



## Multithreading

### 1) What is multithreading?

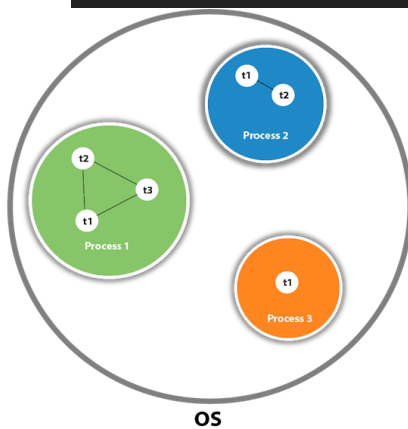
Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking. It consumes less memory and gives the fast and efficient performance. Its main advantages are:

- Threads share the same address space.
- The thread is lightweight.
- The cost of communication between the processes is low.

### 2) What is the thread?

A thread is a lightweight subprocess. It is a separate path of execution because each thread runs in a different stack frame. A process may contain multiple threads. Threads share the process resources, but still, they execute independently.

Aspect	Process	Thread
<b>Definition</b>	A program in execution.	A smaller unit of a process that can run concurrently.
<b>Independence</b>	Processes are independent of each other.	Threads are dependent on the process they belong to.
<b>Memory</b>	Each process has its own memory space.	Threads share the same memory space within a process.
<b>Context Switching</b>	Slower, more expensive context switching.	Faster and cheaper context switching.
<b>Communication</b>	Inter-process communication (IPC) is slower and more expensive.	Inter-thread communication is faster and easier.
<b>Impact of Changes</b>	Changes in a parent process do not affect child processes.	Changes in a parent thread can affect child threads.



#### Scenario for inter-thread communication and example

- **Producer (Deposit):** One thread will deposit money into the account.
- **Consumer (Withdraw):** Another thread will withdraw money from the account, but it will wait if the balance is insufficient (i.e., it will wait for the deposit to occur before it can withdraw).

#### 4) What do you understand by inter-thread communication?

- Inter-thread communication in Java allows threads to communicate with each other using the `wait()`, `notify()`, and `notifyAll()` methods. These methods are used for synchronizing threads so that one thread can wait for another to complete before it proceeds. The communication generally involves two threads: a producer thread and a consumer thread. It can be obtained by `wait()`, `notify()`, and `notifyAll()` methods.

```
class BankAccount {
    private int balance = 0; // Bank account balance
    private boolean isBalanceAvailable = false; // Flag to check if balance is available for withdrawal

    // Method to deposit money
    synchronized void deposit(int amount) {
        while (isBalanceAvailable) {
            try {
                wait(); // Wait if balance is already available for withdrawal
            } catch (InterruptedException e) {}
        }
        balance += amount; // Deposit money into the account
        System.out.println("Deposited: " + amount + ", New Balance: " + balance); // Log the deposit
        isBalanceAvailable = true; // Mark balance as available for withdrawal
        notify(); // Notify the withdrawal thread
    }

    // Method to withdraw money
    synchronized void withdraw(int amount) {
        while (!isBalanceAvailable || balance < amount) {
            try {
                wait(); // Wait if no balance or insufficient balance
            } catch (InterruptedException e) {}
            isBalanceAvailable = false; // Mark the balance as processed (can't withdraw)
            notify(); // Notify the deposit thread
        }
    }
}

public class BankAccountSimulation {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(); // Create the bank account

        // Producer thread: Deposits money into the account
        new Thread(() -> {
            account.deposit(1000); // Deposit 1000
            try { Thread.sleep(500); } catch (InterruptedException e) {}
            account.deposit(2000); // Deposit 2000
        }).start();

        // Consumer thread: Withdraws money from the account
        new Thread(() -> {
            account.withdraw(500); // Withdraw 500
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
            account.withdraw(1500); // Withdraw 1500
        }).start();
    }
}
```

---

6) Why must wait() method be called from the synchronized block?

We must call the wait method otherwise it will throw **java.lang.IllegalMonitorStateException** exception. Moreover, we need wait() method for inter-thread communication with notify() and notifyAll(). Therefore It must be present in the synchronized block for the proper and correct communication.

---

7) What are the advantages of multithreading?

### Advantages of Multithreading

1. **Improved Responsiveness:** Keeps applications reactive even when performing background tasks.
2. **Faster Execution:** Threads run independently, speeding up task completion.
3. **Efficient Resource Use:** Threads share memory, optimizing cache usage.
4. **Cost-Effective:** Reduces server needs as one server can handle multiple threads simultaneously.

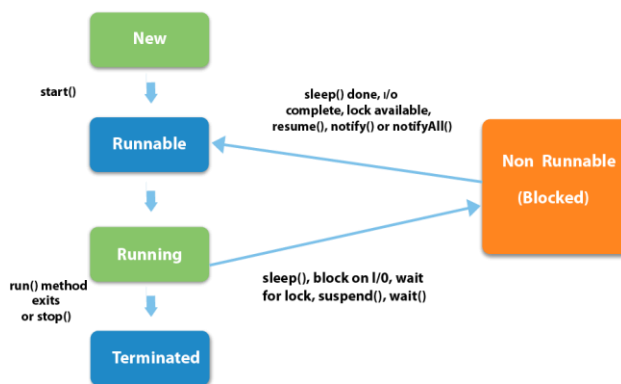
M

---

## Thread Lifecycle States

A thread in Java goes through the following states during its lifecycle:

1. **New:** The thread is created but not yet started.
2. **Runnable:** The thread is ready to run and waiting for the CPU to execute.
3. **Running:** The thread is executing its task.
4. **Blocked/Waiting:** The thread is waiting for some resource or signal to proceed.
5. **Terminated:** The thread has completed execution.



### Real-World Example Scenario:

A food delivery system with a thread representing a delivery driver:

1. **New:** The driver is assigned a new delivery task.
2. **Runnable:** The driver is ready to start the delivery.
3. **Running:** The driver is actively delivering the food.
4. **Blocked/Waiting:** The driver is waiting at a traffic light or restaurant to pick up food.
5. **Terminated:** The delivery is complete, and the driver is done.

```
// Represents a delivery driver's thread lifecycle in a food delivery app
class DeliveryDriver extends Thread {
    public void run() {
        System.out.println("Running: Delivering food.");
        try {
            System.out.println("Waiting: Stopped at traffic light.");
            Thread.sleep(2000); // Simulate waiting
        } catch (InterruptedException e) {
            System.out.println("Delivery interrupted.");
        }
        System.out.println("Terminated: Delivery completed.");
    }

    public static void main(String[] args) {
        // New state
        DeliveryDriver driver = new DeliveryDriver();
        System.out.println("New: Delivery task assigned.");

        // Runnable state
        driver.start(); // Moves thread to Runnable and then Running
    }
}
```

11) Differentiate between the Thread class and Runnable interface for creating a Thread?

The Thread can be created by using two ways.

- By extending the Thread class
- By implementing the Runnable interface

Aspect	Thread Class	Runnable Interface
Inheritance	Extends the <code>Thread</code> class. You cannot extend another class.	Implements <code>Runnable</code> , allowing you to extend other classes.
Flexibility	Less flexible because of single inheritance.	More flexible as it allows multiple inheritance.
Memory Usage	Heavier since it carries <code>Thread</code> class properties.	Lighter as only task logic is defined.
Ease of Use	Simpler for small tasks where thread and task are combined.	Better for separating thread creation and task logic.

Using Thread Class

```
java Copy code

// Timer App Example
class Timer extends Thread {
    public void run() {
        System.out.println("Timer started...");
        for (int i = 5; i > 0; i--) {
            System.out.println(i + " seconds left");
            try {
                Thread.sleep(1000); // Wait for 1 second
            } catch (InterruptedException e) {
                System.out.println("Timer interrupted");
            }
        }
        System.out.println("Timer ended.");
    }

    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.start(); // Directly creates and runs the thread
    }
}
```

Scenario Example

- 1. Using Thread Class  
Imagine you're building a **timer app**, and you need a thread to count down time directly.
- 2. Using Runnable Interface  
Imagine you're creating a **file download manager**, and you want to separate the download logic (task) from thread management.

Using Runnable Interface

```
java Copy code

// File Download Manager Example
class FileDownloadTask implements Runnable {
    public void run() {
        System.out.println("File downloading started...");
        try {
            Thread.sleep(3000); // Simulate file download
        } catch (InterruptedException e) {
            System.out.println("Download interrupted");
        }
        System.out.println("File download completed.");
    }

    public static void main(String[] args) {
        FileDownloadTask downloadTask = new FileDownloadTask();
        Thread downloadThread = new Thread(downloadTask); // Separate logic and thread man.
        downloadThread.start();
    }
}
```

14) What is the difference between wait() and sleep() method?

wait()	sleep()
1) The wait() method is defined in Object class.	The sleep() method is defined in Thread class.
2) The wait() method releases the lock.	The sleep() method doesn't release the lock.

---

15) Is it possible to start a thread twice?

No, we cannot restart the thread, as once a thread started and executed, it goes to the Dead state. Therefore, if we try to start a thread twice, it will give a runtimeException "java.lang.IllegalThreadStateException". Consider the following example.

#### Output

*thread is executing now.....*

*Exception in thread "main" java.lang.IllegalThreadStateException*

*at java.lang.Thread.start(Thread.java:708)*

*at Multithread1.main(Multithread1.java:13)*

---

16) Can we call the run() method instead of start()?

Yes, calling run() method directly is valid, but it will not work as a thread instead it will work as a normal object. There will not be context-switching between the threads. When we call the start() method, it internally calls the run() method, which creates a new stack for a thread while directly calling the run() will not create a new stack.

---

17) What about the daemon threads?

The daemon threads are the low priority threads that provide the background support and services to the user threads. Daemon thread gets automatically terminated by the JVM if the program remains with the daemon thread only, and all other user threads are ended/died. There are two methods for daemon thread available in the Thread class:

- **public void setDaemon(boolean status):** It used to mark the thread daemon thread or a user thread.
  - **public boolean isDaemon():** It checks the thread is daemon or not.
- 

20) When should we interrupt a thread?

We should interrupt a thread when we want to break out the sleep or wait state of a thread. We can interrupt a thread by calling the interrupt() throwing the InterruptedException.

---

21) What is the synchronization?

Synchronization is the capability to control the access of multiple threads to any shared resource. It is used:

---

1. To prevent thread interference.
2. To prevent consistency problem.

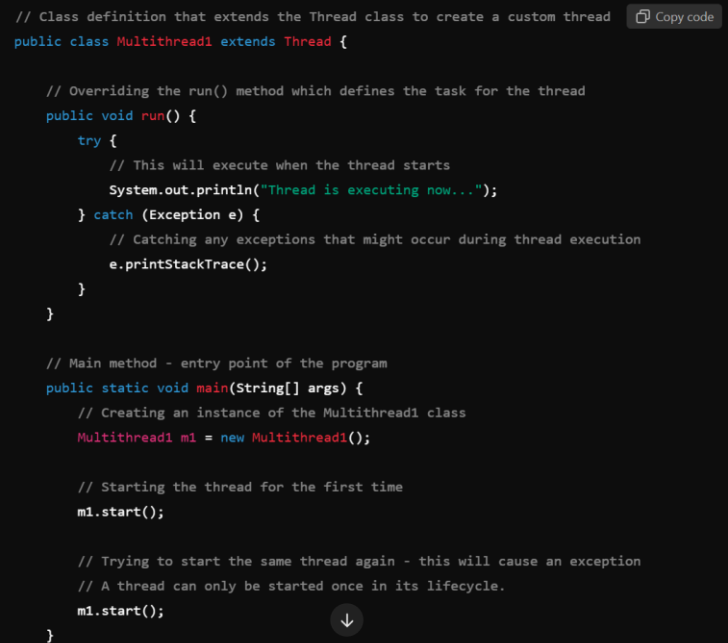
```
// Class definition that extends the Thread class to create a custom thread
public class Multithread1 extends Thread {

    // Overriding the run() method which defines the task for the thread
    public void run() {
        try {
            // This will execute when the thread starts
            System.out.println("Thread is executing now...");
        } catch (Exception e) {
            // Catching any exceptions that might occur during thread execution
            e.printStackTrace();
        }
    }

    // Main method - entry point of the program
    public static void main(String[] args) {
        // Creating an instance of the Multithread1 class
        Multithread1 m1 = new Multithread1();

        // Starting the thread for the first time
        m1.start();

        // Trying to start the same thread again - this will cause an exception
        // A thread can only be started once in its lifecycle.
        m1.start();
    }
}
```



18) Can we make the user thread as daemon thread if the thread is started?

No, if you do so, it will throw `IllegalThreadStateException`. Therefore, we can only create a daemon thread before starting the thread.

```
// Class that extends Thread to define custom thread behavior
class Testdaemon1 extends Thread {

    // Overriding the run() method to define the thread's task
    public void run() {
        System.out.println("Running thread is daemon...");
    }

    // Main method - the entry point of the program
    public static void main(String[] args) {
        // Creating an instance of the Testdaemon1 class
        Testdaemon1 td = new Testdaemon1();

        // Starting the thread
        td.start();

        // Attempting to set the thread as a daemon after it has started
        // This will throw an IllegalThreadStateException
        td.setDaemon(true);
    }
}
```

#### Output

*Running thread is daemon...*

*Exception in thread "main" java.lang.IllegalThreadStateException*


*at java.lang.Thread.setDaemon(Thread.java:1359)*

*at Testdaemon1.main(Testdaemon1.java:8)*

#### Corrected Code

To avoid the exception, you must set the daemon status **before starting the thread**:

java

 Copy code

```
class Testdaemon1 extends Thread {

    public void run() {
        System.out.println("Running thread is daemon...");
    }

    public static void main(String[] args) {
        Testdaemon1 td = new Testdaemon1();

        // Setting the thread as a daemon before starting it
        td.setDaemon(true);

        // Starting the thread after setting it as a daemon
        td.start();
    }
}
```

22) What is the purpose of the Synchronized block?

The Synchronized block can be used to perform synchronization on any specific resource of the method. Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.

- Synchronized block is used to lock an object for any shared resource.
  - The scope of the synchronized block is limited to the block on which, it is applied. Its scope is smaller than a method.
-

23) Can Java object be locked down for exclusive use by a given thread?

Yes. You can lock an object by putting it in a "synchronized" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it.

---

24) What is static synchronization?

If you make any static method as synchronized, the lock will be on the class not on the object. If we use the synchronized keyword before a method so it will lock the object (one thread can access an object at a time) but if we use static synchronized so it will lock a class (one thread can access a class at a time).

---

25) What is the difference between notify() and notifyAll()?

The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

---

26) What is the deadlock?

Deadlock is a situation in which every thread is waiting for a resource which is held by some other waiting thread. In this situation, Neither of the thread executes nor it gets the chance to be executed. Instead, there exists a universal waiting state among all the threads. Deadlock is a very complicated situation which can break our code at runtime.

---

27) How to detect a deadlock condition? How can it be avoided?

We can detect the deadlock condition by running the code on cmd and collecting the Thread Dump, and if any deadlock is present in the code, then a message will appear on cmd.

**Ways to avoid the deadlock condition in Java:**

- **Avoid Nested lock:** Nested lock is the common reason for deadlock as deadlock occurs when we provide locks to various threads so we should give one lock to only one thread at some particular time.
  - **Avoid unnecessary locks:** we must avoid the locks which are not required.
  - **Using thread join:** Thread join helps to wait for a thread until another thread doesn't finish its execution so we can avoid deadlock by maximum use of join method.
- 

28) What is Thread Scheduler in java?

In Java, when we create the threads, they are supervised with the help of a Thread Scheduler, which is the part of JVM. Thread scheduler is only responsible for deciding which thread should be executed. Thread scheduler uses two mechanisms for scheduling the threads: Preemptive and Time Slicing.

Java thread scheduler also works for deciding the following for a thread:

- It selects the priority of the thread.
- It determines the waiting time for a thread
- It checks the Nature of thread

30) How is the safety of a thread achieved?

If a method or class object can be used by multiple threads at a time without any race condition, then the class is thread-safe. Thread safety is used to make a program safe to use in multithreaded programming. It can be achieved by the following ways:



- Synchronization
  - Using Volatile keyword
  - Using a lock based mechanism
  - Use of atomic wrapper classes
- 
- 

32) What is the volatile keyword in java?

Volatile keyword is used in multithreaded programming to achieve the thread safety, as a change in one volatile variable is visible to all other threads so one variable can be used by one thread at a time.

---

33) What do you understand by thread pool?

- Java Thread pool represents a group of worker threads, which are waiting for the task to be allocated.
- Threads in the thread pool are supervised by the service provider which pulls one thread from the pool and assign a job to it.
- After completion of the given task, thread again came to the thread pool.
- The size of the thread pool depends on the total number of threads kept at reserve for execution.

The advantages of the thread pool are :

- Using a thread pool, performance can be enhanced.
  - Using a thread pool, better system stability can occur.
- 
- 

38) What is the difference between Java Callable interface and Runnable interface?

The Callable interface and Runnable interface both are used by the classes which wanted to execute with multiple threads. However, there are two main differences between the both :

- A Callable <V> interface can return a result, whereas the Runnable interface cannot return any result.
  - A Callable <V> interface can throw a checked exception, whereas the Runnable interface cannot throw checked exception.
  - A Callable <V> interface cannot be used before the Java 5 whereas the Runnable interface can be used.
- 
- 

40) What is lock interface in Concurrency API in Java?

The java.util.concurrent.locks.Lock interface is used as the synchronization mechanism. It works similar to the synchronized block. There are a few differences between the lock and synchronized block that are given below.

- Lock interface provides the guarantee of sequence in which the waiting thread will be given the access, whereas the synchronized block doesn't guarantee it.
- Lock interface provides the option of timeout if the lock is not granted whereas the synchronized block doesn't provide that.
- The methods of Lock interface, i.e., Lock() and Unlock() can be called in different methods whereas single synchronized block must be fully contained in a single method.

Three threads simulate banking actions:

1. **Deposit thread:** Adds money to the account.
2. **Withdraw thread:** Waits for sufficient balance to withdraw.
3. **Notification thread:** Notifies when an important update (like balance change) happen

```
class BankAccount {
    private int balance = 0;

    // Deposit money into the account
    public synchronized void deposit(int amount) {
        System.out.println("Depositing: " + amount);
        balance += amount;
        System.out.println("Balance after deposit: " + balance);
        notify(); // Notify waiting threads when deposit happens
    }

    // Withdraw money from the account
    public synchronized void withdraw(int amount) {
        try {
            // Wait if balance is insufficient
            while (balance < amount) {
                System.out.println("Insufficient balance! Waiting for deposit...");
                wait(); // Wait until notified
            }
            System.out.println("Withdrawing: " + amount);
            balance -= amount;
            System.out.println("Balance after withdrawal: " + balance);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

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

        // Thread 1: Deposit thread
        Thread depositThread = new Thread(() -> {
            try {
                Thread.sleep(2000); // Simulate delay in deposit
                account.deposit(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // Thread 2: Withdraw thread
        Thread withdrawThread = new Thread(() -> {
            account.withdraw(300);
        });

        // Thread 3: Notification thread
        Thread notificationThread = new Thread(() -> {
            System.out.println("Bank system update notification running...");
            try {
                depositThread.join(); // Wait for depositThread to complete
                System.out.println("Deposit completed. You can now proceed with transactions.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
}
```

### Output Example:

yaml

```
Insufficient balance! Waiting for deposit...
Bank system update notification running...
Depositing: 500
Balance after deposit: 500
Deposit completed. You can now proceed with transactions.
Withdrawing: 300
Balance after withdrawal: 200
```

```
    }
    });

    // Start all threads
    withdrawThread.start();
    depositThread.start();
    notificationThread.start();
}
}
```

#### **start ()**

- ☐ This method starts a new thread.
- ☐ It makes the thread run the code written inside the run() method.

**Example:**

depositThread.start(); starts the deposit process in a new thread.

---

#### 2. **join()**

- ☐ Makes one thread wait until another thread finishes its work.
- ☐ Useful when a thread depends on the result of another thread.

**Example:**

notificationThread.join(); ensures the notification thread waits until the deposit thread completes.

---

#### 3. **sleep()**

- ☐ Pauses the current thread for some time.
- ☐ Used to simulate delays or give time to other threads.

**Example:**

Thread.sleep(2000); pauses the thread for 2 seconds.

---

#### 4. **wait()**

- ☐ Makes a thread stop and wait until another thread signals it to continue.
- ☐ Often used when a thread is waiting for a specific condition to be met.

**Example:**

withdrawThread.wait(); stops the withdrawal process until there is enough balance.

---

#### 5. **notify()**

- ☐ Wakes up one thread that is waiting.
- ☐ Signals a waiting thread that it can continue.

**Example:**

notify(); tells the withdraw thread to continue after depositing money.

---

#### 6. **notifyAll()**

- ☐ Wakes up all threads that are waiting on the same object.
- ☐ Useful when multiple threads need to be notified at once.

**Example:**

If many threads are waiting for a deposit, notifyAll(); will wake them all.

-----