Threads and Threading in Java

Java is built to be able to handle threads and multithreading. All Java programs contain at least one thread, called the main thread. It is possible to create a second (in fact, an arbitrary number of) threads in code. To do so, one of two things must be done to create a thread object. Two examples are given by Oracle:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

And:

```
public class HelloThread extends Thread {


    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

In other words, we must either create an class that implements the Runnable interface (taking care to implement the `run()` method of Runnable) then pass an instance of that class to the thread constructor, calling `.start()` on the new Thread object:

```
(new Thread(new HelloRunnable())).start();
```

Otherwise, we must create a subclass of thread, overriding the `run()` method thereof, then calling start on an instance of that subclass:

```
(new HelloThread()).start();
```

Once `.start()` is called, a thread will continue to execute until it either returns from run(), encounters an exception, or the deprecated `.stop()` is called on the thread. Additionally, note that one should not call a Thread object's `.run()` method directly, since this would simply execute the runnable code of the object as a standard method, rather than spawning a new thread to simultaneously execute the code contained in `.run()` (which calling `.start()` would accomplish).

Additionally, if we wanted the current thread to halt it's execution until another thread had finished, we could call `otherThread.join()`, which would not return until `otherThread` had finished executing or been stopped.

An interesting condition arises when we want to access the same object from multiple threads. We must ensure that the object is "thread-safe", that is, that unexpected behavior will not be caused by simultaneous access to the same fields in the object. Typically, this would be accomplished by adding the `synchronized` keyword to a method definition as follows:

```
public synchronized int incrementAndReturn() {
        field++;
        return field;
}
```

This ensures that access to this method is "locked", so that only a single thread at a time can access it. In this case, the chance of unexpected behavior(were the `synchronized` keyword not included) is very small due to the tiny amount of time between when field is incremented and when it is returned, but the risk is still present, and should be addressed.