### **Addressing a Thread**

### Every thread in Java has its own identity, which includes a unique name, a priority level, and a state. You can give threads names to make it easier to identify and debug them.

### Java also lets you change a thread's priority using the built-in method called **setPriority()`.**

**Analogy**: Imagine a line at a bank. Priority customers might be served first, similar to how high-priority threads might get more CPU time.

### **Example**

### ****class PriorityThread extends Thread {

### public void run() {

### System.out.println("Thread with priority: " + this.getPriority());

### }

### }

### public class MainApp {

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

### PriorityThread t1 = new PriorityThread();

### t1.setPriority(Thread.MAX\_PRIORITY); // Set to highest priority

### t1.start();

### }

### }

### ** Real-World Use**: Useful in real-time systems where certain threads (like monitoring or emergency tasks) need more immediate attention than others.

### 

### **isAlive() Method**

### **Purpose**: The isAlive() method checks if a thread is still running. It returns true if the thread has been started and hasn’t finished execution, otherwise false.

### **Why It's Used**: This is useful to check the status of a thread, especially if certain actions need to be taken only after the thread completes.

### **Analogy**: Think of a chef cooking in a kitchen. You could check to see if the chef is still working by peeking inside. If the chef is in there, they’re "alive" (still working).

### **Example**:

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running...");

}

}

public class MainApp {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

System.out.println("Is the thread alive? " + t1.isAlive());

}

}



### **Real-World Use**: Often used in server applications to check if all required background threads are still running or to confirm that critical tasks haven’t stalled.

### **setName() and setName(String name) Methods**

**Purpose**: getName() retrieves the name of a thread, while setName(String name) sets the name of the thread.

**Why It's Used**: Naming threads helps identify and debug specific threads in complex applications.

**Analogy**: Giving names to employees in a company so you can call on them individually or recognize who completed which task.

**Example**:

class MyThread extends Thread {

public void run() {

System.out.println("Running thread: " + this.getName());

}

}

public class MainApp {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.setName("WorkerThread"); // Name the thread

t1.start();

}

}



**Real-World Use**: Useful for logging and monitoring, especially in large systems where multiple threads run similar tasks. Names make it easy to track each thread’s status.

### **currentThread() Method**

**Purpose**: currentThread() is a static method that returns a reference to the currently executing thread.

**Why It's Used**: It allows you to access information about the thread that’s currently running.

**Analogy**: Think of it like looking in a mirror to see who you are in the middle of a task.

**Example**:

public class MainApp {

public static void main(String[] args) {

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

}

}

**Real-World Use**: Often used in debugging or logging to track which thread is executing specific parts of a program.

### **getState() Method**

**Purpose**: getState() retrieves the current state of a thread (e.g., NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, or TERMINATED).

**Why It's Used**: Checking a thread’s state can help diagnose performance issues and understand how threads interact with each other.

**Analogy**: Think of it like checking a traffic light. The light can be red, yellow, or green, indicating whether you should go, slow down, or stop.

**Example:**

public class MainApp extends Thread {

@Override

public void run (){

System.out.println("Thread running");

System.out.println("Thread state: " +getState());

// NEW state before starting

}

public static void main(String[] args) {

MainApp t1 = new MainApp();

t1.start();

System.out.println("Thread state after start: " + t1.getState());

// RUNNABLE or TERMINATED

}

}

### 

### **Real-World Use**: Useful in performance monitoring tools and multi-threaded debugging.

**Locking Using Synchronization in Java**

In Java, synchronization is a method or technique used to manage access to shared resources when multiple threads are running. It makes sure that only one thread can use a specific part of the code at a time. This helps prevent problems like data corruption and inconsistencies.

## **Real-World Analogy:** Think of synchronization like a single bathroom in a house shared by several people. When one person locks the door while using it, others have to wait for their turn. This lock keeps anyone else from entering until the bathroom is free, avoiding interruptions or conflicts. In programming, synchronization acts like this lock for shared resources when multiple threads are active.

**Why Synchronization is Needed**

**1. Thread Interference:** When multiple threads try to modify a shared resource (like a variable or list) without synchronization, they might interfere with each other. This can lead to inconsistent results.

**Example:** Two threads are counting coins and updating a shared "total coins" variable. If both read the value at the same time, they might both update it to the same value, ignoring each other's updates. Synchronization prevents this by allowing only one thread to update the total at a time.

**Key Concepts in Java Synchronization**

**Intrinsic Locks (Monitor Locks):**

Every object in Java has an intrinsic or "monitor" lock.

When a thread enters a synchronized block or method, it "locks" the object, preventing other threads from entering any other synchronized block on the same object.

When the thread exits the block or method, it releases the lock, allowing another waiting thread to proceed.

**How Synchronization Works**

The synchronized keyword in Java is used to lock a method or a block of code so that only one thread can run it at any given time.

This prevents issues that can occur when different threads try to change the same data at the same time..

**Types of Synchronization:**

**Synchronized Method:** Locks the entire method, allowing only one thread to execute it at a time.

**Synchronized Block:** Locks only a specific block of code within a method, improving performance by allowing other parts of the method to run concurrently.

**Synchronized Method**

A synchronized method locks the whole method, which can slow down performance if the method is long or has non-critical parts. Still, it's a straightforward way to ensure thread safety.

**Example Without Synchronization:**

package synchronization.test;

class TestClass extends Thread {

SyncDemo sd; // Reference to the SyncDemo object shared between threads

String name = null; // Name to be passed for printing

// Constructor to initialize SyncDemo object and name

public TestClass(SyncDemo sd, String name) {

this.sd = sd;

this.name = name;

}

// Overridden run method to call the print method in SyncDemo

@Override

public void run() {

sd.print(name);

}

}

public class SyncDemo {

// print method (not synchronized, so not thread-safe)

public void print(String name) {

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

System.out.println(name + ", Welcome to Aimerz "); // Prints a message with the name

try {

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

} catch (InterruptedException e) { // Exception handling for interruption

e.printStackTrace();

}

}

}

// Main method to create and start threads

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

SyncDemo sd = new SyncDemo(); // Single SyncDemo instance shared by both threads

// Creating two threads with different names

TestClass tc1 = new TestClass(sd, "Sunitha"); // Thread tc1 with name "Sunitha"

TestClass tc2 = new TestClass(sd, "Sukumar"); // Thread tc2 with name "Sukumar"

tc1.start(); // Start thread tc1

tc2.start(); // Start thread tc2

}

}

### 

**Let’s understand the above Code**

**Classes and Inheritance:**

We have two classes here: SyncDemo and TestClass.

TestClass extends Thread, which means each TestClass object can act as a separate thread.

**SyncDemo Class:**

This class has a method called print(). It prints a message five times, with a half-second pause between each message.

The print() method takes a String parameter (name) and uses it to personalize the output message by printing the name with "Welcome to Aimerz".

**TestClass Class:**

This class is a thread that will run the print() method in SyncDemo.

TestClass has a reference to SyncDemo (sd) and a name to be passed to print().

The run() method in TestClass calls sd.print(name), which means each thread will print the welcome message five times with its assigned name.

**Main Method (main in SyncDemo):**

In the main method, an instance of SyncDemo (sd) is created.

Two threads (tc1 and tc2) are created using TestClass. Each thread has a different name:

tc1 has the name "Sunitha".

tc2 has the name "Sukumar".

start() is called on both tc1 and tc2, which begins execution of the run() method for each thread, leading each thread to call sd.print(name).

**Expected vs. Actual Output and Thread Safety**

**We might expect the following Output:**

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

### 

**Actual Output:**However, because both threads tc1 and tc2 are running at the same time, their messages will likely mix together, as they are both accessing the print() method in SyncDemo without synchronization.

**For example:**

Sunitha, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sunitha, Welcome to Aimerz

### 

**Why is the Output Mixed?**

**Lack of Synchronization:**

The print() method is not synchronized, meaning both tc1 and tc2 can access it at the same time.

Each thread is printing "Welcome to Aimerz" with its assigned name, but since they are running simultaneously, they “interfere” with each other.

**Thread Interference:**

Since both threads are accessing print() at the same time, their outputs are mixed up. For example, tc1 might print "Sunitha, Welcome to Aimerz" and then tc2 might immediately print "Sukumar, Welcome to Aimerz" before tc1 completes all five messages.

This causes the output to appear mixed up.

**How to Fix It**

To make the print() method "thread-safe," we can add the synchronized keyword, which will ensure only one thread can access print() at a time

Now let’s add the Synchronized keyword to the print() method in below example

**Example With Synchronized Method:**

package synchronization.test;

class TestClass extends Thread {

SyncDemo sd; // Reference to the SyncDemo object shared between threads

String name = null; // Name to be passed for printing

// Constructor to initialize SyncDemo object and name

public TestClass(SyncDemo sd, String name) {

this.sd = sd;

this.name = name;

}

// Overridden run method to call the print method in SyncDemo

@Override

public void run() {

sd.print(name);

}

}

public class SyncDemo {

// Synchronized print method to ensure thread-safe access

public synchronized void print(String name) {

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

System.out.println(name + ", Welcome to Aimerz "); // Prints a message with the name

try {

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

} catch (InterruptedException e) { // Exception handling for interruption

e.printStackTrace();

}

}

}

// Main method to create and start threads

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

SyncDemo sd = new SyncDemo(); // Single SyncDemo instance shared by both threads

// Creating two threads with different names

TestClass tc1 = new TestClass(sd, "Sunitha"); // Thread tc1 with name "Sunitha"

TestClass tc2 = new TestClass(sd, "Sukumar"); // Thread tc2 with name "Sukumar"

tc1.start(); // Start thread tc1

tc2.start(); // Start thread tc2

}

}

### 

**Let us understand the code with synchronized Keyword**

**Classes and Inheritance:**

The code structure is the same as before. We have a SyncDemo class with a print() method and a TestClass class that extends Thread to enable multi-threading.

**Synchronization Added:**

In the SyncDemo class, the print() method now has the synchronized keyword.

This means that only one thread can execute print() at any time. If one thread is already using print(), other threads must wait until it finishes.

**Behavior of print() Method:**

Inside print(), a message is printed five times, with a 0.5-second pause between each print.

With synchronization, only one thread can complete all five prints before the other thread can start.

**Main Method (main in SyncDemo):**

An instance of SyncDemo (sd) is created.

Two threads, tc1 and tc2, are created and associated with the same SyncDemo instance.

Each thread starts with start(), which calls their run() methods, triggering sd.print(name).

**Effect of Synchronization**

**Without synchronized:**

When print() was not synchronized, both threads could access print() at the same time.

This caused their outputs to mix, with messages from Sunitha and Sukumar appearing in random order.

**With synchronized:**

Now, synchronized makes sure that only one thread can run print() at a time.

When tc1 starts and enters print(), tc2 has to wait until tc1 completes all five prints.

Only after tc1 finishes, tc2 can enter print() and start printing its messages.

**Output with synchronized**

Now, the output will be in a cleaner order, as each thread completes all its messages before the other thread starts. Here’s what the output will look like:

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

### 

**Why This Happens**

The synchronized keyword on print() locks the method so that only one thread can enter it at a time.

If tc1 starts first, tc2 will be blocked until tc1 completes all of its five print statements.

This prevents both threads from "interfering" with each other, resulting in organized, sequential output for each name.

**Synchronized Block**

A synchronized block gives you more control over which parts of your code are locked. Instead of locking the whole method, you can lock only the important sections. This makes the program run more efficiently.



**Real-World Analogy:** Think of a shared printer in an office. While the printer is in use, the printer room is locked, but the waiting area outside is open. This approach allows only the printer itself to be controlled, rather than locking access to the entire room.

**Example with Synchronized Block:**

package synchronization.test;

class TestClass extends Thread {

SyncDemo sd; // Reference to the SyncDemo object shared between threads

String name = null; // Name to be passed for printing

// Constructor to initialize SyncDemo object and name

public TestClass(SyncDemo sd, String name) {

this.sd = sd;

this.name = name;

}

// Overridden run method to call the print method in SyncDemo

@Override

public void run() {

sd.print(name);

}

}

public class SyncDemo {

// print method using a synchronized block to make it thread-safe

public void print(String name) {

synchronized (this) { // Only this block is synchronized

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

System.out.println(name + ", Welcome to Aimerz "); // Prints a message with the name

try {

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

} catch (InterruptedException e) { // Exception handling for interruption

e.printStackTrace();

}

}

}

}

// Main method to create and start threads

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

SyncDemo sd = new SyncDemo(); // Single SyncDemo instance shared by both threads

// Creating two threads with different names

TestClass tc1 = new TestClass(sd, "Sunitha"); // Thread tc1 with name "Sunitha"

TestClass tc2 = new TestClass(sd, "Sukumar"); // Thread tc2 with name "Sukumar"

tc1.start(); // Start thread tc1

tc2.start(); // Start thread tc2

}

}

### 

**What happened to the Code**

We used a **synchronized block** instead of a synchronized method.

synchronized (this) locks only the print block inside the print method, ensuring thread safety by allowing only one thread to execute this block at a time.

Other parts of the method (if any) could run without blocking, but since this method only contains the critical code in this example, it works similarly to a synchronized method.

**Now, Expected Output with Synchronized Block**

Since the critical section is protected by a synchronized block, the output will be in sequential order without interference, similar to the result of using a synchronized method:

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sunitha, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

Sukumar, Welcome to Aimerz

### 

Now, each thread will print its full set of messages in order, without mixing with the other thread's output, because the synchronized block ensures only one thread can enter this part of the code at a time.