Monitor-like Behavior

- Java provides language support for monitor lock: **synchronized** keyword

- At most one of `foo`, `bar` may run at any moment:

```
public class ThreadExample implements Runnable {
    private int var1;   // instance variables
    private int var2;

    public synchronized void foo() {
      // statements accessing var1 and/or var2
    }

    public synchronized void bar() {
      // statements accessing var1 and/or var2
    }
}
```

Synchronized Keyword

- `synchronized` means: at most one of all the methods declared `synchronized` may be executing at a time

- This applies across ALL threads

- This is not the complete monitor concept—that requires “wait” and “signal” operations—but it is the “monitor lock” part of the concept

Object Lock

- Each object has one hidden **object lock** that is ...

  - Obtained upon entry to `synchronized` method
  - Dropped upon exit from `synchronized` method

- Lock is dropped no matter how method exits, e.g.,

  - By `return`
  - By throwing exception
  - By failing to catch thrown exception

Object Lock is Smart

It is OK to multiply acquire the lock:

```
public class ThreadExample implements Runnable {
    public synchronized void foo() {
        bar();
    }

    public synchronized void bar() {
        ...
    }
}
```

Smart Does Not Mean Idiot-Proof

- Suppose programmer declares `run` to be `synchronized`:

```
public class ThreadExample implements Runnable {
    public synchronized void run() {
        bar();
    }

    public synchronized void bar() {
        ...
    }
}
```

- Since thread is ALWAYS executing `run`, other threads could never invoke `bar`

Synchronized and Unsynchronized Methods, I

- Consider:

```
public class ThreadExample implements Runnable {
    public synchronized void foo() {
        bar();
    }

    public void bar() {
        ...
    }
}
```

- ANY number of threads may simultaneously invoke `bar`

- Only ONE thread may invoke `foo`

Q: What happens if `foo` calls `bar` while N other threads are executing `bar`?

Synchronized and Unsynchronized Methods, II

A: Thread enters `bar` – so that N+1 threads are executing `bar`

- All "`synchronized`" means is: "get (reentrant) object lock when entering this method & drop it when exiting"

- Even though `foo` is `synchronized` there is no restriction on `bar`

Synchronized Blocks, I

- `synchronized` keyword can be applied to units smaller than entire method

- This is a performance optimization

- Consider:

```
public synchronized void silly() {
    1. ... compute Pi to a billion digits ...
    2. ... store result in shared memory ...
}
```

- Need lock during #2 (very short) but not during #1 (very long)

Synchronized Blocks, II

- Can write as:

```
public void silly() {
    1. ... compute Pi to a billion digits ...
    synchronized(this) {
        2. ... store result in shared memory ...
    }
}
```

- Lock held only during #2

- Argument "(`this`)" specifies which object's lock to obtain

- Recall: `this` refers to current object

Synchronized Static Methods

Q: Can static methods be synchronized?

A: Yes

Q: How?

A: Each CLASS has one hidden **class lock**

- Entry into `synchronized` static method gets class lock

- Therefore: at most one of class's static methods declared `synchronized` may be executing at a time

Java Memory Model

- Java has a **memory model**

- See

`www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`

- Memory model specifies required behavior of underlying memory (as implemented by hardware and JVM)

- Mostly relevant to caching & multiprocessors, esp. NUMAs

- One aspect of Java memory model: loads & stores of any one-word variable must be atomic (except `long` and `double`, which are each 64 bits)

Another Way to Synchronize: Volatile

- You can declare an ATOMIC variable to be "`volatile`"

- `volatile` variable will not be cached, meaning:

  - Current value always read from memory
  - New value always written to memory

- `volatile` does not apply to non-atomic types: `long`, `double`

Volatile: What It's Good For

- If one method assigns to `int` and another reads `int`, can declare `int` as `volatile` and avoid using `synchronized`

- Use of `volatile` makes this class thread-safe without "`synchronized`" keyword:

```
public class ThreadExample implements Runnable {
    private volatile int x;

    public void set(int arg) {
        x = arg;   // atomic write to int x
    }

    public int get() {
        return x;  // atomic read of int x
    }
}
```

Volatile:
What It's Bad For, I

- My opinion: `volatile` is dangerous – don't use it!

- Very easy to misuse

- For example, this code is WRONG – does not provide synchronized access:

```
private volatile int foo;  // declaration
    ...
foo++;                    // use in some method
```

Q: Do you understand why?

Volatile: What It's Bad For, II

A: "`foo++`" compiles to instructions that load AND store `foo`, though it is easy to think of it as only a store

- This also is wrong:

```
private volatile int[] foo;
     ...
foo[4] = 17;
```

- Reason: "`foo`" alone — the REFERENCE to the array — is volatile

- Individual array elements are NOT made volatile by the declaration

Aside: Volatile Keyword

- C, C++, Java, and C# all have the "`volatile`" keyword

- Keyword has slightly different meaning in all languages!

- More danger: you may think you know what `volatile` means but you are mistaken