

**PROJECT REPORT**

**JAVA PROGRAMMING FUNDAMENTALS**

**(EBDS22ET2)**

2024-2025(EVEN SEMESTER)

**DEPARTMENT OF B.Tech DS & AI (E&T)**

**COURSE :** B-TECH CSE-DS (AI)

**YEAR/SEM/SEC : 1st/ llnd /BB1**

**PROJECT TITLE : LIFE cycle of threads**



**BONAFIDE CERTIFICATE**

**JAVA PROGRAMMING FUNDAMENTALS**

**DEPARTMENT OF B.Tech DS & AI (E&T)**

Certified that this project report **“LIFE CYCLE OF THREADS ”** is confirmed work of **NAME**  I-year B-Tech CSE- DS(AI) in **JAVA PROGRAMMING FUNDAMENTALS (EBDS22ET2)** who carried out the project work under the supervision

Signature of Lab-in-Charge Signature of Head of Dept

Submitted for the Practical Examination held on

Internal Examiner External Examiner

# ABSTRACT

.

Multithreading is an essential feature in Java that allows multiple threads to run concurrently, enabling the development of high-performance and responsive applications. Understanding the life cycle of a thread is crucial for managing thread behavior, avoiding concurrency issues, and optimizing resource utilization in Java programs.

A thread in Java passes through several defined stages during its lifetime. These stages are: New, Runnable, Running, Blocked/Waiting, Timed Waiting, and Terminated. The cycle begins when a thread object is created using the Thread class or by implementing the Runnable interface, entering the New state. Once the start() method is called, the thread moves to the Runnable state, indicating that it is ready to run and is waiting for the CPU to allocate processing time. When the thread is selected by the thread scheduler, it transitions to the Running state, where the thread’s run() method executes.

During execution, a thread may enter the Blocked, Waiting, or Timed Waiting states depending on specific conditions such as synchronization or waiting for another thread to complete a task. For instance, methods like sleep(), join(), and wait() move threads into timed or indefinite waiting states. After completing its task or if it is forcefully stopped, a thread enters the Terminated state, marking the end of its life cycle.

Managing the thread life cycle effectively ensures proper synchronization, minimizes the risk of deadlocks, and improves system performance. Java provides robust tools and APIs to control and monitor threads, making it easier for developers to implement efficient multithreaded applications. A clear understanding of the thread life cycle is therefore vital for any Java developer working with concurrent programming

**Introduction to Java:**

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle). It is designed to be platform-independent using the Java Virtual Machine (JVM), which allows Java programs to run on any device that supports the JVM—following the principle of "Write Once, Run Anywhere."

**Key features of Java include:**

Object-Oriented Programming (OOP)

Platform Independence

Automatic Memory Management (Garbage Collection)

Robust Exception Handling

Built-in Multithreading

One of Java’s most powerful features is multithreading, which allows concurrent execution of two or more threads. This enables efficient CPU utilization and responsive applications, especially in real-time and GUI-based software

**Aim:**

**To demonstrate the life cycle of a thread in Java using a sample Java program, which shows transitions between different thread states such as New, Runnable, Running, Timed Waiting, and Terminated.**

**Algorithm:**

**1. Start the program.**

**2. Define a class TicketBooking that implements Runnable. In the run() method, make the thread sleep for 200 ms (Thread.sleep(200)) to simulate Timed Waiting, print the state of the main thread, then sleep again for 100 ms.**

**3. Define a class TicketSystem that implements Runnable. Declare two static variables: mainThread and ticketSystem.**

**4. In the run() method of TicketSystem, create an object of TicketBooking, then create a thread bookingThread using it. Print the initial state (NEW) of bookingThread. Call bookingThread.start() to move it to the RUNNABLE state. Pause mainThread for 100 ms to allow bookingThread to enter TIMED\_WAITING. Print the current state of bookingThread. Then call bookingThread.join() to move mainThread to WAITING state. After completion, print the final state of bookingThread (TERMINATED).**

**5. In the main() method, create an instance of TicketSystem. Create a thread mainThread using that instance. Print its state (NEW), start it using start() to move it to RUNNABLE.**

**6. End the program.**

## Source Code:

**// Java program to demonstrate thread states using a ticket booking scenario**

**class TicketBooking implements Runnable {**

**@Override**

**public void run() {**

**try {**

**// Thread goes into Timed Waiting state**

**Thread.sleep(200);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**// Display state of mainThread while it is waiting**

**System.out.println("State of mainThread while bookingThread is sleeping: " +**

**TicketSystem.mainThread.getState());**

**try {**

**// Another Timed Waiting state**

**Thread.sleep(100);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**public class TicketSystem implements Runnable {**

**// References to the main thread and system object**

**public static Thread mainThread;**

**public static TicketSystem ticketSystem;**

**@Override**

**public void run() {**

**TicketBooking booking = new TicketBooking();**

**// Creating a new thread for booking**

**Thread bookingThread = new Thread(booking);**

**// 1. State: NEW**

**System.out.println("State after creating bookingThread: " + bookingThread.getState());**

**// 2. Start the thread -> State: RUNNABLE**

**bookingThread.start();**

**System.out.println("State after starting bookingThread: " + bookingThread.getState());**

**try {**

**// Delay to allow bookingThread to enter Timed Waiting**

**Thread.sleep(100);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**// 3. Likely state: TIMED\_WAITING**

**System.out.println("State after sleeping bookingThread: " + bookingThread.getState());**

**try {**

**// Main thread waits for bookingThread to finish**

**bookingThread.join();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**// 4. State: TERMINATED**

**System.out.println("State after bookingThread finishes: " + bookingThread.getState());**

**}**

**public static void main(String[] args) {**

**ticketSystem = new TicketSystem();**

**mainThread = new Thread(ticketSystem);**

**// 1. State: NEW**

**System.out.println("State after creating mainThread: " + mainThread.getState());**

**// 2. Start the main thread -> State: RUNNABLE**

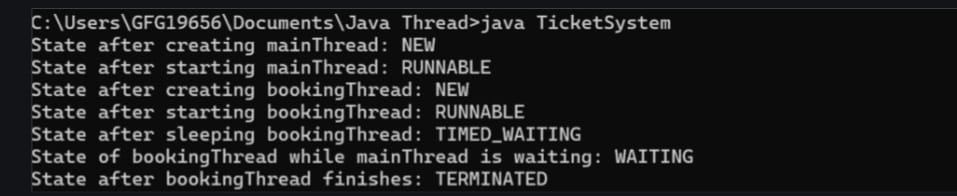
**mainThread.start();**

**System.out.println("State after starting mainThread: " + mainThread.getState());**

**}**

**}**

**output:**

****

**RESULT:**

**The life cycle of a thread in Java includes five main states: New, Runnable, Running, Waiting, and Terminated.**

**We created and observed a thread transitioning between these states during execution.**

**Methods like start(), sleep(), and join() help demonstrate these states.**

**The experiment proved how Java handles multithreading efficiently.**

**It enhanced understanding of thread management and process synchronization.**