Week 10

1. **JAVA THREAD MODEL**

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously. In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent. Another benefit of using thread is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

A thread is actually a lightweight process. Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called thread and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

**Multithreading In Java :**

* Used to maximize the CPU utilization.
* We don't want our CPU to be in a free state; for example, Func1() comes into the memory and demands any input/output process. The CPU will need to wait for unit Func1() to complete its input/output operation in such a condition. But, while Func1() completes its I/O operation, the CPU is free and not executing any thread. So, the efficiency of the CPU is decreased in the absence of multithreading.
* In the case of multithreading, if a thread demands any I/O operation, then the CPU will let the thread perform its I/O operation, but it will start the execution of a new thread parallelly. So, in this case, two threads are executing at the same time.

A screenshot of a computer screen

Description automatically generated

**Fig 1: Multithreading**

**The Java Thread Model**

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles. Threads exist in several states. Following are those states:

* New — When we create an instance of Thread class, a thread is in a new state.
* Runnable — The Java thread is in running state.
* Suspended — A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
* Blocked — A java thread can be blocked when waiting for a resource.
* Terminated — A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.

A diagram of a flowchart

Description automatically generated

**Fig 2: Java thread model**

**Thread Class and Runnable Interface**

Java’s multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads. The table below displays the same:

A screenshot of a computer

Description automatically generated

**Main Java Thread**

Now let us see how to use Thread and Runnable interface to create and manage threads, beginning with the main java thread, that all Java programs have. So, let us discuss the main thread. Because this thread affects the other ‘child’ threads. Because it performs various shutdown actions. It is created automatically when your program is started.

**How to Create a Java Thread?**

Java lets you create a thread in following two ways:-

* By implementing the Runnable interface.
* By extending the Thread

**Runnable Interface**

The easiest way to create a thread is to create a class that implements the Runnable interface.

**Extending Java Thread**

The second way to create a thread is to create a new class that extends Thread, then override the run() method and then to create an instance of that class. The run() method is what is executed by the thread after you call start().

1. **THREAD CLASS:**

A Thread class has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the Object class. The Object class implements the Runnable interface. The thread class has the following constructors that are used to perform various operations.

* Thread()
* Thread(Runnable, String name)
* Thread(Runnable target)
* Thread(ThreadGroup group, Runnable target, String name)
* Thread(ThreadGroup group, Runnable target)
* Thread(ThreadGroup group, String name)
* Thread(ThreadGroup group, Runnable target, String name, long stackSize)

**Runnable Interface(run() method)**

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the run() method to perform an action for the thread.

**Syntax:**

|  |
| --- |
| public class Main extends thread {  public void test() {  System.out.println("Threads are very helpful in java");  }  } |

**start() method**

The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the start() method, the thread changes the state from New to Runnable. It executes the run() method when the thread gets the correct time to execute it.

|  |
| --- |
| // Implementing runnable interface by extending Thread class  public class ThreadExample1 extends Thread {  // run() method to perform action for thread.  public void run()  {  int a= 10;  int b=12;  int result = a+b;  System.out.println("Thread started running..");  System.out.println("Sum of two numbers is: "+ result);  }  public static void main( String args[] )  {  // Creating instance of the class extend Thread class  ThreadExample1 t1 = new ThreadExample1();  //calling start method to execute the run() method of the Thread class  t1.start();  }  } |

**Creating thread by implementing the runnable interface**

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

**Syntax:**

|  |
| --- |
| public class Main implements runnable {  public void test() {  System.out.println("Threads are very helpful in java");  }  } |

|  |
| --- |
| class NewThread implements Runnable {  String name;  Thread thread;  NewThread (String name){  this.name = name;  thread = new Thread(this, name);  System.out.println( "A New thread: " + thread+ "is created\n" );  thread.start();  }  public void run() {  try {  for(int j = 5; j > 0; j--) {  System.out.println(name + ": " + j);  Thread.sleep(1000);  }  }catch (InterruptedException e) {  System.out.println(name + " thread Interrupted");  }  System.out.println(name + " thread exiting.");  }  }  class ThreadExample2 {  public static void main(String args[]) {  new NewThread("1st");  new NewThread("2nd");  new NewThread("3rd");  try {  Thread.sleep(8000);  } catch (InterruptedException excetion) {  System.out.println("Inturruption occurs in Main Thread");  }  System.out.println("We are exiting from Main Thread");  }  } |

**Output:**

**A computer screen shot of a program

Description automatically generated**

**Advantages of thread**

* Reduces development time.
* Reduces maintenance costs.
* Improves the performance of complex applications.
* Useful for improving the responsiveness of the user interfaces.
* Used in server applications to improve high throughput and resource utilization.
* Parallelize tasks.
* If a thread cannot use all the computing resources of the CPU (because instructions depend on each other’s result), running another thread can avoid leaving these idle.
* Take advantage of multiprocessor systems

**Disadvantages of thread**

* Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs).
* Execution times of a single thread can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.
* Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing.

**Uses of Thread**

* Java applications are naturally threaded. The runtime environment begins the execution of the program with the main() method in one thread. Garbage collection takes place in another thread. Screen updating occurs in a third thread. There may be other threads running as well, mostly related to the behavior of the virtual machine. All of this happens invisibly to the programmer. Sometimes you’re only concerned with what happens in the primary thread which includes the main() method of a program. If this is the case, you may not need to worry about threading at all.
* The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to utilize the CPU time as much as possible. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread. Each thread has a separate path of its execution. This way a single program can perform two or more tasks simultaneously.
* Threads are lightweight processes; they share the same address space. In a Multithreaded environment, programs make maximum use of CPU so that the idle time can be kept to a minimum.
* To perform asynchronous or background processing.

**Thread class vs Runnable interface**

A screenshot of a computer

Description automatically generated

1. **CREATING MULTIPLE THREADS**

Creating Multiple Threads in Java | In the previous all thread programs, we have used only two threads: main thread, and one new thread (known as child thread). Now, we will learn methods of creating multiple threads in Java program. Basically, when we need to perform several tasks at a time, we can create multiple threads to perform multiple tasks in a program.

For example, to perform two tasks, we can create two threads and attach them to two tasks. Hence, creating multiple threads in Java programming helps to perform more than one task simultaneously. Creating more than one thread to perform multiple tasks is called multithreading in Java. In multiple threading programming, multiple threads are executing simultaneously that improves the performance of CPU because CPU is not idle if other threads are waiting to get some resources.

Multiple threads share the same address space in the heap memory. Therefore, It is good to create multiple threads to execute multiple tasks rather than creating multiple processes.

A diagram of a process

Description automatically generated

**Fig 3: Multiple threads**

|  |
| --- |
| // Two threads performing two tasks at a time.  public class MyThread extends Thread  {  // Declare a String variable to represent task.  String task;    MyThread(String task)  {  this.task = task;  }  public void run()  {  for(int i = 1; i <= 5; i++)  {  System.out.println(task+ " : " +i);  try  {  Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.  }  catch(InterruptedException ie) {  System.out.println(ie.getMessage());  }  } // end of for loop.  } // end of run() method.  public static void main(String[] args)  {  // Create two objects to represent two tasks.  MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an argument to its constructor.  MyThread th2 = new MyThread("Show your seat number");    // Create two objects of Thread class and pass two objects as parameter to constructor of Thread class.  Thread t1 = new Thread(th1);  Thread t2 = new Thread(th2);  t1.start();  t2.start();  }  }  **Output:**  Cut the ticket : 1  Show your seat number : 1  Show your seat number : 2  Cut the ticket : 2  Show your seat number : 3  Cut the ticket : 3  Show your seat number : 4  Cut the ticket : 4  Show your seat number : 5  Cut the ticket : 5 |

1. In the preceding example program, we have created two threads on two objects of MyThread class. Here, we created two objects to represent two tasks. When we will run the above program, the main thread starts running immediately. Two threads will generate from the main thread that will perform two different tasks.

2. When t1.start(); is executed by JVM, it starts execution of code inside run() method and print the statement “Cut the ticket” on the console.

3. When JVM executes Thread.sleep(1000); inside the try block, it pauses the thread execution for 1000 milliseconds. Here. sleep() method is a static method that is used to pauses the execution of thread for a specified amount of time. For example, Thread.sleep(1000); will pause the execution of thread for 1000 milliseconds (1 sec). 1000 milliseconds means 1 second. Since sleep() method can throw an exception named InterruptedException, we will catch it into catch block.

4. Meanwhile, JVM executes t2.start(); and second thread starts execution of code inside the run() method almost simultaneously. It will print the statement “Show your seat number”. Now, the second thread will undergo to sleep for 1000 milliseconds.

5. When the pause time period of the first thread is elapsed, it will reenter into running state and starts the execution of code inside run() method. The same process will also happen for second thread. In this manner, both threads will perform two tasks almost simultaneously.

A line of lines with black dots and a black dot

Description automatically generated

**Fig 4: Multithreading**

**Multiple Threads acting on Single object**

It is also possible to create two or more threads on a single object.

|  |
| --- |
| public class MultipleThread implements Runnable  {  String task;  MultipleThread(String task)  {  this.task = task;  }  public void run()  {  for(int i = 1; i <= 5; i++)  {  System.out.println(task+ ":" +i);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  public static void main(String[] args)  {  Thread nThread = Thread.currentThread();  System.out.println("Name of thread: " +nThread);    // Multiple child threads acting on single object.  MultipleThread mt = new MultipleThread("Hello Java");  Thread t1 = new Thread(mt);  Thread t2 = new Thread(mt);  Thread t3 = new Thread(mt);  t1.start();  t2.start();  t3.start();  int count = Thread.activeCount();  System.out.println("No of active threads: " +count);  }  }  **Output:**  Name of thread: Thread[main,5,main]  No of active threads: 4  Hello Java:1  Hello Java:1  Hello Java:1  Hello Java:2  Hello Java:2  . . . . . .  . . . . . .  Hello Java:5 |

1. **THREAD PRIORITIES**

Multithreading is a powerful concept in Java that allows multiple threads to run concurrently within a single program. Multithreading allows developers to improve the performance, responsiveness, and efficiency of their applications. One aspect of thread management is assigning them priorities, which can influence their execution order and resource allocation.

**Understanding thread priority in Java**

Each thread in Java is given a priority value ranging from 1 to 10, with 1 being the lowest and 10 being the highest. A new thread inherits the priority of its parent thread by default. The thread priorities are used as a hint by the Java Virtual Machine (JVM) to determine the order in which threads should be scheduled for execution. It is important to note, however, that thread priorities do not always have the same effect across different JVM implementations and operating systems.

**Setting Thread Priority in Java**

Java provides the setPriority() method in the Thread class to set the priority of a thread. Here’s an example:

Thread thread1 = new Thread(new MyRunnable());

Thread thread2 = new Thread(new MyRunnable());

thread1.setPriority(Thread.MIN\_PRIORITY); // Setting lowest priority

thread2.setPriority(Thread.MAX\_PRIORITY); // Setting highest priority

In this example, we create two threads, thread1 and thread2, and assign them different priorities using the setPriority() method. Thread.MIN\_PRIORITY represents the lowest priority, while Thread.MAX\_PRIORITY represents the highest priority.

**Setter and Getter Methods of Thread Priority**

* public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.
* public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method changes or assigns the thread’s priority to newPriority. If the value newPriority is outside the range of 1 (minimum) to 10 (maximum), the method throws an IllegalArgumentException.
* public static int MIN\_PRIORITY
* public static int NORM\_PRIORITY
* public static int MAX\_PRIORITY

A thread’s default priority is 5 (NORM\_PRIORITY). The MIN\_PRIORITY value is 1 and the MAX\_PRIORITY value is 10.

|  |
| --- |
| import java.lang.\*;  public class ThreadPriorityExample extends Thread {  public void run() {  System.out.println("Inside the run() method");  }  public static void main(String argvs[]) {  ThreadPriorityExample th1 = new ThreadPriorityExample();  ThreadPriorityExample th2 = new ThreadPriorityExample();  ThreadPriorityExample th3 = new ThreadPriorityExample();  System.out.println("Priority of the thread th1 is : " + th1.getPriority());  System.out.println("Priority of the thread th2 is : " + th2.getPriority());  System.out.println("Priority of the thread th2 is : " + th2.getPriority());  th1.setPriority(6);  th2.setPriority(3);  th3.setPriority(9);  System.out.println("Priority of the thread th1 is : " + th1.getPriority());  System.out.println("Priority of the thread th2 is : " + th2.getPriority());  System.out.println("Priority of the thread th3 is : " + th3.getPriority());  System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());  System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  Thread.currentThread().setPriority(10);  System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  }  }  **Output:**  Priority of the thread th1 is : 5  Priority of the thread th2 is : 5  Priority of the thread th2 is : 5  Priority of the thread th1 is : 6  Priority of the thread th2 is : 3  Priority of the thread th3 is : 9  Currently Executing The Thread : main  Priority of the main thread is : 5  Priority of the main thread is : 10 |

When it comes to thread execution, we know that a thread with a high priority will take precedence over a thread with a lower priority. However, there may be other situations in which two threads have the same priority. The Java thread scheduler handles all of the processing for thread management.

**Impact of Thread Priorities**

Thread priorities influence how the JVM schedules and allocates resources to threads. A higher priority thread has a better chance of being executed before lower priority threads. However, it’s important to remember that priority alone does not determine the order of thread execution. The JVM’s thread scheduler takes other factors, such as the underlying operating system and thread states, into account.

Thread priority is useful when certain threads require more processing time or have critical tasks. By giving such threads a higher priority, developers can ensure that they receive enough CPU time to complete their tasks on time. It is not, however, recommended to rely solely on thread priorities to achieve specific execution behavior.

**Best Practices and Considerations for Thread Priority in Java**

When working with thread priorities in Java, it’s essential to keep the following considerations in mind:

* Thread priorities are relative: Thread priorities are relative to each other and not absolute. The relationship between priorities is what matters, rather than the specific values assigned.
* Platform dependency: Thread scheduling behavior can vary across different JVM implementations and operating systems. Therefore, it’s crucial to avoid making assumptions about the precise behavior of thread priorities.
* Avoid excessive priority use: Overusing thread priorities or assigning extreme values to them may lead to unexpected and non-portable behavior. It is generally advisable to use priorities sparingly and focus on designing efficient algorithms and proper thread synchronization.
* Design robust and responsive applications: Prioritizing threads alone does not guarantee responsive and well-performing applications. It is important to design applications carefully, taking into account proper thread synchronization, efficient resource utilization, and algorithm optimization