Lecture 13

Note Title 2/21/2008

Java Concurrency in Practice, Brian Goetz, Addison Wesley 2006

Basic rules for using locks:

- Every shared, mutable variable should be protected by exactly one lock.
- If an invariant property depends on more than one variable all variables must be protected by the same lock.
- Avoid excessive locking

(2) Synchronized m () { ——— ; cache result for future } ← long functional computation

m () { ——— Synchronized (this) } cache result } }

(1) locks held over I/O operations

Visibility rules

- For unsynchronized variable access only guarantee is that a thread sees its own changes in the order that it makes them.
  - for concurrent programmers - assume nothing

- synchronized order and visibility

- volatile variables are:
  - not cached
  - must be flushed on every write
  - a read always returns the most written value (from any thread)

Furthermore:

- write to a volatile variable V from thread A ensures that thread B sees all values that A had seen before the write when B reads V. of variables.

A

volatile bool v;

y = x

v = true

y = z

v = v + 1

B

if (v) { --- y ... }

this

C

y = w

v = true

Volatile w/o synchronized is ok if:

- writes do not depend on the current value unless the variable is only written by one thread.
- variable does not participate in an invariant with other variables.
- locking is not required for some other reason.

Definition:

Publishing — is making an object available to code outside the current scope (usually the scope in which it is created)

Storing in a global variable;
returning it from a non-private method
passing it to a method of a different class

Unintended publication is escape.

Blatant examples of publishing

```
private
~~public~~ static Foo foo;
public void init() { foo = new Foo(); }

public getFoo() { return foo; }
```

```
public class ThisEscapes {
    public ThisEscapes(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

Safe construction:

- An object is in a predictable state only after its constructor returns.

Pitfalls:

- Creating and starting a thread from a constructor. (ok to create the thread)
- calling an overridable method from a constructor
- Registering an object instance from constructor.

```
class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() { ... onEvent { doSomeThing... }}

    public static SafeListener newInstance(EventSource source) {

        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe
    }
}
```

"Safe" ways to avoid synchronization.

- thread confinement
- immutability

Thread Confinement —
use variable / object in only one thread.

(1) Document the rules (Ad hoc)

(2) objects only referenced from local variables cannot escape.
Stack confinement

(3) Thread Local Storage