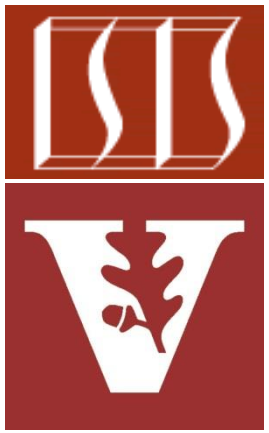
Java Volatile Variables: Example Application



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

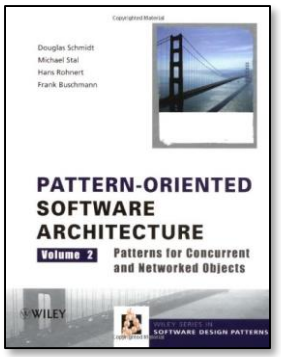
- Volatile is relatively simple & efficient means to ensure atomic reads & writes

```
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 implement the *Double-Checked Locking* 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 implement the *Double-Checked Locking* 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;
  ...
```

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

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 implement the *Double-Checked Locking* 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;
...
```

Ensures just the right amount of synchronization

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 implement the *Double-Checked Locking* 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;
  ...
```

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

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 implement the *Double-Checked Locking* 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;
  ...
```

No synchronization after sInst is created

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 Volatile Variables: Example Application