Monitor and Synchronization

1. Introduction

In multithreaded programming, multiple threads execute concurrently and may access shared resources. Without proper control, this can lead to **race conditions**, data inconsistency, and unpredictable behavior. Java provides a built-in high-level synchronization mechanism called a **monitor** to handle these issues safely.

---

2. What is a Monitor?

A **monitor** is a synchronization construct that:

- Ensures **mutual exclusion** (only one thread executes a critical section at a time)
- Supports **thread coordination** using wait(), notify(), and notifyAll()

Note: In Java, **every object has an intrinsic monitor (lock)**.

---

3. Monitor in Java

Java implements monitors using the synchronized keyword.

Key Properties

- Each object has **one monitor**
- A thread must **acquire the monitor lock** before entering a synchronized section
- If the lock is held by another thread, the current thread enters the **BLOCKED** state

---

4. Critical Section

A **critical section** is a block of code that accesses shared resources and must not be executed by more than one thread at the same time.

Example:

```
synchronized(this) {
// critical section
}
```

---

5. Types of Synchronization

5.1 Synchronized Method

```
synchronized void display() {
// critical section
}
```

- Entire method is synchronized
- Lock is on the current object (this)

5.2 Synchronized Block

```
void display() {
synchronized(this) {
// critical section
}
}
```

- Synchronizes only required code
- More efficient and flexible

---

6. Monitor Working Mechanism

1. Thread requests entry to synchronized code
2. JVM checks monitor availability
3. If free → lock is acquired
4. If busy → thread is blocked
5. After execution → lock is released

---

7. Inter-Thread Communication Using Monitor

Monitors also provide coordination between threads.

Methods Used

- wait() – releases lock and waits

- notify() – wakes up one waiting thread
- notifyAll() – wakes up all waiting threads

Note: These methods **must be called inside synchronized context**.

---

8. Example: Monitor with wait and notify

```
class Shared {
    synchronized void method() {
        try {
            wait();
        } catch (InterruptedException e) {}
        notify();
    }
}
```

---

9. Monitor States of Thread

- **RUNNABLE** – ready to run
- **BLOCKED** – waiting for monitor lock
- **WAITING** – waiting after calling wait()
- **TERMINATED** – execution finished

---

10. Advantages of Monitor

- Automatic locking and unlocking
- Prevents race conditions
- Easy to use and maintain
- Object-oriented synchronization

---

13. Conclusion

A monitor is a powerful synchronization mechanism provided by Java that ensures safe and controlled access to shared resources. By combining mutual exclusion and inter-thread communication, monitors form the backbone of Java’s multithreading model.