**How to transfer the value of an object from one thread to another in Java?**

In Java, you can transfer the value of an object from one thread to another using **thread-safe mechanisms** to ensure proper synchronization and avoid issues like race conditions or data corruption.

**1. Using volatile Keyword**

* The volatile keyword ensures that changes made to a variable by one thread are immediately visible to other threads.
* It is useful for transferring simple values (e.g., flags or counters) between threads.
* class SharedData {
* volatile int value = 0;
* }
* Thread thread1 = new Thread(() -> {
* sharedData.value = 10; // Thread 1 sets the value
* });
* Thread thread2 = new Thread(() -> {
* System.out.println(sharedData.value); // Thread 2 reads the value
* });
* thread1.start();
* thread2.start();

**. Using synchronized Blocks or Methods**

* The synchronized keyword ensures that only one thread can access a block of code or method at a time.
* This is useful for transferring complex objects between threads.

class SharedData {

private int value = 0;

public synchronized void setValue(int value) {

this.value = value;

}

public synchronized int getValue() {

return value;

}

}

SharedData sharedData = new SharedData();

Thread thread1 = new Thread(() -> {

sharedData.setValue(10); // Thread 1 sets the value

});

Thread thread2 = new Thread(() -> {

System.out.println(sharedData.getValue()); // Thread 2 reads the value

});

thread1.start();

thread2.start();

**. Using BlockingQueue**

* A BlockingQueue is a thread-safe queue that allows one thread to produce data and another thread to consume it.
* It is ideal for transferring objects between threads in a producer-consumer scenario.

import java.util.concurrent.Exchanger;

Exchanger<String> exchanger = new Exchanger<>();

Thread thread1 = new Thread(() -> {

try {

String data = "Hello from Thread 1";

System.out.println("Thread 1 sending: " + data);

String response = exchanger.exchange(data); // Thread 1 sends data

System.out.println("Thread 1 received: " + response);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

Thread thread2 = new Thread(() -> {

try {

String data = "Hello from Thread 2";

System.out.println("Thread 2 sending: " + data);

String response = exchanger.exchange(data); // Thread 2 sends data

System.out.println("Thread 2 received: " + response);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

thread1.start();

thread2.start();

**. Using Atomic Classes**

* The java.util.concurrent.atomic package provides atomic classes (e.g., AtomicInteger, AtomicReference) for thread-safe operations on single variables.

import java.util.concurrent.atomic.AtomicReference;

AtomicReference<String> sharedData = new AtomicReference<>("Initial Value");

Thread thread1 = new Thread(() -> {

sharedData.set("Updated by Thread 1"); // Thread 1 updates the value

});

Thread thread2 = new Thread(() -> {

System.out.println(sharedData.get()); // Thread 2 reads the value

});

thread1.start();

thread2.start();

**Meaning of the Question:**

The question is asking how to **safely share or transfer data (an object)** between two threads in Java. Since threads run concurrently, direct sharing of data can lead to **race conditions** or **inconsistent states**. Therefore, the question emphasizes the need for **thread-safe mechanisms** to ensure proper synchronization and data integrity.

**Key Points to Remember:**

1. **Thread Safety:** Always ensure that shared data is accessed in a thread-safe manner.
2. **Synchronization:** Use synchronization mechanisms like synchronized, volatile, or atomic classes to avoid race conditions.
3. **Inter-Thread Communication:** Use tools like BlockingQueue or Exchanger for more complex scenarios.