**CSE 310**

**Week 5**

1. **Daemon thread:**
   1. **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
   2. There are many java daemon threads running automatically e.g. gc, finalizer etc.
   3. Points to remember:
      1. It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
      2. Its life depends on user threads.
      3. It is a low priority thread.
   4. Why JVM terminates the daemon thread if there is no user thread?
      1. The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread? That is why JVM terminates the daemon thread if there is no user thread.
   5. **Methods for Daemon thread for java thread class:**
      1. **public void setDaemon(boolean status)-** is used to mark the current thread as daemon thread or user thread.
      2. **public boolean isDaemon()-**is used to check that current is daemon.
   6. **Please refer to codes section**
   7. If you call the setDaemon() method after starting the thread, it would throw **IllegalThreadStateException**. Please refer to code.
2. **Thread State:**
   1. A [thread](https://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states:
      1. New
      2. Runnable
      3. Blocked
      4. Waiting
      5. Timed Waiting
      6. Terminated

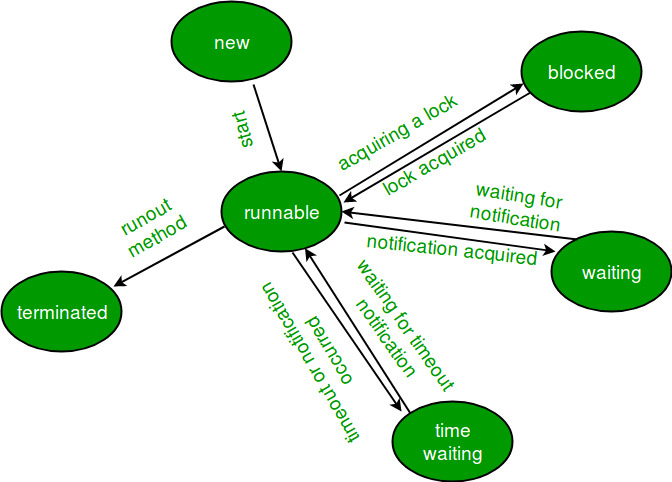
****

Figure Life cycle of a thread

* 1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.
  2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.  
     A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
  3. **Blocked/ Waiting state:** When a thread is temporarily inactive, then it’s in one of the following states:
     1. **Blocked**
     2. **Waiting**

For example, when a thread is waiting for **I/O** to complete, it lies in the **blocked** state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the **blocked** state when it tries to access a protected section of code that is currently **locked** by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the **waiting** state when it **waits for another thread** on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

* 1. **Timed Waiting state:** A thread lies in timed waiting state when it calls a method with a **time out parameter**. A thread lies in this state until the **timeout is completed** or until **a notification is received**. For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.
  2. **Terminated state:** A thread terminates because of either of the following reasons:
     1. Becauseit exists normally. This happens when the code of thread has entirely executed by the program.
     2. Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.
  3. Please refer to code.

1. **Thread Synchronization:**
   1. Synchronization in java is the capability to control the access of multiple threads to any shared resource.
   2. Java Synchronization is better option where we want to allow only one thread to access the shared resource.
   3. **Why use synchronization?**
      1. To prevent thread interference.
      2. To prevent consistency problem.
   4. There are two types of thread synchronization mutual exclusive and inter-thread communication.
      1. Mutual exclusive
      2. Cooperation (Inter thread communication)
   5. Mutual Exclusive:
      1. Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
         1. By synchronized block
         2. By synchronized method
         3. By static synchronization
   6. **Concept of lock in java**
      1. Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
   7. Please refer to codes for understanding problem without synchronization.
   8. **Java Synchronized Method:**
      1. If you declare any method as synchronized, it is known as synchronized method.
      2. Synchronized method is used to lock an object for any shared resource.
      3. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.
      4. Syntax: <access modifier> **synchronized** <return type> <method name>
      5. Please refer to code.
   9. **Synchronized Block in Java**
      1. Synchronized block can be used to perform synchronization on any specific resource of the method.
      2. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block
      3. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
      4. Synchronized block is used to lock an object for any shared resource.
      5. Scope of synchronized block is smaller than the method.
      6. Syntax:

**synchronized** (object reference expression)

{

//code block

}

* + 1. Please refer to code.
  1. **Static Synchronization**
     1. **If you make any static method as synchronized, the lock will be on the class not on object.**
     2. Problem without static synchronization

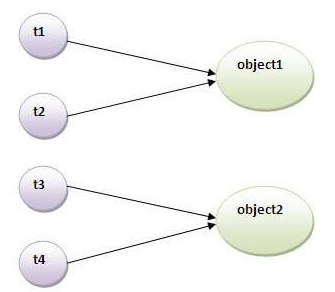
****

Figure 2: Problem without static synchronization

* + 1. Please refer to code.
  1. **Inter thread communication / Cooperation:**
     1. **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
     2. Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
     3. It is implemented by following methods of **Object class**:
        1. wait()
        2. notify()
        3. notifyAll()
     4. **wait() method:**
        1. Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
        2. The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.
        3. Methods:
           1. **public final void wait()throws InterruptedException**: waits until object is notified.
           2. **public final void wait(long timeout)throws InterruptedException:** waits for the specified amount of time.
     5. **notify() method:**
        1. Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
        2. Syntax:

**public final void notify()**

* + 1. **notifyAll() method:**
       1. Wakes up all threads that are waiting on this object's monitor.
       2. Syntax:

**public final void notifyAll()**

* + 1. **Understanding the process of inter thread communication**

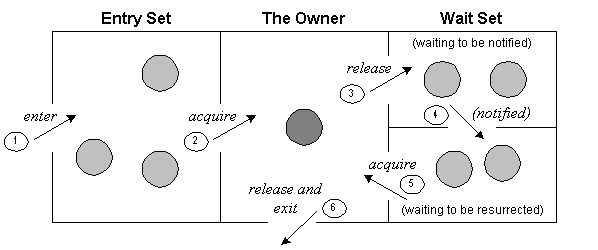
****

Figure 3: Inter thread communication

The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

### It is because they are related to lock and object has a lock.

* + 1. Please refer to code.

**Design Patterns:**

* + - 1. Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.
      2. Gang of Four (GOF)
      3. Usage of design patterns:

### Common platform for developers

* + - * 1. Best practices
      1. Types of design patterns:
         1. Creational pattern
         2. Structural pattern
         3. Behavioral pattern
         4. J2EE Patterns
      2. **Factory Pattern:**
         1. Creational design pattern
         2. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

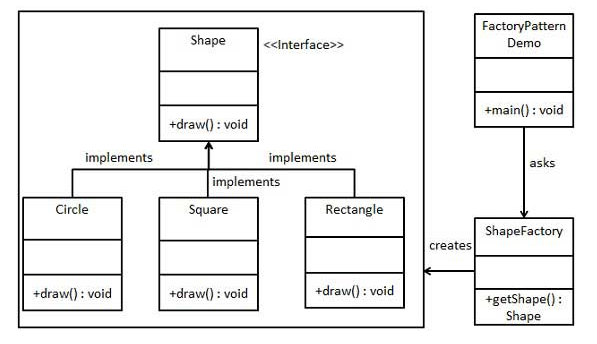


Figure : Factory Pattern

* + - * 1. Please refer to code.
      1. **Singleton pattern:**
         1. Creational pattern.
         2. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

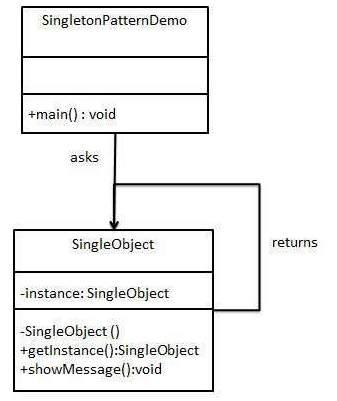


Figure : Singleton Pattern

* + - * 1. Please refer to code.
      1. **Façade pattern:**
         1. Structural design pattern.
         2. Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.

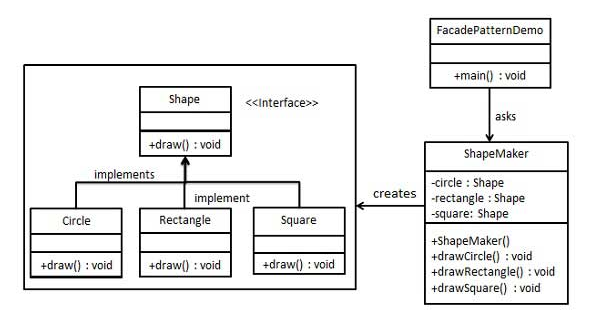
****

Figure : Facade pattern

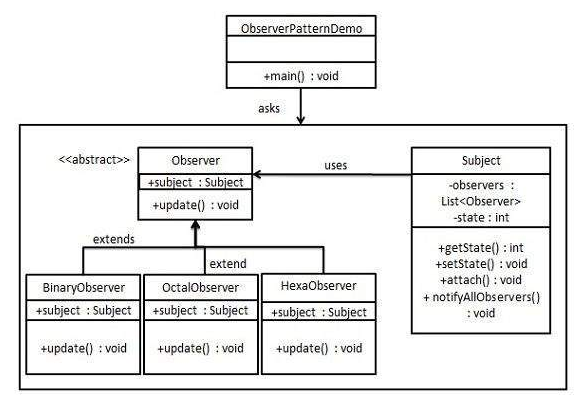
* + - * 1. Please refer to code.
      1. **Observer pattern:**
         1. Behavioral pattern.
         2. Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

Figure : Observer pattern

* + - * 1. Please refer to code.