Multithreading

Threads

- Threads are lightweight processes as the overhead of switching between threads is less
- The can be easily spawned
- The Java Virtual Machine spawns a thread when your program is run called the Main Thread

Why do we need threads?

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

Example

- Consider a simple web server
- The web server listens for request and serves it
- If the web server was not multithreaded, the requests processing would be in a queue, thus increasing the response time and also might hang the server if there was a bad request.
- By implementing in a multithreaded environment, the web server can serve multiple request simultaneously thus improving response time

Creating threads

- In java threads can be created by extending the Thread class or implementing the Runnable Interface
- It is more preferred to implement the Runnable Interface so that we can extend properties from other classes
- Implement the run() method which is the starting point for thread execution

Running threads

- Example

```
class mythread implements Runnable{
  public void run(){
        System.out.println(“Thread Started”);
  }
}

class mainclass {
  public static void main(String args[]){
        Thread  t = new Thread(new mythread()); // This is the way to instantiate a
                        thread implementing runnable interface
        t.start(); // starts the thread by running the run method
        }
}
```

- Calling t.run() does not start a thread, it is just a simple method call.
- Creating an object does not create a thread, calling start() method creates the thread.

Examples

Extending thread

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String a[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output : thread is running...

Implementing Runnable

```
class Multi implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output : thread is running...

Synchronization

- Synchronization is prevent data corruption
- Synchronization allows only one thread to perform an operation on a object at a time.
- If multiple threads require an access to an object, synchronization helps in maintaining consistency.

Example

```
public class Counter{
    private int count = 0;
    public int getCount(){
        return count;
    }

    public setCount(int count){
        this.count = count;
    }
}
```

- In this example, the counter tells how many an access has been made.
- If a thread is accessing setCount and updating count and another thread is accessing getCount at the same time, there will be inconsistency in the value of count.

Fixing the example

```
public class Counter{
    private static int count = 0;
    public synchronized int getCount(){
        return count;
    }

    public synchoronized setCount(int count){
        this.count = count;
    }
}
```

- By adding the synchronized keyword we make sure that when one thread is in the setCount method the other threads are all in waiting state.
- The synchronized keyword places a lock on the object, and hence locks all the other methods which have the keyword synchronized. The lock does not lock the methods without the keyword synchronized and hence they are open to access by other threads.

What about static methods?

```
public class Counter{
    private int count = 0;
    public static synchronized int getCount(){
       return count;
    }

    public static synchronized setCount(int count){
        this.count = count;
    }
}
```

- In this example the methods are static and hence are associated with the class object and not the instance.
- Hence the lock is placed on the class object that is, Counter.class object and not on the object itself. Any other non static synchronized methods are still available for access by other threads.

Common Synchronization mistake

```
public class Counter{
      private int count = 0;
      public static synchronized int getCount(){
          return count;
      }

      public synchronized setCount(int count){
           this.count = count;
      }
}
```

- The common mistake here is one method is static synchronized and another method is non static synchronized.
- This makes a difference as locks are placed on two different objects. The class object and the instance and hence two different threads can access the methods simultaneously.

Object locking

- The object can be explicitly locked in this way

```
synchronized(myInstance){
try{
wait();
}catch(InterruptedException ex){

}
      System.out.println("Iam in this ");
      notifyAll();
}
```

- The synchronized keyword locks the object. The wait keyword waits for the lock to be acquired, if the object was already locked by another thread. Notifyall() notifies other threads that the lock is about to be released by the current thread.
- Another method notify() is available for use, which wakes up only the next thread which is in queue for the object, notifyall() wakes up all the threads and transfers the lock to another thread having the highest priority.