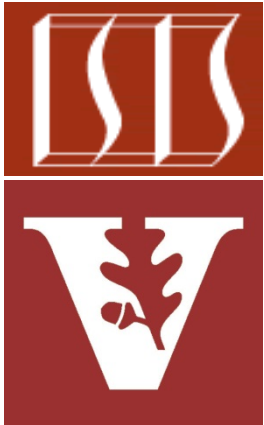


Example Application of Java Volatile Variables



Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

**Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Part of the Lesson

- Understand how Java volatile variables provide concurrent programs with thread-safe mechanisms to read from & write to single variables
- Know how to use a Java volatile variable in practice

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

Using a Java Volatile Variable in Practice

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes

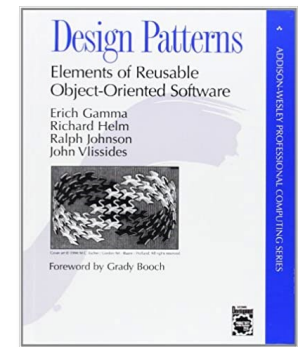
```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
    Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern



```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

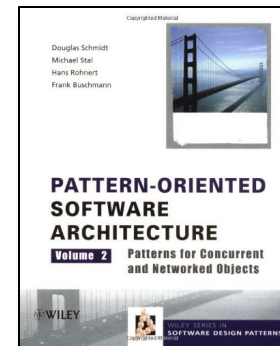


See en.wikipedia.org/wiki/Singleton_pattern

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```



See en.wikipedia.org/wiki/Double-checked_locking

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

Reduces locking overhead via "lazy initialization" in a multi-threaded environment

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

See en.wikipedia.org/wiki/Lazy_initialization

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

Ensures just the right amount of synchronization

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```



Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

Only synchronizes when sInst is null, i.e., the "first time in"

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

Note there are two checks for null (i.e., the "double-check")

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

Using a Java Volatile Variable in Practice

- Volatile is relatively simple & efficient means to ensure atomic reads & writes
- e.g., it can be used to apply the *Double-Checked Locking* pattern to the *Singleton* pattern

*No synchronization
after sInst is created*

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

Using a Java Volatile Variable in Practice

- Volatile is limited to a single read or write operation

```
class Singleton {  
    private static volatile  
        Singleton sInst = null;  
    public static  
        Singleton instance() {  
        Singleton result = sInst;  
        if (result == null) {  
            synchronized(Singleton.class)  
            {  
                result = sInst;  
                if (result == null)  
                    sInst = result =  
                        new Singleton();  
            }  
        }  
        return result;  
    }  
    ...  
}
```

*Volatile read
operation*

*Volatile write
operation*

End of Example Application
of Java Volatile Variables