MULTI-THREADING

MULTI-THREADING IN JAVA

Java provides built-in support for multi-threaded programming.

- Implementing the **Runnable** interface.
- Extending the **Thread** class.

PROVIDE A RUNNABLE OBJECT (1)

- The **Runnable** interface defines a single methos, **run**, meant to contain the code executed in the thread. The **Runnable** object is passed to the Thread constructor.

PROVIDE A RUNNABLE OBJECT (2)

```
Public class HelloRunnable implements Runnable{

    public void run(){
        System.out.println("Hello");
    }

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

SUBCLASS THREAD (1)

- The **Thread** class itself implements **Runnable**, though its **run** method does nothing. An class can subclass Thread, providing its own implementation of **run**.

SUBCLASS THREAD (2)

```
Public class HelloThread extends Thread{

    public void run(){
        System.out.println(“Hello”);
    }

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

IMPORTANT METHODS OF THE THREAD CLASS

- void start()
- void run()
- void setName()
- int getPriority
- void setPriority
- static void sleep(long)
- static void sleep(long,long)
- static void yield
- void join(long)
- boolean isAlive()

PAUSING EXECUTION WITH SLEEP

- Thread.sleep causes the current thread to suspend execution for a specified period.
- Two overloaded versions of sleep are provided: ont that specifies the sleep time to the milisecond and one that specifies the sleep time to the nanosecond. These sleep times are not guaranteed to be precise, because they are limited by the facilities provides by the underlying OS.

INTERRUPTS

- An **interrupt** is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt.

JOINS

- The **join** method allows one thread to wait for the completion of another. As with **sleep**, **join** is dependent on the OS for timing.

JOINS

- The **join** method allows one thread to wait for the completion of another. As with **sleep**, **join** is dependent on the OS for timing.

SYNCHRONIZATION

- Thread communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication in extremely effiecient, but make two kinds of errors possible: **thread interference** and **memory consistency error**. The tool needed to prevent these errors is **synchronization**.

THREAD INTERFERENCE (1)

- Interference happens when two operations, running in the different threads, but acting on the same data, interleave. This mean that two operations consist of multiple steps, and the sequences of steps overlap.

THREAD INTERFERENCE (2)

```
class Counter{
    private int c = 0;
    private void increment(){
        c++;
    }
    private void decrement(){
        c--;
    }
    public int value(){
        return c;
    }
}
```

THREAD INTERFERENCE (3)

Suppose Thread A invoke increment at about the same time Thread B invokes decrement. In the initial value of c is 0 then:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment, c = 1.
4. Thread B: Decrement, c=-1.
5. Thread A: Store result, c=1.
6. Thread B: Store result, c=-1.

A's result is lost, overwritten by B.

MEMORY CONSISTENCY ERROR

- Memory consistenct errors occurs when different threads have inconsistent views of the shared data.

SYNCHRONIZED

- The Java programming language provides two basic synchronization idioms: **synchronized methods** and **synchronized statements**.

SYNCHRONIZED METHODS (1)

- To make a method synchronized, simply add the **synchronized** keywork to its declaration.

SYNCHRONIZED METHODS (2)

- It is not possible for two invications of synchronized methods on the same object to interleave. When one thread is executung a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done with the object.

SYNCHRONIZED STATEMENTS

- Synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name){
        synchronized(this){
            lastName=name;
            nameCount++;
        }
        nameList.add(name);
}
```

VOLATILE

- Using **volatile** variables reduces the risk of memory consistency errors.