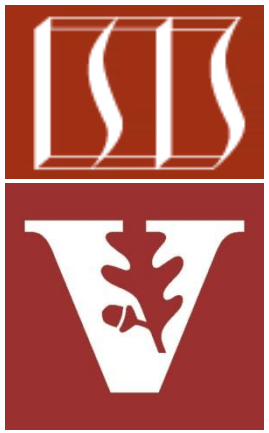
Java Volatile Variables: Usage Considerations



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
- Appreciate usage considerations for Java volatile variables

Usage Considerations for Volatile Variables

Usage Considerations for Volatile Variables

- Concurrent apps should use volatile variables carefully to avoid "busy waiting"

```
class LoopMayNeverEnd {
  volatile boolean mDone = false;

  void work() {
    // Thread T2 read
    while (!mDone) {
      // do work
    }
  }

  void stopWork() {
    // Thread T1 write
    mDone = true;
  }

  ...
}
```

If "do work" isn't time consuming this loop will spin excessively..

See en.wikipedia.org/wiki/Busy_waiting

Usage Considerations for Volatile Variables

- Concurrent apps should use volatile variables carefully to avoid "busy waiting"
  - Busy waiting is most effective when encapsulated in higher-evel concurrency libraries


```
public class AtomicLong
    ... {
  private volatile long value;
  ...
  private static final Unsafe unsafe
    = Unsafe.getUnsafe();

  private static final long
    valueOffset;

  static {
    ...
    valueOffset = unsafe.
      objectFieldOffset
        (AtomicLong
        .class
        .getDeclaredField("value"));
    ...
```

See www.youtube.com/watch?v=sq0MX3fHkro

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself

```
volatile int counter = 0;

// In Thread t1
counter++;
// load counter into register r1
// increment register r1
// store register r1 into counter



// In Thread t2
counter++;
// load counter into register r1
// increment register r1
// store register r1 into counter
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

```
volatile int counter = 0;

// In Thread t1
counter++;
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

```
volatile int counter = 0;

// In Thread t1
counter++;
// load counter into register r1
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

```
volatile int counter = 0;

// In Thread t1
counter++;
// load counter into register r1
// increment register r1
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

```
volatile int counter = 0;

// In Thread t1
counter++;
// load counter into register r1
// increment register r1
// store register r1 into counter
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

```
volatile int counter = 0;

// In Thread t1
counter++;
// load counter into register r1
// increment register r1
// store register r1 into counter


// In Thread t2
counter++;
// load counter into register r1
// increment register r1
// store register r1 into counter
```

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer

time ↓

$Thread_1$	$Thread_2$		Long value
initialized			0
read value	read value	←	0
increase value by 1	increase value by 1		0
write back	write back	→	1 or 2?

If these steps interleave in multiple threads the results may be inconsistent

See en.wikipedia.org/wiki/Write-write_conflict

Usage Considerations for Volatile Variables

- Complex operations that perform multiple instructions can't use volatile by itself, e.g.
  - Incrementing an integer
  - Use an atomic variable instead of a volatile variable

```
AtomicLong mCounter =
   new AtomicLong(0);

// In Thread t1
mCounter.getAndIncrement();
// load counter into register r1
// increment register r1
// store register r1 into counter


// In Thread t2
mCounter.getAndIncrement();
// load counter into register r1
// increment register r1
// store register r1 into counter
```

See docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html

Usage Considerations for Volatile Variables

- Declaring an array or an object as volatile only makes the *reference* volatile

```
public class Vector<E> ... {
  /**
   * The number of elements or
   * the size of the vector.
   */
  protected int elementCount;

  /**
   * The elements of the vector.
   */
  protected Object[] elementData;
  ...
}

volatile Vector v = new Vector();
```

Usage Considerations for Volatile Variables

- Declaring an array or an object as volatile only makes the *reference* volatile

```
public class Vector<E> ... {
  /**
   * The number of elements or
   * the size of the vector.
   */
  protected int elementCount;

  /**
   * The elements of the vector.
   */
  protected Object[] elementData;
  ...
}

volatile Vector v = new Vector();
```

See docs.oracle.com/javase/8/docs/api/java/util/Vector.html

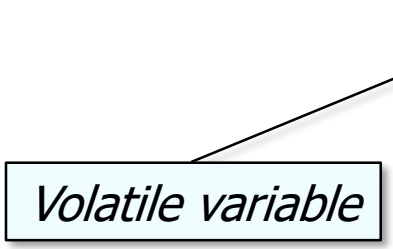
Usage Considerations for Volatile Variables

- Declaring an array or an object as volatile only makes the *reference* volatile

```
public class Vector<E> ... {
  /**
   * The number of elements or
   * the size of the vector.
   */
  protected int elementCount;

  /**
   * The elements of the vector.
   */
  protected Object[] elementData;
  ...
}

volatile Vector v = new Vector();
```



Usage Considerations for Volatile Variables

- Declaring an array or an object as volatile only makes the *reference* volatile
  - However, the contents pointed to by the reference are not volatile

```
public class Vector<E> ... {
  /**
   * The number of elements or
   * the size of the vector.
   */
  protected int elementCount;

  /**
   * The elements of the vector.
   */
  protected Object[] elementData;
  ...
}

volatile Vector v = new Vector();
```

Non-volatile fields

Usage Considerations for Volatile Variables

- Declaring an array or an object as volatile only makes the *reference* volatile
  - However, the contents pointed to by the reference are not volatile
- Therefore, more powerful types of synchronization are needed

```
public class Vector<E> ... {
  ...
  public synchronized E set
        (int location, E object) {
    if (location < elementCount){
      E result = (E)
        elementData[location];
       elementData[location] =
         object;
      return result;
    }
  ...
}

volatile Vector v = new Vector();
```

See upcoming lessons on "*Java Monitor Object*" & "*Java Synchronizers*"

Usage Considerations for Volatile Variables

- Java semantics of volatile aren't the same as in C or C++

In C and C++ [edit]

In C, and consequently C++, the `volatile` keyword was intended to[1]

- allow access to memory mapped devices
- allow uses of variables between `setjmp` and `longjmp`
- allow uses of `sig_atomic_t` variables in signal handlers.

Operations on `volatile` variables are not atomic, nor do they establish a proper happens-before relationship for threading. This is according to the relevant standards (C, C++, POSIX, WIN32),[2] and this is the matter of fact for the vast majority of current implementations. Thus, the usage of `volatile` keyword as a portable synchronization mechanism is discouraged by many C/C++ groups.[3][4][5]

Example of memory-mapped I/O in C [edit]

In this example, the code sets the value stored in `foo` to `0`. It then starts to poll that value repeatedly until it changes to `255`:

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

An optimizing compiler will notice that no other code can possibly change the value stored in `foo`, and will assume that it will remain equal to `0` at all times. The compiler will therefore replace the function body with an infinite loop similar

See www.drdobbs.com/parallel/volatile-vs-volatile/212701484

Usage Considerations for Volatile Variables

- Java semantics of volatile aren't the same as in C or C++
  - Volatiles in C/C++ aren't atomic & don't create a happens-before relationship

In C and C++ [edit]

In C, and consequently C++, the `volatile` keyword was intended to[1]

- allow access to memory mapped devices
- allow uses of variables between `setjmp` and `longjmp`
- allow uses of `sig_atomic_t` variables in signal handlers.

Operations on `volatile` variables are not atomic, nor do they establish a proper happens-before relationship for threading. This is according to the relevant standards (C, C++, POSIX, WIN32),[2] and this is the matter of fact for the vast majority of current implementations. Thus, the usage of `volatile` keyword as a portable synchronization mechanism is discouraged by many C/C++ groups.[3][4][5]

Example of memory-mapped I/O in C [edit]

In this example, the code sets the value stored in `foo` to `0`. It then starts to poll that value repeatedly until it changes to `255`:

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

An optimizing compiler will notice that no other code can possibly change the value stored in `foo`, and will assume that it will remain equal to `0` at all times. The compiler will therefore replace the function body with an infinite loop similar

See en.wikipedia.org/wiki/Volatile_variable#In_C_and_C++

Usage Considerations for Volatile Variables

- Java semantics of volatile aren't the same as in C or C++
  - Volatiles in C/C++ aren't atomic & don't create a happens-before relationship
  - They largely just disable compiler optimizations

In C and C++ [edit]

In C, and consequently C++, the `volatile` keyword was intended to[1]

- allow access to memory mapped devices
- allow uses of variables between `setjmp` and `longjmp`
- allow uses of `sig_atomic_t` variables in signal handlers.

Operations on `volatile` variables are not atomic, nor do they establish a proper happens-before relationship for threading. This is according to the relevant standards (C, C++, POSIX, WIN32),[2] and this is the matter of fact for the vast majority of current implementations. Thus, the usage of `volatile` keyword as a portable synchronization mechanism is discouraged by many C/C++ groups.[3][4][5]

Example of memory-mapped I/O in C [edit]

In this example, the code sets the value stored in `foo` to `0`. It then starts to poll that value repeatedly until it changes to `255`:

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

An optimizing compiler will notice that no other code can possibly change the value stored in `foo`, and will assume that it will remain equal to `0` at all times. The compiler will therefore replace the function body with an infinite loop similar

End of Java Volatile Variables: Usage Considerations