|  |
| --- |
|  |
| Thread: |
|  | A Thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time. |
|  | A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread. |
|  | Following are the stages of the life cycle − |
|  | 1.) New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread. |
|  | 2.) Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task. |
|  | 3.) Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing. |
|  | 4.) Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs. |
|  | 5.) Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates |
|  |  |
|  | Code : |
|  |  |
|  | class RunnableDemo implements Runnable { |
|  | private Thread t; |
|  | private String threadName; |
|  |  |
|  | RunnableDemo( String name) { |
|  | threadName = name; |
|  | System.out.println("Creating " + threadName ); |
|  | } |
|  |  |
|  | public void run() { |
|  | System.out.println("Running " + threadName ); |
|  | try { |
|  | for(int i = 4; i > 0; i--) { |
|  | System.out.println("Thread: " + threadName + ", " + i); |
|  | // Let the thread sleep for a while. |
|  | Thread.sleep(50); |
|  | } |
|  | } catch (InterruptedException e) { |
|  | System.out.println("Thread " + threadName + " interrupted."); |
|  | } |
|  | System.out.println("Thread " + threadName + " exiting."); |
|  | } |
|  |  |
|  | public void start () { |
|  | System.out.println("Starting " + threadName ); |
|  | if (t == null) { |
|  | t = new Thread (this, threadName); |
|  | t.start (); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public class TestThread { |
|  |  |
|  | public static void main(String args[]) { |
|  | RunnableDemo R1 = new RunnableDemo( "Thread-1"); |
|  | R1.start(); |
|  |  |
|  | RunnableDemo R2 = new RunnableDemo( "Thread-2"); |
|  | R2.start(); |
|  | } |
|  | } |
|  |  |
|  | Singleton Class: |
|  | Singleton Pattern is said that just define a class that has only one instance and provides a global point of access to it. |
|  | In other words, a class must ensure that only single instance should be created and single object can be used by all other classes. |
|  | There are two forms of singleton design pattern : |
|  | 1.) Early Instantiation: creation of instance at load time. |
|  | Code : |
|  | class A{ |
|  | private static A obj=new A();//Early, instance will be created at load time |
|  | private A(){} |
|  |  |
|  | public static A getA(){ |
|  | return obj; |
|  | } |
|  |  |
|  | public void doSomething(){ |
|  | //write your code |
|  | } |
|  | } |
|  |  |
|  | 2.) Lazy Instantiation: creation of instance when required. |
|  | Code : |
|  |  |
|  | class A{ |
|  | private static A obj; |
|  | private A(){} |
|  |  |
|  | public static A getA(){ |
|  | if (obj == null){ |
|  | synchronized(Singleton.class){ |
|  | if (obj == null){ |
|  | obj = new Singleton();//instance will be created at request time |
|  | } |
|  | } |
|  | } |
|  | return obj; |
|  | } |
|  | public void doSomething(){ |
|  | //write your code |
|  | } |
|  | } |
|  |  |
|  | 3.) Making Singletons With Enum in Java |
|  | Since enums are inherently serializable, we don't need to implement it with a serializable interface. The reflection problem is also not there. |
|  | public enum Singleton { |
|  | INSTANCE; |
|  | } |
|  | Advantage of Singleton design pattern : |
|  | Saves memory because object is not created at each request. Only single instance is reused again and again. |
|  |  |
|  | Usage of Singleton design pattern : |
|  | Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc. |
|  |  |
|  | Looping Basic Concepts , break , continue and return |
|  | A Loop executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times. |
|  | Types of Loops |
|  | Depending upon the position of a control statement in a program, a loop is classified into two types: |
|  | 1. Entry controlled loop : In an entry controlled loop, a condition is checked before executing the body of a loop. It is also called as a pre-checking loop. |
|  |  |
|  | 2. Exit controlled loop : In an exit controlled loop, a condition is checked after executing the body of a loop. It is also called as a post-checking loop. |
|  | While Loop |
|  | A while loop is the most straightforward looping structure. The basic format of while loop is as follows: |
|  | while (condition) { |
|  | statements; |
|  | } |
|  | Do-While loop |
|  | A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop. |
|  | The basic format of while loop is as follows: |
|  | do { |
|  | statements |
|  | } while (expression); |
|  | For loop |
|  | A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows: |
|  | for (initial value; condition; incrementation or decrementation ) |
|  | { |
|  | statements; |
|  | } |
|  | Break Statement |
|  | The break statement is used mainly in in the switch statement. It is also useful for immediately stopping a loop. |
|  | We consider the following program which introduces a break to exit a while loop: |
|  | #include <stdio.h> |
|  | int main() { |
|  | int num = 5; |
|  | while (num > 0) { |
|  | if (num == 3) |
|  | break; |
|  | printf("%d\n", num); |
|  | num--; |
|  | }} |
|  | Output: |
|  | 5 |
|  | 4 |
|  | Continue Statement |
|  | When you want to skip to the next iteration but remain in the loop, you should use the continue statement. |
|  | For example: |
|  | #include <stdio.h> |
|  | int main() { |
|  | int nb = 7; |
|  | while (nb > 0) { |
|  | nb--; |
|  | if (nb == 5) |
|  | continue; |
|  | printf("%d\n", nb); |
|  | }} |
|  | Output: |
|  | 6 |
|  | 5 |
|  | 4 |
|  | 3 |
|  | 2 |
|  | 1 |
|  |  |
|  | return |
|  | A return statement ends the execution of a function, and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A return statement can return a value to the calling function. |
|  |  |
|  | Data structure (List , Set , Map): |
|  | List : |
|  | List allows duplicate elements. Any number of duplicate elements can be inserted into the list without affecting the same existing values and their indexes. |
|  | List allows any number of null values. |
|  | List and all of its implementation classes maintains the insertion order. |
|  | Commonly used classes in List: ArrayList, LinkedList etc. |
|  |  |
|  |  |
|  |  |
|  |  |
|  | Set : |
|  | Set doesn’t allow duplicates. Set and all of the classes which implements Set interface should have unique elements. |
|  | Set allows single null value at most. |
|  | Set doesn’t maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order. |
|  | Commonly used classes in Set:HashSet, LinkedHashSet, TreeSet, SortedSet etc |
|  |  |
|  | Map : |
|  | Map stored the elements as key & value pair. Map doesn’t allow duplicate keys while it allows duplicate values. |
|  | Map can have single null key at most and any number of null values. |
|  | Map also doesn’t stores the elements in an order, however few of its classes does the same. For e.g. TreeMap sorts the map in the ascending order of keys and LinkedHashMap sorts the elements in the insertion order, the order in which the elements got added to the LinkedHashMap. |
|  | Commonly used classes in Map: HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap etc. |
|  |  |