Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java: Multi-Threading/5b71846f17a1437e093bd496

TUTORIAL

Java: Multi-Threading

Topics

1.2 Creating a new Thread

1.3 Interface Runnable

1.8 States of Threads

1.9 Thread Scheduling and Priority

Java supports single-thread as well as multi-thread operations. A single-thread program has a single entry point (the main() method) and a single exit point. A multi-thread program has an initial entry point (the main() method), followed by many entry and exit points, which are run concurrently with the main(). The term "concurrency" refers to doing multiple tasks at the same time.

Java has built-in support for concurrent programming by running multiple threads concurrently within a single program. A thread, also called a lightweight process, is a single sequential flow of programming operations, with a definite beginning and an end. During the lifetime of the thread, there is only a single point of execution. A thread by itself is not a program because it cannot run on its own. Instead, it runs within a program.

If your computer has only a single CPU, you might be wondering how it can execute more than one thread at the same time. In single-processor systems, only a single thread of execution occurs at a given instant. The CPU quickly switches back and forth between several threads to create the illusion that the threads are executing at the same time. Single-processor systems support logical concurrency, not physical concurrency. Logical concurrency is the characteristic exhibited when multiple threads execute with separate, independent flows of control. On multiprocessor systems, several threads do, in fact, execute at the same time, and physical concurrency is achieved. The important feature of multithreaded programs is that they support logical concurrency, not that physical concurrency is actually achieved.

Many programming languages support multiprogramming. Multiprogramming is the logically concurrent execution of multiple programs. For example, a program can request that the operating system execute programs A, B, and C by having it spawn a separate process for each program. These programs can run in a parallel manner, depending upon the multiprogramming features supported by the underlying operating system. Multithreading differs from multiprogramming in that multithreading provides concurrency within the context of a single process, but multiprogramming also provides concurrency between processes. Threads are not complete processes in and of themselves. They are a separate flow of control that occurs within a process.

The advantage of multithreading is that concurrency can be used within a process to implement multiple instances of simultaneous services. Multithreading also requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more effectively.

An example of a multithreaded application is a multithreaded Web server, which is able to efficiently handle multiple browser requests—one request per each processing thread.

Creating a new Thread

Java, unlike many other programming languages, provides native support for multithreading. This support is centered on the java.lang.Thread class, the java.lang.Runnable interface, and methods of the java.lang.Object class. Support is also provided through synchronized methods and statements.

The Thread class provides the capability to create objects of class Thread, each with their own separate flow of control. The Thread class encapsulates the data and methods associated with separate threads of execution and enables multithreading to be integrated within Java's object-oriented framework. The minimal multithreading support required of the Thread class (or other classes that support multithreading) is specified by the java.lang.Runnable interface. This interface defines an important method—the run() method—which provides the entry point for a separate thread of execution. The Object class supports multithreading through the wait(), notify(), and notifyAll() methods. These methods allow threads to suspend their execution to wait for processing resources and to be informed when to resume execution as the resources become available. Other classes of the java.lang package, such as ThreadGroup, also support multithreading.

Java provides two approaches to creating threads. In the first approach, you create a subclass of class Thread and override the run() method to provide an entry point into the thread's execution. When you create an instance of your Thread subclass, you invoke its start() method to cause the thread to execute as an independent sequence of instructions. The start() method is inherited from the Thread class. It initializes the Thread object, using your operating system's multithreading capabilities, and invokes the run() method. Following code provides an example of creating threads.

Class Thread

The class `java.lang.Thread` has the following constructors:

```
public Thread();
public Thread(String threadName);
public Thread(Runnable target);
public Thread(Runnable target, String threadName);
```

The first two constructors are used for creating a thread by sub-classing the `Thread` class. The next two constructors are used for creating a thread with an instance of class that implements `Runnable` interface.

Interface Runnable

The interface `java.lang.Runnable` declares one `abstract` method `run()`, which is used to specify the running behavior of the thread:

```
public void run();
```

```java
class Thread1 extends Thread
{
  static String message[] = {"CODE","QUOTIENT","GET",
"BETTER,","@", "CODING,"};
  public Thread1(String id)
  {
    super(id);
  }
  public void run()
  {
    String name = getName();
    for(int i=0;i<message.length;++i) {
      randomWait();
      System.out.println(name+message[i]);
    }
  }
  void randomWait()
  {
    try
    {
      sleep((long)(300*Math.random()));
    }
    catch (InterruptedException x)
    {
      System.out.println("Interrupted!");
    }
  }
}
class Main
{
  public static void main(String args[])
  {
    Thread1 thread1 = new Thread1("Thread1: ");
    Thread1 thread2 = new Thread1("Thread2: ");
    thread1.start();
    thread2.start();
    boolean thread1IsAlive = true;
    boolean thread2IsAlive = true;
    do {
      if(thread1IsAlive && !thread1.isAlive())
      {
        thread1IsAlive = false;
        System.out.println("Thread 1 is dead.");
      }
      if(thread2IsAlive && !thread2.isAlive())
      {
        thread2IsAlive = false;
        System.out.println("Thread 2 is dead.");
      }
    }while(thread1IsAlive || thread2IsAlive);
  }
}
```

The above output shows how two threads execute in sequence, displaying information to the console window. The program creates two threads of execution, thread1 and thread2, from the *Thread1* class. It then starts both threads and executes a do statement that waits for the threads to die. The threads display the “CODE QUOTIENT GET BETTER @ CODING.” message word by word, while waiting a short, random amount of time between each word. Because both threads share the console window, the program's output identifies which threads wrote to the console at various times during the program's execution.

The approach to creating threads in above code is very simple and straightforward. However, it has the drawback of requiring your Thread objects to be under the Thread class in the class hierarchy. Java's other approach to creating threads does not limit the location of your Thread objects within the class hierarchy. In this approach, your class implements the java.lang.Runnable interface. The Runnable interface consists of a single method, the run() method, which must be overridden by your class. The run() method provides an entry point into your thread's execution. In order to run an object of your class as an independent thread, you pass it as an argument to a constructor of class Thread. Code below shows this approach:

Java

```java
class MyClass implements Runnable
{
  static String message[]={"CODE","QUOTIENT","GET",
"BETTER,","@", "CODING,"};
  String name;
  public MyClass(String id)
  {
    name = id;
  }
  public void run()
  {
    for(int i=0;i<message.length;++i)
    {
      randomWait();
      System.out.println(name+message[i]);
    }
  }
  void randomWait()
  {
    try
    {
      Thread.currentThread().sleep((long)
(300*Math.random()));
    }
    catch (InterruptedException x)
    {
      System.out.println("Interrupted!");
    }
  }
}

class Main
{
  public static void main(String args[])
  {
    Thread thread1 = new Thread(new MyClass("Thread1:
"));
    Thread thread2 = new Thread(new MyClass("Thread2:
"));
    thread1.start();
    thread2.start();
    boolean thread1IsAlive = true;
    boolean thread2IsAlive = true;
    do {
      if(thread1IsAlive && !thread1.isAlive())
      {
        thread1IsAlive = false;
        System.out.println("Thread 1 is dead.");
      }
      if(thread2IsAlive && !thread2.isAlive())
      {
        thread2IsAlive = false;
        System.out.println("Thread 2 is dead.");
      }
    }while(thread1IsAlive || thread2IsAlive);
  }
}


```

The Thread2 program is very similar to Thread1. It may even displays the same output. Thread2 differs from Thread1 only in the way that the threads are created. Because these two examples are so similar, you might be wondering why you would pick one approach to creating a class over another. The advantage of using the Runnable interface is that your class does not need to extend the Thread class. This is a very helpful feature when you create multithreaded applets. The only disadvantage to this approach is that you have to do a little more work to create and execute your threads.

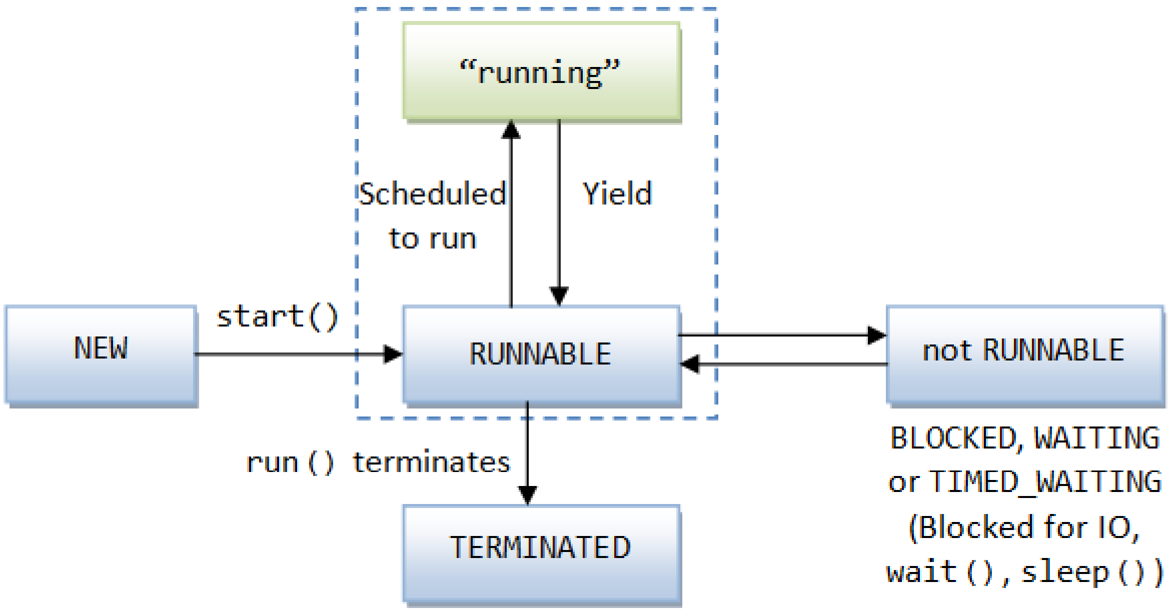
States of Threads

A thread is referred as being dead when their processing had completed. This is the standard Java term for referring to that state of a thread's execution. After a thread reaches the dead state, it can't be restarted. The thread still exists as an object (that is as a Thread or Runnable object), it just doesn't execute as a separate thread of execution.

Threads have several other well-defined states in addition to the dead state. These states are: -

- Ready—When a thread is first created, it doesn't begin executing immediately. You must first invoke its start() method, and then the thread scheduler must allocate it CPU time. A thread may also enter the ready state if, after previously executing, it stopped for a while and then became ready to resume its execution.
- Running—Threads are born to run. A thread is in the running state when it is actually executing. It may leave this state for a number of reasons.
- Waiting—A running thread may perform a number of actions that will cause it to wait. A common example is when the thread performs some type of input or output operation. In Codes above, threads are waiting because they invoked their sleep() methods.

In general, a thread is either ready, running, waiting, or dead. A thread begins as a ready thread and then enters the running state when it is scheduled by the thread scheduler. The thread may be preempted by other threads and returned to the ready state, or it may wait on a resource, object lock, or simply go to sleep. When this happens, the thread enters the waiting state. To run again, the thread must enter the ready state. Eventually, the thread will cease its execution and enter the dead state. The life cycle of a Thread can be easily summarized as below figure: -



Thread Scheduling and Priority

After examining the thread state diagram shown in figure above you may wonder exactly what causes a thread to move from one state to another. In this section, we'll look at how threads move between the ready and running state. In later sections, we'll look at movement in and out of the waiting state. A thread moves back and forth between the ready and running state because multi-threading requires multiple threads to share execution time with each other, based on the availability of the system's CPU (or CPUs). The approach used to determine which threads should execute at a given time is referred to as scheduling. Scheduling is performed by the Java runtime system. It schedules threads based on their priority. Higher priority threads are run before lower priority threads. Threads of equal priority have an equal chance of running. While the above rules guide thread scheduling, the actual details of thread scheduling are platform-dependent.

Most operating systems support one of two common approaches to thread scheduling: -

Preemptive scheduling — The highest priority thread continues to execute unless it dies, waits, or is preempted by a higher priority thread coming into existence. The latter can occur as the result of a thread lowering its priority or creating a higher priority thread.

Time slicing — A thread executes for a specific slice of time and then enters the ready state.

Thread Priority: A thread's priority is an integer value between MIN_PRIORITY and MAX_PRIORITY. These constants are defined in the Thread class. A thread's priority is set when it is created. It is set to the same priority as the thread that created it. The default priority of a thread is NORM_PRIORITY. The priority of a thread can be changed using the setPriority() method.

At this point, the thread scheduler determines whether it should return the thread to the ready state or schedule a different thread. Both approaches have their advantages and disadvantages. Preemptive scheduling is more predictable. However, its disadvantage is that a higher priority thread could execute forever, preventing lower priority threads from executing. The time slicing approach is less predictable but better able to handle selfish threads. Windows and Macintosh implementations of Java follow a time slicing approach. Solaris and other UNIX implementations follow a preemptive scheduling approach.

Note: Yielding, The simplest way for a running thread to enter the ready state is for it to invoke its yield() method. The yield() method is a static method of the Thread class. When a thread invokes its yield() method, it simply returns to the ready state. It then returns to the running state at the discretion of the thread scheduler.

Sleeping and Waking

A thread can enter the waiting state by invoking its sleep() method. The sleep() method, like the yield() method, is a static method of the Thread class. It takes a millisecond time value and an optional nanosecond time value as arguments. When a thread invokes its sleep() method, it enters the waiting state. It returns to the ready state after the specified time has expired. It is possible for another thread to awaken a sleeping thread by invoking the sleeping thread's interrupt() method. The sleeping thread then enters the ready state. When it reenters the running state, execution continues with the thread's InterruptedException handler.