**Java notes**

**Unit 4,5**

Unit 4

* Multithreaded program

Multithreading is a programming concept that allows multiple threads to run concurrently within a single program. In Java, threads can be created using two primary methods: extending the Thread class or implementing the Runnable interface.

**Creating Threads in Java**

**1. Using the Thread Class**

To create a thread by extending the Thread class, follow these steps:

1. **Extend the Thread class**.
2. **Override the run() method** to define the code that should be executed by the thread.
3. **Create an instance of the class** and call the start() method.

**Example: Extending the Thread Class**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println("Thread: " + Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadExample {

public static void main(String[] args) {

// Creating two thread instances

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

// Starting the threads

thread1.start();

thread2.start();

}

}

**Key Concepts in Multithreading:**

1. **Thread Lifecycle**: Understand the various states a thread can be in during its lifecycle.
2. **Synchronization**: Control the access of multiple threads to shared resources.
3. **Thread Interference**: When multiple threads access shared data, they can interfere with each other, leading to data inconsistency.
4. **Deadlock**: A situation where two or more threads are blocked forever, each waiting for the other to release a resource.

* Java thread model

The Java Thread Model describes how threads are created, managed, and executed in a Java application. It defines the interactions between threads, including their life cycle, states, and how they share resources. Understanding the thread model is crucial for developing efficient and safe multithreaded applications.

**Key Components of the Java Thread Model**

1. **Thread Creation**:
   * In Java, threads can be created by extending the Thread class or implementing the Runnable interface. Both approaches allow you to define the code that will run in the new thread.
2. **Thread Lifecycle**:
   * A thread goes through various states during its lifecycle, each indicating the current status of the thread. The main states are:
     + **New**: The thread is created but not yet started.
     + **Runnable**: The thread is ready to run and is waiting for CPU time. It can be in this state after calling start() or if it is waiting for resources.
     + **Blocked**: The thread is waiting to acquire a lock to enter a synchronized block/method.
     + **Waiting**: The thread is waiting indefinitely for another thread to perform a particular action (e.g., waiting for a notification).
     + **Timed Waiting**: The thread is waiting for a specified period (e.g., using sleep() or wait(timeout)).
     + **Terminated**: The thread has completed its execution or has been terminated.
3. **Thread States Diagram**:

+---------+ start() +---------+

| New |------------------->| Runnable|

+---------+ +---------+

| / | \

| / | \

| / | \

| v | v

+---------+ +---------+ | +---------+

|Blocked |<-----------|Timed Waiting|<--| Waiting |

+---------+ +---------+ | +---------+

^ ^ |

| | |

| notify()/notifyAll()|

| | |

+-------------------+ |

|

+---------+ |

|Terminated|<---------------------+

+---------+

1. **Thread Priorities**:

* Each thread has a priority level (between 1 and 10, where 1 is the lowest and 10 is the highest) that can influence the thread's scheduling. However, thread priority is not a guarantee of execution order; it is merely a hint to the thread scheduler.

1. **Thread Scheduling**:

* The thread scheduler is part of the Java Virtual Machine (JVM) and is responsible for determining the order in which threads are executed. The scheduling algorithm can vary based on the operating system.

1. **Synchronization**:

* Synchronization is crucial in a multithreaded environment to ensure that shared resources are accessed safely. Java provides the synchronized keyword to prevent thread interference and maintain data consistency.
* Other synchronization tools include:
  + **Locks**: Use java.util.concurrent.locks.Lock for more complex thread control.
  + **Condition Variables**: Used with locks for waiting and notifying threads.
  + **Atomic Variables**: Classes in java.util.concurrent.atomic allow for thread-safe operations on single variables.

1. **Inter-thread Communication**:
   * Java provides mechanisms like wait(), notify(), and notifyAll() to allow threads to communicate about the availability of resources.

**Example: Thread Lifecycle in Java**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(1000); // Simulate some work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadLifecycleExample {

public static void main(String[] args) {

// Create a new thread (New state)

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

// Start the threads (Runnable state)

thread1.start();

thread2.start();

try {

// Main thread waits for thread1 to complete (Waiting state)

thread1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// At this point, thread1 has terminated

System.out.println("Thread1 has finished execution.");

}

}

* The main thread

In Java, the **main thread** is the thread that starts the execution of a Java program. Every Java application begins with the execution of the main() method, which is defined as follows:

public static void main(String[] args) {

// Code to be executed

}

**Key Characteristics of the Main Thread:**

1. **Entry Point**:
   * The main() method is the entry point for any standalone Java application. When you run a Java program, the Java Virtual Machine (JVM) creates the main thread to execute this method.
2. **Thread Creation**:
   * The main thread is created by the JVM when the program starts. You can create additional threads from the main thread using the Thread class or implementing the Runnable interface.
3. **Thread Name**:
   * The main thread is usually named "main" by default. You can change its name using the setName() method.
4. **Lifecycle**:
   * The main thread goes through the same lifecycle states as any other thread: **New**, **Runnable**, **Blocked**, **Waiting**, **Timed Waiting**, and **Terminated**.
5. **Termination**:
   * The main thread terminates when the main() method finishes executing. Once the main thread ends, the entire program terminates unless there are other non-daemon threads running.

**Example of the Main Thread in Action**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(500); // Simulate some work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class MainThreadExample {

public static void main(String[] args) {

System.out.println("Main thread: " + Thread.currentThread().getName());

// Creating additional threads

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

// Start the threads

thread1.start(); // Starts thread1

thread2.start(); // Starts thread2

// Main thread continues executing

for (int i = 0; i < 5; i++) {

System.out.println("Main thread - Count: " + i);

try {

Thread.sleep(500); // Simulate some work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

// Wait for other threads to finish

try {

thread1.join(); // Wait for thread1 to finish

thread2.join(); // Wait for thread2 to finish

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Main thread finished execution.");

}

}

* Creating thread

Creating a thread in Java can be accomplished in two primary ways: by extending the Thread class or by implementing the Runnable interface. Below are detailed explanations and examples for both methods.

**1. Creating a Thread by Extending the Thread Class**

When you extend the Thread class, you need to override the run() method to define the code that will execute in the new thread.

**Steps to Create a Thread by Extending Thread:**

1. Create a new class that extends Thread.
2. Override the run() method.
3. Create an instance of your thread class.
4. Call the start() method to initiate the thread.

**Example: Extending the Thread Class**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadExample {

public static void main(String[] args) {

// Creating an instance of MyThread

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

// Starting the threads

thread1.start(); // Starts thread1

thread2.start(); // Starts thread2

}

}

**2. Creating a Thread by Implementing the Runnable Interface**

Implementing the Runnable interface is often preferred for better flexibility, as it allows you to extend other classes as well.

**Steps to Create a Thread by Implementing Runnable:**

1. Create a class that implements the Runnable interface.
2. Override the run() method.
3. Create an instance of Thread, passing the Runnable instance as an argument.
4. Call the start() method on the Thread instance.

**Example: Implementing the Runnable Interface**

class MyRunnable implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class RunnableExample {

public static void main(String[] args) {

// Creating an instance of MyRunnable

MyRunnable myRunnable = new MyRunnable();

// Creating thread instances

Thread thread1 = new Thread(myRunnable);

Thread thread2 = new Thread(myRunnable);

// Starting the threads

thread1.start(); // Starts thread1

thread2.start(); // Starts thread2

}

}

* Creating multiple thread

**Steps to Create Multiple Threads in Java**

1. **Define a Class for the Thread**: Create a class that extends the Thread class or implements the Runnable interface.
2. **Override the run() Method**: Implement the run() method with the code you want to execute in the thread.
3. **Instantiate the Thread Class**: Create multiple instances of your thread class.
4. **Start Each Thread**: Call the start() method on each thread instance to begin execution.

Sample Code: Creating Multiple Threads by Extending Thread Class

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(500); // Sleep for 0.5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class MultipleThreadsExample {

public static void main(String[] args) {

// Create multiple thread instances

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

MyThread thread3 = new MyThread();

// Start the threads

thread1.start();

thread2.start();

thread3.start();

}

}

* Using isAlive() and join()

**Using isAlive() and join() Methods in Java Threads**

The isAlive() and join() methods are used to manage and monitor the state of threads in Java.

* **isAlive()**: This method checks if a thread has been started and has not yet died (terminated). It returns true if the thread is still running and false if it has finished execution.
* **join()**: This method makes the calling thread wait until the thread on which it is called has finished executing. This is useful for ensuring that a particular thread completes before proceeding with the next steps in the program.

**Steps to Use isAlive() and join()**

1. **Create Thread Instances**: Define your thread class and create instances of it.
2. **Start the Threads**: Call the start() method on each thread.
3. **Check Thread Status with isAlive()**: Use isAlive() to check if a thread is still running.
4. **Use join() to Wait**: Call join() on a thread to wait for its completion.

**Sample Code: Using isAlive() and join()**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(500); // Sleep for 0.5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadAliveAndJoinExample {

public static void main(String[] args) {

// Create multiple thread instances

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

MyThread thread3 = new MyThread();

// Start the threads

thread1.start();

thread2.start();

thread3.start();

// Use isAlive() and join()

try {

// Check if threads are alive

System.out.println("Thread1 is alive: " + thread1.isAlive());

System.out.println("Thread2 is alive: " + thread2.isAlive());

System.out.println("Thread3 is alive: " + thread3.isAlive());

// Wait for threads to finish

thread1.join(); // Wait for thread1 to finish

thread2.join(); // Wait for thread2 to finish

thread3.join(); // Wait for thread3 to finish

// After joining, threads are no longer alive

System.out.println("Thread1 is alive: " + thread1.isAlive());

System.out.println("Thread2 is alive: " + thread2.isAlive());

System.out.println("Thread3 is alive: " + thread3.isAlive());

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("All threads have finished execution.");

}

}

* Thread priorities

In Java, thread priorities are used to indicate the relative importance of threads when scheduling them for execution by the Java Virtual Machine (JVM). Thread priorities can help the scheduler make decisions about which thread to run next, especially in environments where multiple threads are competing for CPU time.

**Thread Priority Levels**

Java provides a set of constants in the Thread class to define thread priorities:

* **Thread.MIN\_PRIORITY**: A constant for the minimum priority (value = 1).
* **Thread.NORM\_PRIORITY**: A constant for the default priority (value = 5).
* **Thread.MAX\_PRIORITY**: A constant for the maximum priority (value = 10).

**Default Priority**

When a thread is created, it inherits the priority of the thread that created it. If the priority is not explicitly set, it defaults to NORM\_PRIORITY (5).

**Setting Thread Priority**

You can set the priority of a thread using the setPriority(int priority) method. The priority value must be between MIN\_PRIORITY (1) and MAX\_PRIORITY (10).

**Example of Using Thread Priorities**

class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(500); // Sleep for 0.5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ThreadPriorityExample {

public static void main(String[] args) {

// Create threads

MyThread thread1 = new MyThread("Low Priority Thread");

MyThread thread2 = new MyThread("Normal Priority Thread");

MyThread thread3 = new MyThread("High Priority Thread");

// Set thread priorities

thread1.setPriority(Thread.MIN\_PRIORITY); // Low priority

thread2.setPriority(Thread.NORM\_PRIORITY); // Normal priority

thread3.setPriority(Thread.MAX\_PRIORITY); // High priority

// Start the threads

thread1.start();

thread2.start();

thread3.start();

}

}

* Synchronization

**Synchronization** is a mechanism in Java that ensures that only one thread can access a resource at a time. It is used to prevent thread interference and to ensure data consistency when multiple threads are operating on shared resources.

**Why Synchronization is Necessary**

* **Data Consistency**: When multiple threads modify shared data simultaneously, it can lead to inconsistent or unpredictable results.
* **Thread Safety**: Synchronization helps in making code thread-safe, meaning that the code behaves correctly when accessed by multiple threads.

**Types of Synchronization**

1. **Method Synchronization**: Synchronizing the entire method to restrict access to that method by multiple threads.
2. **Block Synchronization**: Synchronizing specific code blocks within a method, allowing for more granular control over which parts of the code can be accessed by multiple threads.

**How to Implement Synchronization**

**1. Synchronized Methods**

You can declare a method as synchronized by adding the synchronized keyword to its declaration.

class Counter {

private int count = 0;

// Synchronized method

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

**2. Synchronized Blocks**

You can synchronize specific blocks of code within a method by using the synchronized keyword with an object lock.

class Counter {

private int count = 0;

public void increment() {

synchronized (this) { // Synchronized block

count++;

}

}

public int getCount() {

return count;

}

}

* Inter thread communication

Inter-thread communication is a mechanism that allows threads to communicate with each other and coordinate their actions. In Java, this is achieved using methods provided by the Object class, specifically wait(), notify(), and notifyAll(). These methods help manage thread execution and resource sharing efficiently.

**Key Concepts**

1. **Wait**: The wait() method makes the current thread wait until another thread invokes the notify() or notifyAll() method on the same object. It releases the lock on the object and enters the waiting state.
2. **Notify**: The notify() method wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, one is selected randomly to be awakened.
3. **NotifyAll**: The notifyAll() method wakes up all threads that are waiting on the object's monitor.

**How Inter-Thread Communication Works**

* **Monitors**: Every object in Java has a monitor that threads can use to synchronize their access to the object's resources.
* **Waiting and Notifying**: Threads must call wait() and notify()/notifyAll() from within synchronized blocks or methods to avoid IllegalMonitorStateException.

**Example of Inter-Thread Communication**

Here’s a simple producer-consumer example demonstrating inter-thread communication using wait() and notify():

class SharedResource {

private int data;

private boolean available = false; // Flag to indicate data availability

// Producer method

public synchronized void produce(int value) {

while (available) {

try {

wait(); // Wait until the data is consumed

} catch (InterruptedException e) {

e.printStackTrace();

}

}

data = value; // Produce data

System.out.println("Produced: " + data);

available = true; // Data is now available

notify(); // Notify consumers

}

// Consumer method

public synchronized int consume() {

while (!available) {

try {

wait(); // Wait until data is produced

} catch (InterruptedException e) {

e.printStackTrace();

}

}

available = false; // Data is now consumed

System.out.println("Consumed: " + data);

notify(); // Notify producers

return data;

}

}

class Producer extends Thread {

private SharedResource sharedResource;

public Producer(SharedResource sharedResource) {

this.sharedResource = sharedResource;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

sharedResource.produce(i);

}

}

}

class Consumer extends Thread {

private SharedResource sharedResource;

public Consumer(SharedResource sharedResource) {

this.sharedResource = sharedResource;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

sharedResource.consume();

}

}

}

public class InterThreadCommunicationExample {

public static void main(String[] args) {

SharedResource sharedResource = new SharedResource();

Producer producer = new Producer(sharedResource);

Consumer consumer = new Consumer(sharedResource);

producer.start(); // Start the producer thread

consumer.start(); // Start the consumer thread

}

}

* Suspending resuming and stopping threads

In Java, controlling the execution of threads is essential for managing resources and ensuring that applications run smoothly. However, it's important to note that Java's original methods for suspending, resuming, and stopping threads (suspend(), resume(), and stop()) are deprecated due to safety concerns. Instead, developers are encouraged to use other techniques, such as flags or higher-level concurrency utilities from the java.util.concurrent package.

**Deprecated Methods**

* **suspend()**: Temporarily pauses a thread without releasing the locks it holds, which can lead to deadlocks if not managed carefully.
* **resume()**: Resumes a suspended thread, but may also lead to deadlocks if the thread was holding locks when suspended.
* **stop()**: Forcefully terminates a thread, which can leave shared resources in an inconsistent state.

**Recommended Approaches**

Since these methods are deprecated, the preferred way to manage thread lifecycle in modern Java is to use flags or other synchronization mechanisms. Below are some alternatives for controlling thread execution:

**1. Using Volatile Flags**

You can use a volatile boolean flag to control the suspension and resumption of a thread safely:

class MyThread extends Thread {

private volatile boolean running = true;

@Override

public void run() {

while (running) {

// Perform task

System.out.println(Thread.currentThread().getName() + " is running");

try {

Thread.sleep(1000); // Simulate work

} catch (InterruptedException e) {

Thread.currentThread().interrupt(); // Preserve interrupt status

break; // Exit loop on interruption

}

}

}

public void stopThread() {

running = false; // Set flag to false to stop the thread

}

}

public class ThreadControlExample {

public static void main(String[] args) throws InterruptedException {

MyThread thread = new MyThread();

thread.start(); // Start the thread

// Let the thread run for 5 seconds

Thread.sleep(5000);

// Stop the thread

thread.stopThread();

thread.join(); // Wait for the thread to finish

System.out.println("Thread has stopped.");

}

}

**2. Using join() Method**

You can also control the lifecycle of threads by using the join() method, which allows one thread to wait for the completion of another. This is not suspension but can be used to coordinate execution.

**3. Using java.util.concurrent Package**

For more complex thread management, consider using classes from the java.util.concurrent package, such as ExecutorService, CountDownLatch, or CyclicBarrier, which provide a higher-level API for handling concurrency.

**Example Using ExecutorService**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class ExecutorServiceExample {

public static void main(String[] args) throws InterruptedException {

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.submit(() -> {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt(); // Preserve interrupt status

}

}

});

executor.submit(() -> {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " - Count: " + (i + 5));

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt(); // Preserve interrupt status

}

}

});

executor.shutdown(); // Prevent new tasks from being submitted

executor.awaitTermination(1, TimeUnit.MINUTES); // Wait for tasks to finish

System.out.println("All tasks are finished.");

}

}

Unit 5

* **i/o basic**

Input/Output (I/O) is a crucial mechanism for interacting with the user or external systems like files, databases, or networks. Java provides a standard I/O library under the java.io package that supports input and output streams, reading/writing files, and handling standard input/output.

**Types of I/O in Java:**

1. **Byte Streams**:
   * Used for handling I/O of raw binary data.
   * Classes: InputStream, OutputStream.
   * Example: FileInputStream, FileOutputStream.
2. **Character Streams**:
   * Used for handling I/O of characters (textual data).
   * Classes: Reader, Writer.
   * Example: FileReader, FileWriter.
3. **Buffered Streams**:
   * Provide buffering to enhance I/O efficiency.
   * Classes: BufferedReader, BufferedWriter.
4. **Standard I/O**:
   * Input from the keyboard: System.in.
   * Output to the console: System.out.
5. **Data Streams**:
   * Used for reading and writing primitive data types.
   * Classes: DataInputStream, DataOutputStream.

**Working of I/O in Java:**

1. **Input**:
   * Data flows from the source (keyboard, file, network) into the program.
   * Scanner class or BufferedReader is often used for console input.
2. **Output**:
   * Data flows from the program to a destination (console, file, network).
   * System.out.print and FileWriter are commonly used for output.

**Advantages of Java I/O:**

1. **Platform Independence**: Works across different platforms (Windows, Linux, etc.).
2. **Built-in Libraries**: Rich set of libraries for file handling, networking, and more.
3. **Efficient**: Buffered I/O classes improve performance.
4. **Supports Unicode**: Character streams support reading and writing Unicode characters.

**Disadvantages of Java I/O:**

1. **Complexity**: Requires understanding of various streams and their combinations.
2. **Limited Flexibility**: Java's I/O streams are synchronous and blocking, limiting flexibility in some advanced applications (e.g., non-blocking I/O).
3. **Verbose Syntax**: The code can be lengthy and harder to manage for beginners compared to other languages.

**Common Usage:**

1. **File Handling**: Reading from and writing to files.
2. **Console Input/Output**: Interaction with the user via the terminal.
3. **Data Serialization**: Saving objects into files or reading them back.
4. **Network Programming**: Sending and receiving data over a network.

**Sample Code:**

1. **Reading from Console Using Scanner:**

import java.util.Scanner;

public class ConsoleInputExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");

scanner.close();

}

}

2. **Reading and Writing Files:**

import java.io.\*;

public class FileIOExample {

public static void main(String[] args) {

// Writing to a file

try (FileWriter writer = new FileWriter("output.txt")) {

writer.write("Hello, this is a sample file.");

} catch (IOException e) {

e.printStackTrace();

}

// Reading from a file

try (BufferedReader reader = new BufferedReader(new FileReader("output.txt"))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

* **reading console input**

reading input from the console is typically done using the standard input stream, System.in. Java provides several classes for handling console input efficiently, such as Scanner, BufferedReader, and older approaches like using InputStreamReader.

**Methods to Read Console Input in Java:**

**1. Using Scanner Class (Java 5 and above)**

The Scanner class is one of the easiest and most commonly used ways to read user input from the console. It provides methods to read different data types like int, double, float, String, etc.

**Key Methods of Scanner**:

* nextLine(): Reads the entire line as a String.
* nextInt(): Reads an integer.
* nextDouble(): Reads a double.
* nextBoolean(): Reads a boolean.
* next(): Reads a single word (until whitespace).

**Example: Using Scanner to Read Input**

import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

// Create a Scanner object for reading console input

Scanner scanner = new Scanner(System.in);

// Reading different types of inputs

System.out.print("Enter your name: ");

String name = scanner.nextLine(); // Reads the entire line of input

System.out.print("Enter your age: ");

int age = scanner.nextInt(); // Reads an integer value

System.out.print("Enter your salary: ");

double salary = scanner.nextDouble(); // Reads a double value

// Output the data

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

// Close the Scanner to prevent resource leaks

scanner.close();

}

}

**Advantages**:

* Easy to use and supports multiple data types.
* Convenient methods like nextInt(), nextDouble(), etc., for parsing primitive data types.

**Disadvantages**:

* When mixing nextLine() with other input methods (nextInt(), nextDouble()), there can be input buffer issues. A common issue is that after calling nextInt(), if you try to call nextLine(), it will return an empty string. This is because nextInt() leaves a newline character in the input buffer.

**Solution to Buffer Issue:**

System.out.print("Enter your age: ");

int age = scanner.nextInt();

scanner.nextLine(); // Clear the buffer

**2. Using BufferedReader and InputStreamReader (Pre-Java 5)**

Before Scanner was introduced, BufferedReader was commonly used to read input. It allows you to read text from the console efficiently by buffering the input, which means reading in larger chunks rather than one character at a time.

**Steps**:

1. Wrap System.in inside an InputStreamReader (converts byte stream to character stream).
2. Wrap InputStreamReader inside a BufferedReader (buffers the input for efficiency).

**Example: Using BufferedReader to Read Input**

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class BufferedReaderExample {

public static void main(String[] args) {

// Wrap System.in in an InputStreamReader and then in a BufferedReader

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

try {

// Reading string input

System.out.print("Enter your name: ");

String name = reader.readLine(); // Reads the entire line

// Reading integer input

System.out.print("Enter your age: ");

int age = Integer.parseInt(reader.readLine()); // Convert string to integer

// Reading double input

System.out.print("Enter your salary: ");

double salary = Double.parseDouble(reader.readLine()); // Convert string to double

// Output the data

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Advantages**:

* Efficient for large input due to buffering.
* Provides more control over input types, as you can explicitly convert String to the desired type.

**Disadvantages**:

* Requires handling exceptions, such as IOException.
* More verbose compared to Scanner for simple tasks.

**3. Using Console Class (Java 6 and above)**

The Console class can be used to read input, but it is more suited for command-line applications. It's not available when running Java programs in some environments (e.g., IDEs like Eclipse or IntelliJ). It is commonly used for secure input, such as passwords.

**Key Methods of Console**:

* readLine(): Reads a line of text.
* readPassword(): Reads a password without echoing the characters to the console.

**Example: Using Console to Read Input**

import java.io.Console;

public class ConsoleExample {

public static void main(String[] args) {

// Get the system console

Console console = System.console();

if (console != null) {

// Reading string input

String name = console.readLine("Enter your name: ");

// Reading password input (will not echo characters)

char[] password = console.readPassword("Enter your password: ");

// Output the data

System.out.println("Name: " + name);

System.out.println("Password: " + new String(password));

} else {

System.out.println("No console available.");

}

}

}

**Advantages**:

* Secure input for sensitive data (e.g., passwords).
* Simple and easy to use for reading strings and passwords.

**Disadvantages**:

* Not available in all environments (e.g., IDEs).
* Lacks flexibility compared to Scanner or BufferedReader.

**Summary of Methods**

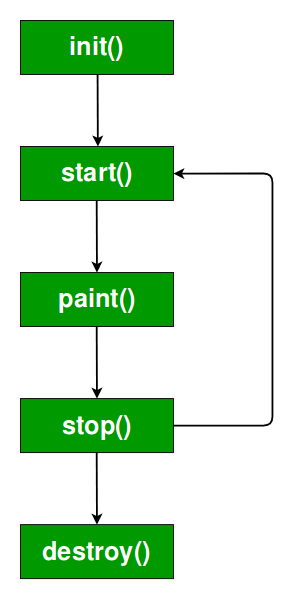
| **Met method** | **Bes best for** | **Key key Methods** | **Ad advantages** | **Dis disadvantages** |
| --- | --- | --- | --- | --- |
| Scanner | Simple, basic inputs  (post Java 5) | nextLine(),  nextInt(),  etc. | Easy to use, supports  multiple data types | Buffering issues with  mixed types |
| Buffered  Reader | Efficient input  (pre-Java 5) | readLine() | Efficient for large inputs,  more control | Verbose, exception  handling required |
| Console | Command-line  Inputs | readLine(),  readPassword() | Secure password entry,  simple input handling | Not supported in all  environments |

* applets

An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.  
Applets are used to make the website more dynamic and entertaining.

**Important points :**

1. All applets are sub-classes (either directly or indirectly) of *[java.applet.Applet](https://docs.oracle.com/javase/7/docs/api/java/applet/Applet.html" \t "_blank)* class.
2. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
3. In general, execution of an applet does not begin at main() method.
4. Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

**Life cycle of an applet :**

* 1. **init( ) :**The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
  2. **start( ) :**The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )**is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

**3. paint( ) :**The **paint( )** method is called each time an AWT-based applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.   
**paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )**is called.   
The **paint( )** method has one parameter of type [Graphics](https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html). This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

**4. stop( ) :**The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don’t need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

**5. destroy( ) :** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

**Features of Applets over HTML**

* Displaying dynamic web pages of a web application.
* Playing sound files.
* Displaying documents
* Playing animations

**Restrictions imposed on Java applets**

Due to security reasons, the following restrictions are imposed on Java applets:

1. An applet cannot load libraries or define native methods.
2. An applet cannot ordinarily read or write files on the execution host.
3. An applet cannot read certain system properties.
4. An applet cannot make network connections except to the host that it came from.
5. An applet cannot start any program on the host that’s executing it.

Sample code :

**import** java.applet.Applet;

**import** java.awt.Graphics;

// HelloWorld class extends Applet

**public** **class** HelloWorld **extends** Applet

{

    // Overriding paint() method

    @Override

**public** **void** paint(Graphics g)

    {

        g.drawString("Hello World", 20, 20);

    }

}

Advantage of Applet

There are many advantages of applet. They are as follows:

* It works at client side so less response time.
* Secured
* It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

* Plugin is required at client browser to execute applet.

**Applet Architecture in Java:**

1. **Applet Class (java.applet.Applet)**: Every Java applet extends the java.applet.Applet class, inheriting basic functionality for browser-based applications.
2. **Lifecycle Methods**: Applets have a lifecycle defined by specific methods that manage their creation, execution, and termination:
   * init(): Called once when the applet is first loaded.
   * start(): Called each time the applet is started or restarted.
   * paint(Graphics g): Draws the applet’s content.
   * stop(): Called when the applet is no longer active but not yet destroyed.
   * destroy(): Called when the applet is being removed from memory.
3. **Applet Context**: An applet runs within a "context" provided by the browser or applet viewer. This includes interacting with the environment, managing the user interface, or handling communication.
4. **Security Restrictions**: Applets have limited access to the local system (e.g., file system) due to security concerns, preventing malicious behavior.
5. **User Interface**: Applets can have graphical interfaces, buttons, forms, etc., using AWT (Abstract Window Toolkit).

* the html applet tag

The <applet> tag in HTML is used to embed a Java applet within a webpage. However, the <applet> tag has been deprecated since HTML 4.01, and browsers no longer support it. Java applets were primarily used in the past to run small Java programs inside a browser. Here's an example of how the <applet> tag was used:

<html>

<body>

<applet code="MyApplet.class" width="300" height="300">

Your browser does not support Java Applets.

</applet>

</body>

</html>

**Attributes of <applet> Tag:**

* **code**: Specifies the Java class file that contains the applet (e.g., MyApplet.class).
* **width**: Defines the width of the applet.
* **height**: Defines the height of the applet.
* **archive** (optional): Specifies the location of the .jar file that contains the applet.
* passing parameters to applets

**Steps to Pass Parameters to an Applet:**

1. **Define Parameters in HTML:** Use the <param> tag inside the <applet> tag to pass parameters.
2. **Retrieve Parameters in Java:** Inside the applet class, use the getParameter() method to access the passed values.

HTML Code to Pass Parameters:

<html>

<body>

<applet code="MyApplet.class" width="300" height="300">

<param name="username" value="JohnDoe">

<param name="age" value="25">

</applet>

</body>

</html>

Java Code to Retrieve Parameters:

import java.applet.Applet;

import java.awt.Graphics;

public class MyApplet extends Applet {

String username;

int age;

// Called when the applet is first loaded

public void init() {

// Retrieve parameters

username = getParameter("username");

String ageParam = getParameter("age");

age = Integer.parseInt(ageParam);

}

// Paint method to display parameter values

public void paint(Graphics g) {

g.drawString("Username: " + username, 20, 20);

g.drawString("Age: " + age, 20, 40);

}

}

* networking basic

Java provides a robust set of libraries and classes for networking, enabling applications to communicate over the internet. The core package for networking in Java is java.net.

* **IP Address**: A unique identifier for devices on a network.
* **Port**: A logical endpoint for network communication, used to differentiate services on a device.
* **Socket**: An endpoint for two-way communication between two machines over a network.
* **Server-Client Model**: A common model where one machine (server) listens for requests and another (client) sends requests.

**Server Side**:

* The ServerSocket listens for incoming connections on a specific port (8080 in this case).
* When a client connects, it creates a Socket to communicate.
* The server reads the client’s message and sends a response back.

**Client Side**:

* The Socket connects to the server’s IP address (localhost) and port.
* The client sends a message to the server and reads the response.
* java and the net

Java provides extensive support for networking through the java.net package, which allows developers to build programs that communicate over a network. Java networking encompasses various features and tools that make it easier to establish communication between computers, handle URLs, and interact with web services.

**Key Networking Features in Java:**

* **Platform Independence**: Java programs can run on any system with a JVM, which is ideal for building cross-platform network applications.
* **Built-in Libraries**: Java includes the java.net package, offering classes and interfaces to work with low-level and high-level networking.
* **Support for TCP/IP**: Java supports the widely-used TCP/IP protocols for communication over the network.
* **Multithreading**: Java's multithreading capabilities allow you to handle multiple connections simultaneously, making it well-suited for server-client architectures.

**2. Important Classes in Java Networking:**

* **Socket**: Represents a client-side connection, used for communicating between two machines.
* **ServerSocket**: Used on the server side to listen for incoming client connections.
* **InetAddress**: Represents an IP address or hostname and provides functionality for DNS lookups.
* **URL and URLConnection**: These classes allow you to work with web resources via HTTP or FTP.

**3. Working with URLs in Java:**

import java.net.\*;

import java.io.\*;

public class URLExample {

public static void main(String[] args) {

try {

// Create a URL object

URL url = new URL("http://www.example.com");

// Open a connection to the URL

URLConnection urlConnection = url.openConnection();

// Get the input stream and read data from the URL

BufferedReader in = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println(inputLine);

}

in.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

**4. Networking Protocols in Java:**

Java supports different networking protocols, including:

* **TCP (Transmission Control Protocol)**: A reliable, connection-oriented protocol used for applications that require guaranteed delivery of data (e.g., web browsing, email).
* **UDP (User Datagram Protocol)**: A connectionless, faster protocol used when speed is more important than reliability (e.g., video streaming, online gaming).

**TCP Example (Client-Server):**

* **Client** and **Server** communicate over TCP, as shown in the earlier example with Socket and ServerSocket.

**UDP Example (DatagramSocket):**

For applications that use UDP, Java provides the DatagramSocket class.

**5. Networking Concepts in Java:**

* **Sockets**: Sockets are the endpoints of a two-way communication link between two programs running on the network.
* **Ports**: A port is a logical address within the host machine that differentiates between multiple network services.
* **Protocols**: Java supports both high-level and low-level protocols (like HTTP, FTP, TCP, UDP).

**Security in Java Networking:**

Java places a strong emphasis on security, especially for network applications. Applets and applications have security restrictions, and developers can use tools like:

* **SecurityManager**: Controls network access.
* **SSL/TLS Support**: Java provides built-in support for secure communication over the network using the javax.net.ssl package.
* tcp/ip client socket url

In Java networking, **TCP/IP**, **Socket**, and **URL** are key components that enable communication between computers over a network. Here's a breakdown of each:

**1. TCP/IP (Transmission Control Protocol/Internet Protocol):**

* **TCP/IP** is the foundation of internet communication. TCP is a reliable, connection-oriented protocol that ensures data delivery, while IP handles the addressing and routing of data packets across the network.
* **TCP** establishes a connection between the client and server before data is transferred, making sure it arrives in the correct order and without errors.

**2. Client-Server Communication:**

* **Client**: The machine or program that initiates communication by sending a request to a server.
* **Server**: The machine or program that listens for and responds to requests from clients.

**3. Sockets in Java:**

A **Socket** is an endpoint for communication between two machines. In Java, Socket is used to establish a TCP connection between a client and a server.

**Client Socket:**

* A **client socket** is used on the client-side to connect to the server. The client initiates the connection to the server using the server's IP address and port number.
* Java provides the Socket class for creating a client socket.

**Client-Server Example Using TCP/IP:**

Here’s how you create a simple TCP/IP client-server program using sockets in Java.

**Server Code (Using ServerSocket)**:

import java.net.\*;

import java.io.\*;

public class SimpleServer {

public static void main(String[] args) {

try {

// Create a server socket on port 8080

ServerSocket serverSocket = new ServerSocket(8080);

System.out.println("Server is running and waiting for client...");

// Wait for a client to connect

Socket clientSocket = serverSocket.accept();

System.out.println("Client connected!");

// Create input and output streams for communication

BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

// Read message from the client and send a response

String clientMessage = in.readLine();

System.out.println("Received from client: " + clientMessage);

out.println("Hello from the server!");

// Close the connection

clientSocket.close();

serverSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Client Code (Using Socket)**:

import java.net.\*;

import java.io.\*;

public class SimpleClient {

public static void main(String[] args) {

try {

// Create a client socket and connect to the server at localhost on port 8080

Socket socket = new Socket("localhost", 8080);

System.out.println("Connected to the server");

// Create input and output streams for communication

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

// Send a message to the server

out.println("Hello from the client!");

// Receive a response from the server

String serverMessage = in.readLine();

System.out.println("Received from server: " + serverMessage);

// Close the connection

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

you can use **TCP/IP Sockets** to establish a connection between a client and a server, and the **URL** class for interacting with web-based resources over HTTP or other protocols.

**Difference Between Socket and URL:**

* **Socket**: Represents a lower-level connection for two-way communication between a client and server using the TCP/IP protocol.
* **URL**: A higher-level abstraction that represents a web resource and is typically used for HTTP communication.
* url connections

In Java, the URLConnection class is part of the java.net package and is used to represent a connection to a remote resource identified by a URL. It provides methods to communicate with the resource, allowing you to read from and write to web resources such as web pages or APIs.

**Key Steps to Work with URLConnection:**

1. **Create a URL object**: The URL represents the remote resource you want to connect to.
2. **Open a connection using openConnection()**: This method returns a URLConnection object, which represents the connection to the specified resource.
3. **Read or Write Data**: You can read from or write to the connection depending on the resource (e.g., a web page or file).
4. **Close the connection**: Properly closing streams is necessary to free resources.

Example: Reading Data from a Webpage using URLConnection

import java.net.\*;

import java.io.\*;

public class URLConnectionExample {

public static void main(String[] args) {

try {

// Create a URL object pointing to the desired webpage

URL url = new URL("http://www.example.com");

// Open a connection to the URL

URLConnection urlConnection = url.openConnection();

// Create an input stream to read data from the URL

BufferedReader in = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));

// Read and print the webpage content

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println(inputLine);

}

// Close the stream

in.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Writing Data to a URL (e.g., POST request):

import java.net.\*;

import java.io.\*;

public class URLConnectionPostExample {

public static void main(String[] args) {

try {

// URL to send POST request to

URL url = new URL("http://www.example.com/submit");

// Open a connection

URLConnection urlConnection = url.openConnection();

urlConnection.setDoOutput(true); // Allow output to the URL

// Send data via POST

OutputStreamWriter out = new OutputStreamWriter(urlConnection.getOutputStream());

out.write("name=JohnDoe&age=25"); // Sending data in key-value pairs

out.flush();

out.close();

// Read the response from the server

BufferedReader in = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println(inputLine);

}

in.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Methods of URLConnection:**

* **openConnection()**: Opens the connection to the specified URL.
* **getInputStream()**: Returns an input stream to read data from the URL.
* **getOutputStream()**: Returns an output stream to write data to the URL (used for POST requests).
* **setDoOutput(boolean flag)**: Enables or disables output capability (required for sending data).
* **setRequestProperty(String key, String value)**: Sets request headers like content type, user-agent, etc.
* **getHeaderField(String name)**: Retrieves a specific header field from the response.
* tcp/ip server sockets

A **TCP/IP server socket** allows a server to listen for incoming client connections and establish a communication link between the server and the client. In Java, the ServerSocket class is used to implement server-side sockets that handle incoming TCP/IP requests.

**How Server Sockets Work:**

1. **Server Creation**: The server creates a ServerSocket bound to a specific port.
2. **Listening for Connections**: The server listens for incoming connections from clients.
3. **Accepting Connections**: When a client tries to connect, the server accepts the connection and establishes a socket for communication.
4. **Data Exchange**: Both client and server use input/output streams to send and receive data over the network.
5. **Closing Connections**: After communication, the socket is closed to free up resources.

**How TCP/IP Server Sockets Work in Java:**

* The server listens for incoming connections on a specific port.
* When a client connects, the server accepts the connection, and a Socket is created for two-way communication.
* Communication happens via input and output streams (sending and receiving data).
* After the communication is done, both the server and client close their respective sockets.

**Key Methods in ServerSocket:**

* **accept()**: Listens for a connection request from a client and creates a socket to handle the connection.
* **close()**: Closes the server socket to free resources.
* database connectivity

In Java, **JDBC** (Java Database Connectivity) is an API that allows Java applications to connect to relational databases, execute queries, and retrieve results. JDBC is a core part of Java's platform, enabling interaction with databases in a consistent manner.

**Steps for Database Connectivity:**

1. **Load the JDBC Driver**: The driver is required to communicate with the database. For example, for MySQL, you use the MySQL JDBC driver.
2. **Establish a Connection**: Connect to the database using a connection string (JDBC URL), username, and password.
3. **Create a Statement**: A Statement object is used to execute SQL queries.
4. **Execute SQL Queries**: Queries like SELECT, INSERT, UPDATE, and DELETE are executed using the statement object.
5. **Process the Results**: Results from a SELECT query are stored in a ResultSet object.
6. **Close the Connection**: After operations are completed, the connection should be closed to free up resources.

**Example: Basic Database Connectivity Using JDBC**

In this example, we will connect to a MySQL database and perform basic operations like querying data.

1. Loading the JDBC Driver

try {

// Load the MySQL JDBC driver

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

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

1. Establishing a Connection

import java.sql.\*;

public class DatabaseExample {

public static void main(String[] args) {

// JDBC URL for MySQL database

String url = "jdbc:mysql://localhost:3306/mydatabase"; // Replace with your DB name

String username = "root"; // Replace with your DB username

String password = "password"; // Replace with your DB password

try {

// Establish a connection to the database

Connection connection = DriverManager.getConnection(url, username, password);

System.out.println("Connected to the database");

// Always close the connection when done

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

1. Creating and Executing a Statement

import java.sql.\*;

public class DatabaseExample {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/mydatabase";

String username = "root";

String password = "password";

try {

// Establish connection

Connection connection = DriverManager.getConnection(url, username, password);

// Create a statement object to execute SQL queries

Statement statement = connection.createStatement();

// Execute a SQL query

String sql = "SELECT \* FROM employees"; // Replace with your table

ResultSet resultSet = statement.executeQuery(sql);

// Process the result set

while (resultSet.next()) {

// Example: Print data from 'employees' table

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

String position = resultSet.getString("position");

System.out.println("ID: " + id + ", Name: " + name + ", Position: " + position);

}

// Close the statement and connection

statement.close();

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

1. Inserting Data into the Database

import java.sql.\*;

public class InsertDataExample {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/mydatabase";

String username = "root";

String password = "password";

try {

// Establish connection

Connection connection = DriverManager.getConnection(url, username, password);

// Create a statement object to execute SQL queries

Statement statement = connection.createStatement();

// Execute an INSERT query

String sql = "INSERT INTO employees (name, position) VALUES ('John Doe', 'Manager')";

int rowsAffected = statement.executeUpdate(sql);

System.out.println("Rows inserted: " + rowsAffected);

// Close the statement and connection

statement.close();

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Important Classes in JDBC:**

1. **DriverManager**:
   * Manages the set of JDBC drivers. Establishes a connection to a database using getConnection().
2. **Connection**:
   * Represents a connection to the database. Created using DriverManager.getConnection().
3. **Statement**:
   * Used to execute SQL queries. Created from a Connection object.
4. **PreparedStatement**:
   * A subclass of Statement used to execute parameterized SQL queries. Helps prevent SQL injection.
5. **ResultSet**:
   * A table of data representing the result of a query. You can loop through rows in the ResultSet to process the data.

**JDBC Driver Dependencies:**

To use JDBC with a specific database (e.g., MySQL, PostgreSQL, Oracle), you must add the corresponding JDBC driver to your project. This driver is often available as a .jar file or via Maven/Gradle dependencies.