|  |  |  |
| --- | --- | --- |
| 1. **Expected Outcome** | 1. **Activity/Analogy/Real-life usecase** | 1. **Demo idea or Code snippet  (with new learnings mentioned)** |
| 1. User understand the use of multithreading | **Use-case #1:** Consider a scenario where a user initiates a file download in an application. In a single-threaded environment, the UI freezes until the download is complete, making the application unresponsive. | **// Single-threaded approach**  public static void main(String[] args) {  System.out.println("Initiating file download...");  **// 1.1 Simulating a time-consuming download task**  downloadFile();  System.out.println("File download completed. Application is now responsive.");  }  **// 1.2 Simulating a time-consuming download task**  private static void downloadFile() {  try {  Thread.sleep(5000); // Sleep for 5 seconds to simulate download  } catch (InterruptedException e) {  e.printStackTrace();  }  }  **//In this single-threaded example, the application prints the initiation message, performs a simulated download (sleeps for 5 seconds), and then prints the completion message. During the download, the application is unresponsive.** |
| **Use-case #2:** Using multithreading so that UI does not freezes | **// 1.3Creating a download thread**  Thread downloadThread = new Thread(() -> {  // Simulating a time-consuming download task  downloadFile();  });  **//1.4 Start the download thread**  downloadThread.start();  **// By using multithreading the program the processor will always running optimally and the UI wont freeze while downloading a time consuming file.** |
| 2. User should be able to create a thread by implementing the Runnable interface | **Activity #1:** To write a program to download multiple files simultaneously without blocking the main thread, ensuring a responsive user interface. | **// 2.1 Class implementing the Runnable interface**  class MyRunnable implements Runnable {  @Override  public void run() {  // Task to be performed concurrently  for (int i = 0; i < 5; i++) {  System.out.println("Child thread: " + i);  }  }  }  **// 2.2 Create an instance of the class implementing Runnable**  MyRunnable myRunnable = new MyRunnable();  **//2.3 Create a Thread and pass the Runnable instance to it**  Thread myThread = new Thread(myRunnable);  **//2.4 Start the thread**  myThread.start();  **//2.5 Main thread continues its work**  for (int i = 0; i < 5; i++) {  System.out.println("Main thread: "+i);} |
| 3. User understands the need for synchronization | **Use-case #3:** Consider a scenario where our application needs to download multiple files concurrently. Each file download is handled by a separate thread. However, all threads need to update a counter variable. | // **3.1** **Without synchronizes approach**  class FileDownloader {  private int filesDownloaded = 0;  public void downloadFile() {  **// some file download logic**  **// A delay to simulate download time**  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  **//3.2 Increment the counter without synchronization**  filesDownloaded++;  System.out.println("Downloaded by Thread " + filesDownloaded);  }  **// The above code snippet will face data corruption due to lack of synchronization** |
| 4. User should be able to use synchronized keyword for synchronization | **Activity #2** To write a program to synchronize the program in **Use-case #3** | //**4.1 Here we have used synchronized method**  public static **synchronized** void downloadFile() {  **// some file download logic**    **// A delay to simulate download time**  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }    filesDownloaded++;  System.out.println("Downloaded by Thread " + filesDownloaded);  } |
| // **4.2 Here we have used synchronized blocks**  synchronized (FileDownloader.class) {  filesDownloaded++;  } |
| 5. User understands excessive thread usage is detrimental to performance  Given different scenarios user is able to identify the ones where multithreading is necessary | **Activity** **#3**: To discuss the concept of diminishing returns in multithreading and how excessive thread usage can lead to performance degradation. | //**5.1 Excessive use of Multithreading**  private static final List<String> fileUrls  public static void main(String[] args) {  List<Thread> downloadThreads = new ArrayList<>();  for (String fileUrl : fileUrls) {  Thread downloadThread = new Thread(() -> {  **// 5.2 Download file from the given URL**  downloadFile(fileUrl);  });  downloadThreads.add(downloadThread);  downloadThread.start();  }  // **5.3 Download file function**  private static void downloadFile(String fileUrl) {  **// Simulate file download**    }  // **The program creates a new thread for each file download, potentially leading to excessive thread usage.** |
| **Activity #4 :** To discuss scenerious when use of Multithreading is apropriate | // **Various scenerious where use of multithreading is appropriate**  **1.Parallelizable task:**  Multithreading can be beneficial for parallelizing tasks.  **2.Responsiveness Requirements:** Introducing threads can help maintain a responsive user interface, especially in applications with long-running tasks.  **3.Resource Utilization:**  Multithreading can improve overall system performance by allowing parallel execution. |
| **Use-case #4 :**Consider a scenario where introducing multithreading might not be the best choice. For example, a program with sequential dependencies . | // **Sequential order is necessary**  private static void process () {  for (int i = 0; i < 5; i++) {  **// Task with dependency on the result of the previous task**  int result = performTask(i);  System.out.println(result);  }  }  private static int performTask(int taskNumber) {  **// Simulate a time-consuming task**  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  return taskNumber \* 2;  }  // **Here since the current result depends on previous result ,so use use of multithreading is not viable.** |