

8. Concurrency

2. Synchronization

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.
- Such communication is extremely efficient
- Errors possible: *thread interference*
- The tool needed to prevent these errors is *synchronization*.

Thread Interference Example

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

Thread Interference I

- Interference happens when two operations, running in different threads, but acting on the same data, *interleave*
- Single expression c++ can be decomposed into three steps:
 - Retrieve the current value of c.
 - Increment the retrieved value by 1.
 - Store the incremented value back in c.

Thread Interference II

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread Interference III

- Thread A's result is lost, overwritten by Thread B.
- Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Exercise: Thread Inference

Modify 511DepoSum project as follows:

- Create new method `add100(int index)` in the `ListDepo` class that adds 100.0 to the deposit with given index. Sleep the current thread for 1 sec before saving result to the deposit
- Create `ThreadTest` class implements `Runnable` interface with field `ListDepo` field. Run method of the class should invoke `add100` method
- Try to modify the same deposit from two threads using `ThreadTest` class

Exercise: Thread Inference

- See `821Unsync` project for the full text.

Synchronized Methods I

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

Synchronized Methods II

- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object
- Synchronized method exits guarantees that changes to the state of the object are visible to all threads

Exercise: Synchronization

- Modify 821Unsync project using synchronized add100 method and check result.

Constructor Synchronization

- **Constructors cannot be synchronized** — using the synchronized keyword with a constructor is a syntax error.
- Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed

Intrinsic Locks and Synchronization

- When a task wishes to execute a piece of code guarded by the **synchronized** keyword, it
 - checks to see if the lock is available
 - then acquires it,
 - executes the code
 - and releases it.

Intrinsic Locks

- If a task is in a call to one of the synchronized methods, all other tasks are blocked from entering **any of the synchronized methods** of that object until the first task returns from its call
- A static synchronized method invocation the thread acquires the intrinsic lock for the Class object associated with the class

Concurrency Class Fields

- Especially important **to make fields private** when working with concurrency
- Otherwise the synchronized keyword cannot prevent another task from accessing a field directly, and thus producing collisions

Synchronized Statements

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

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


Cooperation Between Tasks

- How to make tasks cooperate with each other, so that multiple tasks can work together to solve a problem?
- To accomplish this we use the mutex, which in this case guarantees that only one task can respond to a signal
- This eliminates any possible race conditions, which is safely implemented using the Object methods **wait()** and **notifyAll()**

wait() Method

- `wait()` allows you to wait for a change in some condition that is outside the control of the forces in the current method
- Often, this condition will be changed by another task
- You don't want to idly loop while testing the condition inside your task; this is called busy waiting, and it's usually a bad use of CPU cycles

Don't do this!

```
public void guardedJoy() {  
    // Simple loop guard. Wastes  
    // processor time. Don't do this!  
    while(!joy) { }  
    System.out.println("Joy has been achieved!");  
}
```

wait() Example (1 of 2)

```
public synchronized guardedJoy() {  
    while(!joy) {  
        try { wait(); }  
        catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and efficiency  
have been achieved!");  
}
```

notify() / notifyAll() Methods

- `wait()` suspends the task while waiting for the world to change
- Only when a `notify()` or `notifyAll()` occurs - suggesting that something of interest may have happened - does the task wake up and check for changes
- Thus, `wait()` provides a way to synchronize activities between tasks

wait() Example (2 of 2)

```
public synchronized notifyJoy() {  
    joy = true;  
    notifyAll();  
}
```

Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other

Manuals

- <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>