<Product>` to store cart items
Employee Database	Use `Map<EmpID, Employee>` for fast lookup
Unique Email Storage	Use `Set<String>` to avoid duplicates
Job Scheduling System	Use `Queue` or `PriorityQueue`

✅ Architecture Diagram of Java Collections:

```
                   Collection (Interface)

                               |

          --------------------------------------------

           |                    |                    |

        List                  Set                  Queue

           |                    |                    |

ArrayList,LinkedList HashSet,TreeSet  LinkedList,PriorityQueue
```

```
Map (Interface)
      |
HashMap, TreeMap
```

✅ Syntax Overview:

```
List<String> list = new ArrayList<>();

Set<Integer> set = new HashSet<>();

Map<String, Integer> map = new HashMap<>();

Queue<Double> queue = new PriorityQueue<>();
```

✅ Keywords:

`Collection`, `List`, `Set`, `Map`, `Queue`, `ArrayList`, `HashSet`, `HashMap`, `LinkedList`, `TreeSet`, `PriorityQueue`

✅ Collection vs Collections

Term	Definition	Package
Collection	It is the root interface in the **java.util** package representing a group of objects, known as elements.	`java.util.Collection`
Collections	It is a utility class in **java.util** that consists of static methods to operate on or return collections.	`java.util.Collections`

✅ Core Interfaces in Collections Framework:

Interface	Description	Common Implementations
Collection	Root interface of the hierarchy	-
List	Ordered collection with duplicates	`ArrayList`, `LinkedList`
Set	Unordered collection without duplicates	`HashSet`, `TreeSet`

Queue	FIFO-based collection	`LinkedList`, `PriorityQueue`
Map	Key-value pair collection	`HashMap`, `TreeMap`

✅ Simple Example: List

```java
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Alice"); // Duplicates allowed

        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

✅ Simple Example: Set

```java
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> emails = new HashSet<>();
        emails.add("a@gmail.com");
        emails.add("b@gmail.com");
        emails.add("a@gmail.com"); // Duplicate ignored

        System.out.println(emails);
    }
}
```

✅ Simple Example: Map

```
import java.util.*;

public class MapExample {
    public static void main(String[] args) {
        Map<Integer, String> users = new HashMap<>();
        users.put(101, "Alice");
        users.put(102, "Bob");

        System.out.println(users.get(101)); // Output:
Alice
    }
}
```

✅ Simple Example: Queue

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("Task1");
        queue.add("Task2");
```

```java
        System.out.println(queue.poll()); // Task1
    }
}
```

✅ Real-Time Example: Employee Directory using Map

```java
import java.util.*;

class Employee {
    int id;
    String name;

    Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

```java
public class EmployeeDirectory {
    public static void main(String[] args) {
        Map<Integer, Employee> directory = new HashMap<>();

        directory.put(1, new Employee(1, "Alice"));
        directory.put(2, new Employee(2, "Bob"));

        Employee e = directory.get(2);
        System.out.println("Employee: " + e.name);
    }
}
```

✅ Example: Collection Interface

```java
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

✅ Example: Collections Utility Class

```java
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4, 1, 3, 2);
        Collections.sort(numbers); // Static method from
Collections class
        System.out.println("Sorted list: " + numbers);
    }
}
```

✅ Iterators in Java Collections

➢ Iterator:

- Iterator is an interface used to **iterate through elements** in a collection.

- Available in **java.util** package.

✅ Syntax:

```
Iterator<Type> itr = collection.iterator();

while (itr.hasNext()) {

    Type element = itr.next();

    // Process element

}
```

✅ Example:

```
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        Iterator<String> iterator = names.iterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

➢ ListIterator:

- Can traverse both forward and backward.
- Only available for classes that implement **List** (like ArrayList, LinkedList).

✅ Example:

```
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> colors = new
ArrayList<>(Arrays.asList("Red", "Green", "Blue"));

        ListIterator<String> listIterator =
colors.listIterator();

        System.out.println("Forward Direction:");
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }
        System.out.println("Backward Direction:");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

✅ 15 MCQs with Answers:

1. Which collection allows duplicate values?

   A) Set

   B) List

   ✅ **Answer: B**

2. Which of these maintains insertion order?

   A) HashSet

   B) LinkedHashSet

   ✅ **Answer: B**

3. Which is not part of the Collection interface?

   A) List

   B) Set

   C) Map

   ✅ **Answer: C**

4. Which class implements Map?

    A) ArrayList

    B) HashMap

    ✅ **Answer: B**

5. Which collection does not allow duplicates?

    A) List

    B) Set

    ✅ **Answer: B**

6. Which data structure follows FIFO?

    A) Queue

    ✅ **Answer: A**

7. What is the default capacity of ArrayList?

    A) 5

    B) 10

    ✅ **Answer: B**

8. Which Map maintains order of insertion?

    A) HashMap

    B) LinkedHashMap

    ✅ **Answer: B**

9. Which Map sorts by natural order of keys?

    A) TreeMap

✅ **Answer: A**

10. What method is used to remove element in List?

    A) delete()

    B) remove()

    ✅ **Answer: B**

11. What is the return type of `poll()` in Queue?

    A) Object

    ✅ **Answer: A**

12. Which allows key-value pairs?

    A) Set

    B) Map

    ✅ **Answer: B**

13. Which is a legacy class in Collection?

    A) Vector

    ✅ **Answer: A**

14. Which implementation is thread-safe?

    A) HashMap

    B) Hashtable

    ✅ **Answer: B**

15. Which allows null keys and values?

A) HashMap

✅ **Answer: A**

✅ 20 Interview Q&A:

1. **Q:** What is Java Collections Framework?
   A: A group of interfaces and classes for storing and manipulating groups of data.

2. **Q:** Difference between ArrayList and LinkedList?
   A: ArrayList is faster in search; LinkedList is faster in insertion/deletion.

3. **Q:** Why Map is not a part of Collection interface?
   A: Because it doesn't represent a collection of elements; it's key-value pairs.

4. **Q:** Difference between HashMap and Hashtable?
   A: HashMap is not synchronized; Hashtable is synchronized and legacy.

5. **Q:** What is the difference between HashSet and TreeSet?

   A: HashSet is unordered; TreeSet is sorted.

6. **Q:** What is the difference between List and Set?

   A: List allows duplicates; Set doesn't.

7. **Q:** What is the use of Iterator?

   A: To traverse a collection safely.

8. **Q:** Difference between `Iterator` and `ListIterator`?

   A: `ListIterator` can traverse both forward and backward.

9. **Q:** What is fail-fast and fail-safe?

   A: Fail-fast throws `ConcurrentModificationException`, fail-safe does not.

10. **Q:** Which classes are fail-fast?

    A: ArrayList, HashMap.

11. **Q:** Which classes are fail-safe?

    A: ConcurrentHashMap, CopyOnWriteArrayList.

12. **Q:** What is the time complexity of HashMap get()?

    A: O(1) in average case.

13. **Q:** Can HashMap have null key?

A: Yes, one null key.

14. **Q:** Can HashMap have null values?

A: Yes, multiple null values.

15. **Q:** What is the difference between Comparable and Comparator?

A: Comparable is for natural order; Comparator is for custom order.

16. **Q:** Can we sort HashSet?

A: No, use TreeSet for sorting.

17. **Q:** How does HashSet work internally?

A: Uses HashMap for storing elements.

18. **Q:** Difference between Array and ArrayList?

A: Array is fixed size; ArrayList is dynamic.

19. **Q:** Which class should be used in multithreaded environment?

A: CopyOnWriteArrayList or Collections.synchronizedList()

20. **Q:** Which Map ensures thread-safety?

A: ConcurrentHashMap

✅ Topic Outcome:

After this topic, learners can:

- Choose appropriate collection types based on the use case.
- Use `List`, `Set`, `Map`, and `Queue` effectively.
- Understand time complexity and memory optimization in collections.
- Handle real-time problems using dynamic data structures.

✅ Summary:

- Java Collections Framework is one of the most powerful tools for handling data dynamically.
- From ordered lists to unique sets and key-value pairs, collections are essential for modern Java development and mastering data structure usage in object-oriented programming.

🔹 Topic 28: File Handling in Java (Working with Streams)

✅ Definition:

- **File Handling** in Java refers to the mechanism of reading from and writing to files (text/binary) on the file system.
- Java provides the **java.io** and **java.nio** packages to manage file input/output using **Streams** (byte and character-based).
- **Streams** in Java are abstract representations of input/output devices used to read data from sources (like files, networks) and write data to destinations.

✅ Use Case:

- Storing user data into a file
- Logging application events
- Loading configuration from a file
- Reading input data for processing

✅ Real-Time Usage:

Scenario	File Operation Used
Save logs of transactions	FileWriter / BufferedWriter
Read data from CSV for processing	FileReader / BufferedReader
Exporting reports	PrintWriter / OutputStreamWriter
Storing form data in a file	FileOutputStream / FileWriter

✅ Architecture Diagram: File Handling Stream Flow

```
                ┌──────────────────┐
                │     Java File    │
                └────────┬─────────┘
                         │
              ┌──────────┴────────────┐
              │                       │
   InputStream (Byte)        Reader (Character)
              │                       │
```

```
┌───────────┴───────────┐        ┌─────────┴─────────┐

│ FileInputStream       │        │ FileReader        │

│ BufferedInputStream   │        │ BufferedReader    │

└───────────────────────┘        └───────────────────┘



            ┌───────────────┬──────────────┐

            │                               │

    OutputStream (Byte)            Writer (Character)

            │                               │

┌───────────┴───────────┐        ┌─────────┴─────────┐

│ FileOutputStream      │        │ FileWriter        │

│ BufferedOutputStream  │        │ BufferedWriter    │

└───────────────────────┘        └───────────────────┘
```

✅ Syntax:

```
// Reading from a file
BufferedReader reader = new BufferedReader(new
FileReader("input.txt"));
String line = reader.readLine();
reader.close();
```

```
// Writing to a file
BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"));
writer.write("Hello, File!");
writer.close();
```

✅ Keywords:

`File`, `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, `InputStream`, `OutputStream`, `try-with-resources`, `IOException`

✅ Types of Streams in File Handling:

Stream Type	**Description**	**Classes Used**
Byte Stream	Handles binary data	`FileInputStream`, `FileOutputStream`
Character Stream	Handles text data	`FileReader`, `FileWriter`

Buffered Stream	Improves performance using buffers	`BufferedReader`, `BufferedWriter`
Print Stream	Used for formatted output	`PrintWriter`, `PrintStream`

✅ Simple Example: Writing to a File

```
import java.io.*;

public class WriteFileExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new 
FileWriter("output.txt");
            writer.write("Java File Handling Example");
            writer.close();
            System.out.println("Successfully written to 
file.");
        } catch (IOException e) {
            System.out.println("Error occurred: " + 
e.getMessage());
        }
    }
}
```

✅ Simple Example: Reading from a File

```
import java.io.*;

public class ReadFileExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("output.txt"));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            System.out.println("Error reading file: " +
e.getMessage());
        }
    }
}
```

✅ Real-Time Example: Logging to a File

```
import java.io.*;

public class LogToFile {
    public static void main(String[] args) {
        try (BufferedWriter logWriter = new
BufferedWriter(new FileWriter("log.txt", true))) {
            logWriter.write("User logged in at " +
java.time.LocalDateTime.now());
            logWriter.newLine();
            System.out.println("Log written.");
        } catch (IOException e) {
            System.out.println("Logging failed.");
        }
    }
}
```

✅ 15 MCQ Questions with Answers:

1. Which class is used to read character data?

   A) FileInputStream

   B) FileReader

   ✅ **Answer: B**

2. Which class provides buffering for character input?

   A) BufferedReader

   ✅ **Answer: A**

3. What is used to write formatted data to a file?

   A) PrintWriter

   ✅ **Answer: A**

4. Which package is required for file handling?

   A) java.io

   ✅ **Answer: A**

5. `readLine()` returns what?

   A) char

B) String

✅ **Answer: B**

6. Which stream is used for binary input?

   A) FileInputStream

   ✅ **Answer: A**

7. `FileWriter` extends which class?

   A) OutputStream

   B) Writer

   ✅ **Answer: B**

8. Can `FileReader` read binary files?

   A) Yes

   B) No

   ✅ **Answer: B**

9. What method is used to write a string?

   A) write()

   ✅ **Answer: A**

10. Can you append using FileWriter?

    A) Yes, using constructor overload

    ✅ **Answer: A**

11. What happens if file not found in FileReader?

A) NullPointerException

B) FileNotFoundException

✅ **Answer: B**

12. What method closes the file stream?

A) end()

B) close()

✅ **Answer: B**

13. Which class reads line-by-line?

A) BufferedReader

✅ **Answer: A**

14. Can you use try-with-resources with file streams?

A) Yes

✅ **Answer: A**

15. What does `FileWriter("file.txt", true)` do?

A) Overwrites file

B) Appends to file

✅ **Answer: B**

✅ 20 Interview Q&A:

1. **Q:** What is file handling in Java?
   A: It's the process of reading from and writing to files using `java.io` or `java.nio`.

2. **Q:** Which classes are used for reading characters from a file?
   A: `FileReader`, `BufferedReader`

3. **Q:** How do you write to a file in Java?
   A: Using `FileWriter` or `BufferedWriter`.

4. **Q:** What is the difference between Reader and InputStream?
   A: `Reader` is for characters, `InputStream` is for bytes.

5. **Q:** What is buffering in file handling?
   A: Temporary memory to improve performance using buffered classes.

6. **Q:** What is the use of `flush()`?
   A: To forcefully write buffered content to file.

7. **Q:** Can you append data to an existing file?
   A: Yes, using `FileWriter("file.txt", true)`.

8. **Q:** How do you handle `FileNotFoundException`?

   A: Using try-catch or declaring with `throws`.

9. **Q:** What is the role of `PrintWriter`?

   A: Used for writing formatted text to files.

10. **Q:** What is the difference between `write()` and `append()`?

    A: `write()` overwrites, `append()` adds data at the end.

11. **Q:** Why is `close()` method used?

    A: To release system resources.

12. **Q:** What is `try-with-resources`?

    A: Auto-closes resources after try block.

13. **Q:** How to read multiple lines from a file?

    A: Using `BufferedReader.readLine()` in a loop.

14. **Q:** What happens if the file doesn't exist?

    A: `FileNotFoundException` is thrown.

15. **Q:** Can we read/write binary files?

    A: Yes, using `FileInputStream` and `FileOutputStream`.

16. **Q:** How do we ensure a file is always closed?

    A: Use try-with-resources.

17. **Q:** Can you check if file exists?

    A: Yes, using `File.exists()` method.

18. **Q:** Is file handling thread-safe?

    A: No, it must be managed manually or use synchronization.

19. **Q:** What's the default encoding for file writing?

    A: Platform-dependent (often UTF-8 or ANSI).

20. **Q:** How can you delete a file in Java?

    A: Using `File.delete()`.

✅ Topic Outcome:

After completing this topic, learners can:

- Read and write files using streams.
- Understand and differentiate between byte and character streams.
- Use buffered classes for performance.

- Implement real-world scenarios like logs, file-based storage, reports, etc.

✅ Summary:

- File handling in Java is a critical feature for applications that require persistent data storage.
- Mastery of streams enables reading, writing, and managing file I/O with performance and safety, using the versatile `java.io` package.

🔹 Topic 29: Implementing Thread Synchronization and Synchronized Methods in Java

---

✅ Definition:

- **Thread Synchronization** in Java is a technique to control the access of multiple threads to shared resources.
- It ensures **data consistency** when multiple threads try to read and write shared variables or objects concurrently.

- The `synchronized` keyword in Java is used to lock methods or blocks so that only one thread can execute them at a time, preventing **race conditions** and **data inconsistency**.

✅ Use Case:

- Managing **shared resources** (e.g., counters, logs, files).
- Avoiding **concurrent modification errors**.
- Ensuring **thread-safe operations** in multi-threaded environments.

✅ Real-Time Usage:

Scenario	Why Synchronization is Needed
Bank transactions system	Prevent double withdrawal from same account
Railway booking system	Avoid issuing the same seat to two users
Logging services	Prevent interleaved log entries

Shopping cart updates	Ensure consistent stock updates

✅ Architecture Diagram: Thread Synchronization Flow

```
       ┌─────────────────────────────┐
       │       Shared Resource       │
       └────────────┬────────────────┘
                    │
    ┌───────────────┼────────────────┐
    ▼               ▼                ▼
Thread-1        Thread-2         Thread-3
    │               │                │
    └─────synchronized access────────┘
                    │
        Only one thread at a time
```

✅ Syntax:

➤ Synchronized Method:

```
public synchronized void methodName() {
    // Critical section
}
```

➤ Synchronized Block:

```
synchronized (objectReference) {
    // Critical section
}
```

✅ Keywords:

`synchronized`, `thread`, `Runnable`, `lock`, `multithreading`, `monitor`, `wait()`, `notify()`

✅ Simple Example: Without Synchronization

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}
```

```
public class NoSyncExample {
    public static void main(String[] args) throws
InterruptedException {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++)
counter.increment();
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final Count: " +
counter.count);  // Output may vary due to race condition
    }
}
```

✅ Example: With Synchronized Method

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

```
public class SyncMethodExample {
    public static void main(String[] args) throws
InterruptedException {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++)
counter.increment();
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final Count: " +
counter.count);  // Correct output: 2000
    }
}
```

✅ Example: Synchronized Block

```
class Printer {
    void printMessage(String message) {
        synchronized (this) {
            System.out.print("[ " + message);
            try { Thread.sleep(500); } catch (Exception e)
{}
            System.out.println(" ]");
        }
    }
}
```

✅ Real-Time Example: Bank Transaction

```
class BankAccount {
    private int balance = 1000;

    synchronized void withdraw(int amount) {
        if (balance >= amount) {
            System.out.println("Withdrawn: " + amount);
            balance -= amount;
        } else {
            System.out.println("Insufficient balance");
        }
    }

    int getBalance() {
        return balance;
    }
}
```

```
public class BankExample {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();

        Runnable r1 = () -> acc.withdraw(600);
        Runnable r2 = () -> acc.withdraw(700);

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

✅ 15 MCQ Questions with Answers:

1. What does `synchronized` do in Java?

   A) Starts a thread

   B) Pauses execution

   ✅ **Answer: Controls thread access to shared resource**

2. Which thread can access a synchronized method?

   ✅ **Answer: Only one thread at a time**

3. Where can you use `synchronized`?

   A) Method only

   B) Block only

   ✅ **Answer: Both A and B**

4. Can a static method be synchronized?

   ✅ **Answer: Yes**

5. What is a critical section?

   ✅ **Answer: Code that must be executed by one thread at a time**

6. Can `synchronized` be used with a constructor?

   ✅ **Answer: No**

7. What is race condition?

✅ **Answer: Multiple threads modifying shared data simultaneously**

8. Which keyword notifies a waiting thread?

✅ **Answer: notify()**

9. What is thread starvation?

✅ **Answer: Thread never gets CPU time due to long waits**

10. What is used to implement wait-notify?

✅ **Answer: Object's monitor**

11. What is the default lock object in a synchronized method?

✅ **Answer: The current instance (`this`)**

12. Can we use synchronized in interface?

✅ **Answer: No**

13. Which statement is true about synchronized blocks?

✅ **Answer: They are used to reduce the scope of synchronization**

14. What is thread-safe code?

✅ **Answer: Code that can be safely executed by multiple**

threads

15. Is `ReentrantLock` part of synchronization?

    ✅ **Answer: No, it's from `java.util.concurrent.locks`**

✅ 20 Interview Q&A:

1. **Q:** What is synchronization?
   A: It's a process of controlling access to shared resources in multithreaded programming.

2. **Q:** What is the difference between synchronized method and block?
   A: Method locks the entire object; block allows fine-grained control.

3. **Q:** Can static methods be synchronized?
   A: Yes, they acquire a class-level lock.

4. **Q:** What is a monitor in Java?
   A: Every object in Java has an implicit monitor associated with it for thread control.

5. **Q:** How does `synchronized(this)` work?

   A: It locks the current object before executing the code inside.

6. **Q:** What are some alternatives to `synchronized`?

   A: `Lock`, `ReentrantLock`, `AtomicInteger`

7. **Q:** What is deadlock?

   A: When two or more threads are waiting forever for each other to release resources.

8. **Q:** Can we synchronize a constructor?

   A: No, because object is not fully created yet.

9. **Q:** What is `notify()` and `wait()`?

   A: Methods for inter-thread communication.

10. **Q:** Can synchronized methods be overridden?

    A: Yes, and the overriding method can be synchronized or not.

11. **Q:** What happens if two threads access different synchronized methods of same object?

    A: Only one gets the lock, other waits.

12. **Q:** Can we have synchronized static and non-static methods?

A: Yes, they lock different objects (class vs instance).

13. **Q:** What is atomicity in synchronization?
    A: Ensuring operations execute as one unbreakable unit.

14. **Q:** What happens if one thread does not release lock?
    A: Other threads remain blocked, possibly leading to deadlock.

15. **Q:** Can we use multiple locks in one class?
    A: Yes, by using different synchronized blocks.

16. **Q:** What is thread-safe class in Java?
    A: Class whose objects can be safely used across threads.

17. **Q:** Difference between Lock and synchronized?
    A: `Lock` gives more flexibility (tryLock, interruptible lock, etc.)

18. **Q:** Can threads interfere with each other without synchronization?
    A: Yes, leads to race condition.

19. **Q:** Does synchronization affect performance?
    A: Yes, slight overhead due to locking.

20. **Q:** How to make a class thread-safe?

A: Use `synchronized`, immutable objects, or thread-safe data structures.

✅ Topic Outcome:

After completing this topic, learners can:

- Apply synchronized methods and blocks to prevent race conditions.
- Handle concurrent data modifications safely.
- Write multi-threaded applications that are reliable and thread-safe.

✅ Summary:

- Thread synchronization is essential in multithreaded applications where shared resources are involved.
- Java's `synchronized` keyword provides a powerful way to manage concurrent access and ensure consistent, safe execution without corrupting data.

🔹 Topic 30: Multithreading in Java

✅ Definition:

- **Multithreading** in Java is a process of executing **two or more threads simultaneously** to perform multiple tasks concurrently.
- Each thread runs independently, sharing the same memory space, making Java programs **faster and more efficient**, especially in applications involving UI responsiveness, background processing, or real-time systems.
- A **thread** is the smallest unit of a process that can execute independently.
- Java provides built-in support for multithreading through the `Thread` class and `Runnable` interface.

✅ Use Case:

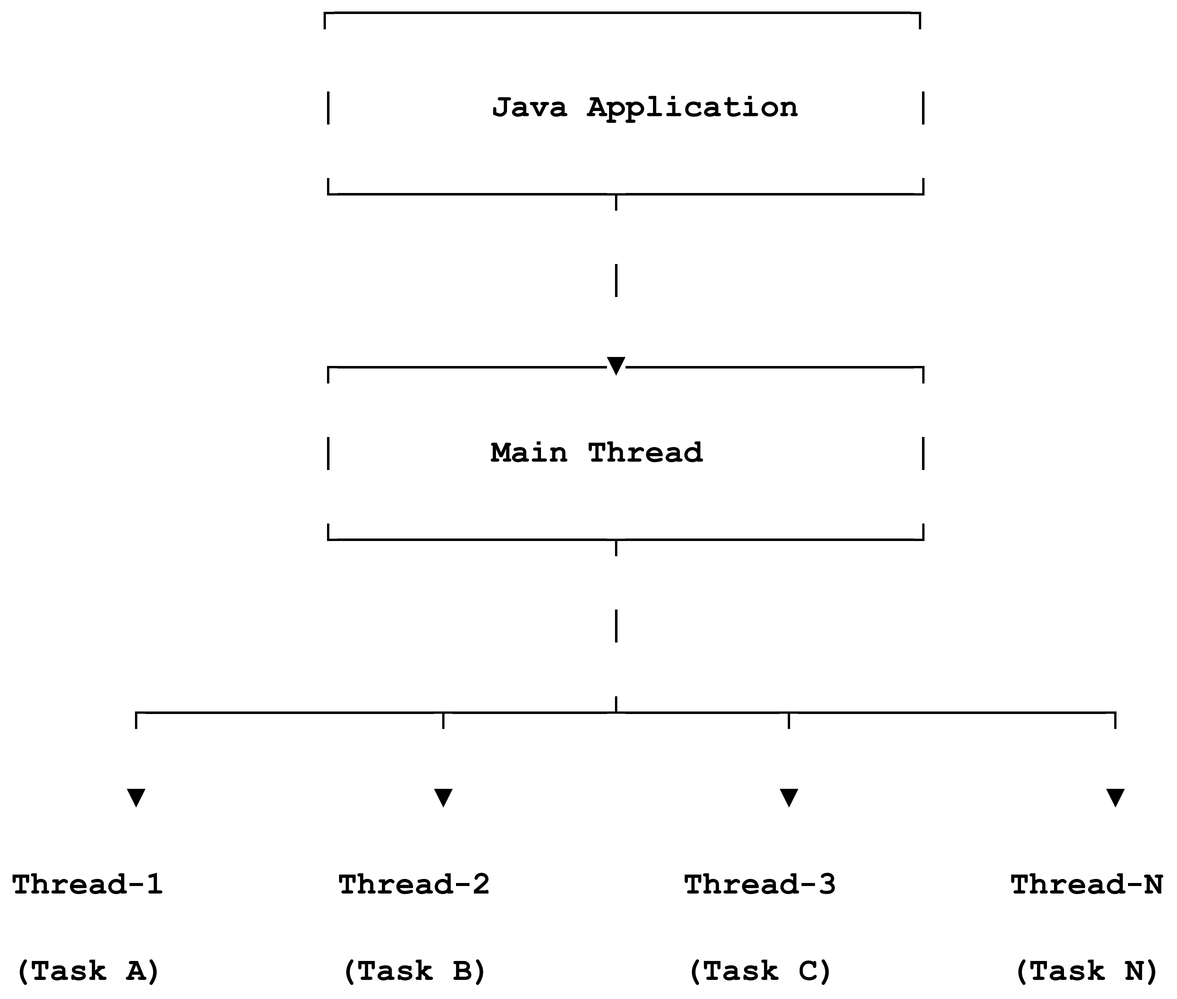
- Background file downloads
- Real-time chat applications
- Parallel searching and sorting

- Server request handling
- Video and audio streaming apps

✅ Real-Time Usage:

Application Area	**Example Use Case**
Web Servers	Handling multiple user requests
Mobile Apps	Background image upload
Games	AI processing, rendering, and sound in parallel
Banking Systems	Processing multiple transactions
IoT Systems	Reading from sensors concurrently

✅ Architecture Diagram: Java Multithreading Model



✅ Syntax:

➤ Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

➢ Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}
```

✅ Keywords:

`Thread`, `Runnable`, `start()`, `run()`, `sleep()`, `join()`, `synchronized`, `yield()`, `wait()`, `notify()`

✅ Simple Example: Using Thread Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " +
Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // creates new thread and runs
    }
}
```

✅ Simple Example: Using Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread: " +
Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

✅ Real-Time Example: Parallel Printing

```
class Printer extends Thread {
    String message;

    Printer(String message) {
        this.message = message;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(message + " " + i);
        }
    }

    public static void main(String[] args) {
        new Printer("Printing A").start();
        new Printer("Printing B").start();
    }
}
```

✅ Important Thread Methods:

Method	Description
`start()`	Starts thread execution
`run()`	Contains the code executed by thread
`sleep(ms)`	Pauses thread temporarily
`join()`	Waits for a thread to finish execution
`isAlive()`	Checks if thread is alive
`yield()`	Pauses current thread to let others execute

✅ Thread Lifecycle:

1. **New**
2. **Runnable**

3. **Running**
4. **Waiting/Blocked/Sleeping**
5. **Dead/Terminated**

What is a Thread?

- A Thread is a lightweight subprocess.
- It’s the smallest unit of processing that can be performed concurrently with other threads in a program.
- Threads allow parallel execution of tasks.

Java supports multithreading by:

- Extending the Thread class
- Implementing the Runnable interface
- Using ExecutorService (from Java 5 onwards)

Thread Life Cycle in Java:

A thread in Java can be in one of the following states:

`NEW → RUNNABLE → RUNNING → BLOCKED/WAITING → TERMINATED`

```
[NEW] --> start() --> [RUNNABLE]

            |

            v

 [RUNNING] --> sleep()/wait() -->[WAITING/TIMED_WAITING]

            |

            v

       Thread ends

       [TERMINATED]
```

1. Multiprocessing

Definition:

- Running multiple processes simultaneously on multiple CPUs or cores.

Key Idea:

- Each process runs independently with its own memory space.

Example:

- Running a video editor, browser, and a game at the same time on different CPU cores.

Real-life analogy:

- Multiple chefs (processes) cooking in different kitchens (separate memory).

2. Multitasking

Definition:

- Ability of an operating system to execute multiple tasks (programs) at the same time by rapidly switching between them (context switching).

Key Idea:

- Tasks appear to run at the same time (on single CPU) or truly run concurrently on multi-core CPUs.

Types:

Preemptive Multitasking – OS decides when to switch tasks (used in modern OS).

Cooperative Multitasking – Tasks voluntarily yield control.

Example:

- Listening to music while typing a document and downloading files.

Real-life analogy:

- One person (CPU) switching between writing, checking phone, and eating (tasks).

3. Multithreading

Definition:

- Running multiple threads (smaller units of a process) within the same process concurrently.

Key Idea:

- Threads share the same memory space and can communicate easily, making it lighter than multiprocessing.

Example:

- A web browser:
- One thread handles UI
- One handles network
- One handles rendering

Real-life analogy:

- One chef (process) preparing multiple dishes (threads) using the same kitchen (shared memory).

- Thread-->class
- Runnable-->interface

✅ 15 MCQ Questions with Answers:

1. Which method starts a thread?
   ✅ **Answer: start()**

2. Can you restart a thread after it finishes?
   ✅ **Answer: No**

3. Which interface provides thread behavior?
   ✅ **Answer: Runnable**

4. What is the return type of `run()`?
   ✅ **Answer: void**

5. Which method pauses a thread?
   ✅ **Answer: sleep()**

6. Can a class implement Runnable and extend another class?

   ✅ **Answer: Yes**

7. Which method checks if thread is alive?

   ✅ **Answer: isAlive()**

8. What does `yield()` do?

   ✅ **Answer: Pauses current thread for others**

9. What is the default priority of a thread?

   ✅ **Answer: 5**

10. Which thread is automatically created in every Java program?

    ✅ **Answer: Main thread**

11. Which is preferred: Runnable or Thread?

    ✅ **Answer: Runnable**

12. Can we override `start()` method?

    ✅ **Answer: Not recommended**

13. Which class is superclass of all threads?

    ✅ **Answer: Thread**

14. Is `run()` called directly to start thread?

    ✅ **Answer: No, use start() instead**

15. What causes race conditions?

    ✅ **Answer: Concurrent access without synchronization**

✅ 20 Interview Q&A:

1. **Q:** What is multithreading?

   A: It's the ability of a CPU or single core to execute multiple threads concurrently.

2. **Q:** Difference between Thread and Runnable?

   A: `Thread` is a class, `Runnable` is an interface. Runnable is preferred due to multiple inheritance.

3. **Q:** What is context switching?

   A: Switching CPU from one thread to another.

4. **Q:** What is thread priority?

   A: Determines the order of thread execution.

5. **Q:** How do you pause a thread?

   A: Using `sleep(milliseconds)`.

6. **Q:** How to safely access shared data?

   A: Use `synchronized` blocks or methods.

7. **Q:** What are daemon threads?

   A: Background threads like garbage collector.

8. **Q:** What is thread starvation?

   A: When low-priority threads are never executed.

9. **Q:** How is a thread created?

   A: Extend `Thread` or implement `Runnable`.

10. **Q:** When does a thread die?

    A: After `run()` completes or exception occurs.

11. **Q:** What is deadlock?

    A: When two threads wait forever for each other to release resources.

12. **Q:** Can we start a thread twice?

    A: No. It throws `IllegalThreadStateException`.

13. **Q:** What is thread-safe code?

    A: Code that can be safely executed by multiple threads.

14. **Q:** Why use `join()`?

    A: To wait for thread completion before proceeding.

15. **Q:** Is `start()` or `run()` used for execution?

    A: Use `start()`; calling `run()` won't create a new thread.

16. **Q:** Can threads share resources?

    A: Yes, but need synchronization to avoid data inconsistency.

17. **Q:** How to handle thread exceptions?

    A: Use try-catch within `run()` method.

18. **Q:** What is the use of `Thread.sleep()`?

    A: Delays execution of thread temporarily.

19. **Q:** Can we synchronize run() method?

    A: Yes, but better to synchronize shared resources.

20. **Q:** Is Java multithreaded by default?

    A: Yes, it supports multithreading from the beginning.

✅ Topic Outcome:

After completing this topic, learners can:

- Create and manage multiple threads
- Use `Thread` and `Runnable` effectively
- Understand thread lifecycle and common thread methods
- Apply synchronization to avoid concurrency issues

✅ Summary:

- Multithreading is a vital concept in Java for building efficient, concurrent applications.
- By utilizing threads and synchronization techniques, developers can create responsive, fast-performing programs suitable for real-world environments.

🔹 Topic 31: JDBC Connectivity in Java

✅ Definition:

- **JDBC (Java Database Connectivity)** is a Java API that enables Java applications to connect, interact, and manipulate relational databases using SQL commands.

- It provides classes and interfaces to establish a connection, execute queries, and retrieve results from databases like MySQL, Oracle, PostgreSQL, etc.

✅ Use Case:

- Performing **CRUD** operations on relational databases.
- Connecting enterprise Java applications to backend databases.
- Generating reports from stored data.
- Building web apps with data-driven features (e.g., login, dashboard, reports).

✅ Real-Time Usage:

Application Area	**Usage**
Banking Systems	Customer & transaction database access
Online Shopping Sites	Product catalog, user data, orders

Employee Management	Employee records CRUD operations
University Portals	Student marks, attendance tracking, fee updates

---

✅ Architecture Diagram: JDBC Communication Flow

```
┌──────────────┐      JDBC API      ┌────────────────┐
│   Java App   │ ─────────────────▶ │  JDBC Driver   │
└──────────────┘                    └───────┬────────┘
                                            │
                                            │
                                            ▼
                                    ┌───────────────┐
                                    │   Database    │
                                    └───────────────┘
```

✅ JDBC Components:

Component	Description
`DriverManager`	Manages JDBC drivers and establishes connections
`Connection`	Represents the connection between Java and DB
`Statement`	Executes SQL queries (static)
`PreparedStatement`	Executes parameterized queries (dynamic + secure)
`ResultSet`	Stores results returned from SELECT queries

✅ Syntax (JDBC Connection):

```
Class.forName("com.mysql.cj.jdbc.Driver"); // Load driver
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb", "username",
"password");

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

while (rs.next()) {
    System.out.println(rs.getString("username"));
}

con.close();
```

✅ Keywords:

`Connection`, `DriverManager`, `ResultSet`, `Statement`, `PreparedStatement`, `executeQuery()`, `executeUpdate()`, `close()`

✅ Simple Example: Read Data from MySQL

```
import java.sql.*;

public class SimpleRead {
    public static void main(String[] args) throws Exception
{
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/demo", "root",
"password");

        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM
student");

        while (rs.next()) {
            System.out.println(rs.getInt("id") + " " +
rs.getString("name"));
```

```
        }

        con.close();
    }
}
```

✅ Example: Insert Record using `PreparedStatement`

```
import java.sql.*;

public class InsertExample {
    public static void main(String[] args) throws Exception
{
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/demo", "root",
"password");

        String query = "INSERT INTO student(name, age)
VALUES (?, ?)";
        PreparedStatement pst =
con.prepareStatement(query);
        pst.setString(1, "John");
        pst.setInt(2, 22);

        int rows = pst.executeUpdate();
        System.out.println("Inserted: " + rows);

        con.close();
    }
}
```

✅ Example: Update and Delete

```
// UPDATE
String query = "UPDATE student SET age=? WHERE name=?";
PreparedStatement pst = con.prepareStatement(query);
pst.setInt(1, 25);
pst.setString(2, "John");
pst.executeUpdate();

// DELETE
String del = "DELETE FROM student WHERE name=?";
PreparedStatement delStmt = con.prepareStatement(del);
delStmt.setString(1, "John");
delStmt.executeUpdate();
```

✅ 15 MCQ Questions with Answers:

1. What does JDBC stand for?

   ✅ **Java Database Connectivity**

2. Which class loads the database driver?

   ✅ **Class.forName()**

3. Which interface manages connection to DB?

   ✅ **Connection**

4. What does `executeUpdate()` return?

   ✅ **Number of rows affected**

5. Which class is used to run parameterized queries?

   ✅ **PreparedStatement**

6. What is used to store SELECT results?

   ✅ **ResultSet**

7. Which JDBC driver type is platform-independent?

   ✅ **Type 4**

8. What method is used to run SELECT queries?

   ✅ **executeQuery()**

9. What is the purpose of `DriverManager`?

   ✅ **To establish DB connections**

10. Which method is used to close connection?

    ✅ **con.close()**

11. Which is more secure: Statement or PreparedStatement?

    ✅ **PreparedStatement**

12. Can you use JDBC with Oracle DB?

    ✅ **Yes**

13. Can ResultSet be updated?
 ✅ **Yes (with correct settings)**

14. Which exception is commonly thrown in JDBC?
 ✅ **SQLException**

15. Can JDBC access NoSQL databases?
 ✅ **No (primarily for RDBMS)**

✅ 20 Interview Q&A:

1. **Q:** What is JDBC?
 A: Java API to connect Java applications with relational databases.

2. **Q:** Explain the steps to connect to a DB using JDBC.
 A: Load driver → Create connection → Create statement → Execute query → Close connection

3. **Q:** What is DriverManager?
 A: A class that loads DB drivers and manages DB connections.

4. **Q:** What is the difference between `Statement` and `PreparedStatement`?

A: `Statement` is for static queries, `PreparedStatement` for dynamic, parameterized queries.

5. **Q:** What is a ResultSet?

   A: It stores the result of SELECT queries.

6. **Q:** Which method is used for INSERT, UPDATE, DELETE?

   A: `executeUpdate()`

7. **Q:** Which method is used for SELECT queries?

   A: `executeQuery()`

8. **Q:** How do you handle SQL injection?

   A: Use `PreparedStatement`.

9. **Q:** What is the use of `Class.forName()`?

   A: Loads and registers the JDBC driver.

10. **Q:** Can you execute stored procedures in JDBC?

    A: Yes, using `CallableStatement`.

11. **Q:** What are JDBC driver types?

    A: Type 1 to Type 4 (Type 4 is pure Java driver)

12. **Q:** What is batch processing in JDBC?

    A: Executing multiple queries together using `addBatch()` and `executeBatch()`.

13. **Q:** How to retrieve auto-generated keys?

    A: Use `getGeneratedKeys()`.

14. **Q:** Can you have multiple connections open?

    A: Yes, each with its own object.

15. **Q:** What is a transaction in JDBC?

    A: A set of queries executed as a unit using `commit()` and `rollback()`.

16. **Q:** What is the use of `setAutoCommit(false)`?

    A: Disables auto-commit to manually handle transactions.

17. **Q:** How to prevent memory leaks in JDBC?

    A: Always close `Connection`, `Statement`, and `ResultSet`.

18. **Q:** Can JDBC connect to cloud databases?

    A: Yes, using proper drivers and endpoints.

19. **Q:** What is SQLWarning?

  A: Used to capture non-fatal database warnings.

20. **Q:** Which JDBC version supports `try-with-resources`?

  A: JDBC 4.1+ (Java 7+)

✅ Topic Outcome:

After this topic, learners can:

- Connect and interact with relational databases using JDBC.
- Perform CRUD operations using Statement & PreparedStatement.
- Implement secure, efficient database-driven Java applications.

✅ Summary:

- JDBC is essential for database interaction in Java.
- From setting up connections to executing queries and managing results, JDBC enables seamless integration between Java apps and relational databases, forming the backbone of most enterprise applications.