Java Volatile Variables: Introduction

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

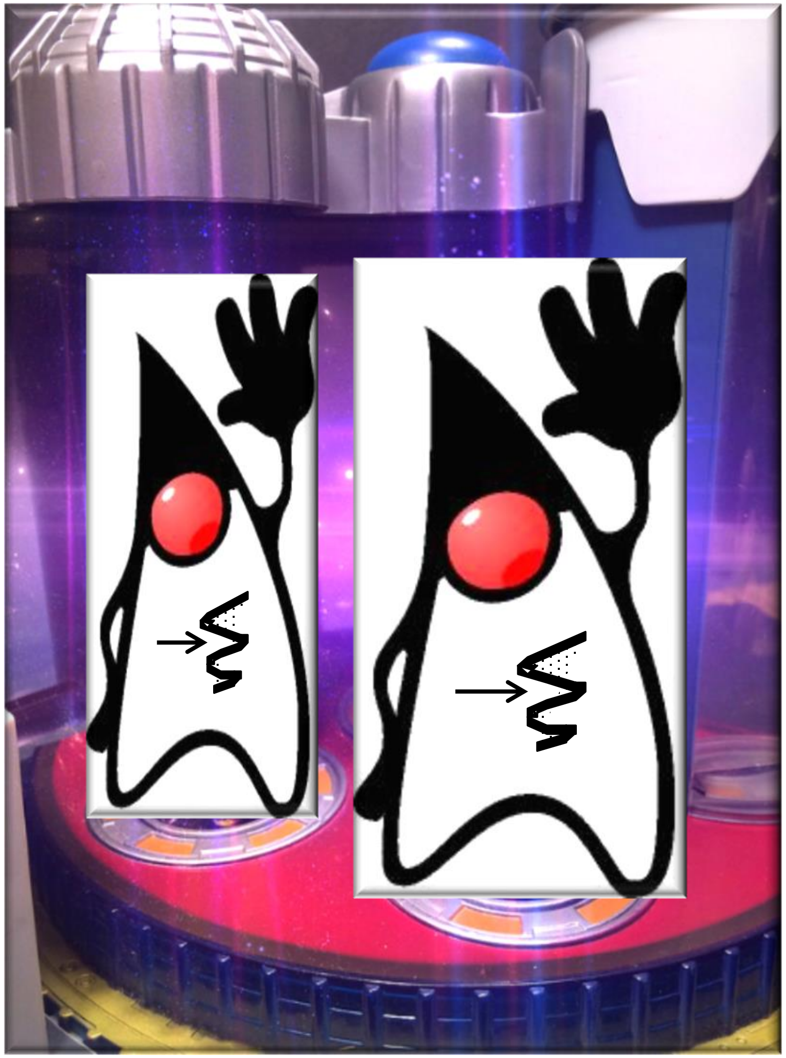
Institute for Software Integrated Systems

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



Overview of Java Volatile Variables

Overview of Java Volatile Variables

- When a concurrent program is not written correctly, the errors tend to fall into three categories: *atomicity*, *visibility*, or *ordering*

See earlier lesson on "*Overview of Atomic Operations*"

Overview of Java Volatile Variables

- Volatile ensures that changes to a variable are always consistent & visible to other threads atomically

See tutorials.jenkov.com/java-concurrency/volatile.html

Overview of Java Volatile Variables

- Volatile ensures that changes to a variable are always consistent & visible to other threads atomically
  - Reads & writes go directly to main memory (not registers/cache) to avoid *read/write conflicts* on Java fields storing shared mutable data

Overview of Java Volatile Variables

- Volatile ensures that changes to a variable are always consistent & visible to other threads atomically
  - Reads & writes go directly to main memory (not registers/cache) to avoid *read/write conflicts* on Java fields storing shared mutable data
  - Volatile reads/writes cannot be reordered

Overview of Java Volatile Variables

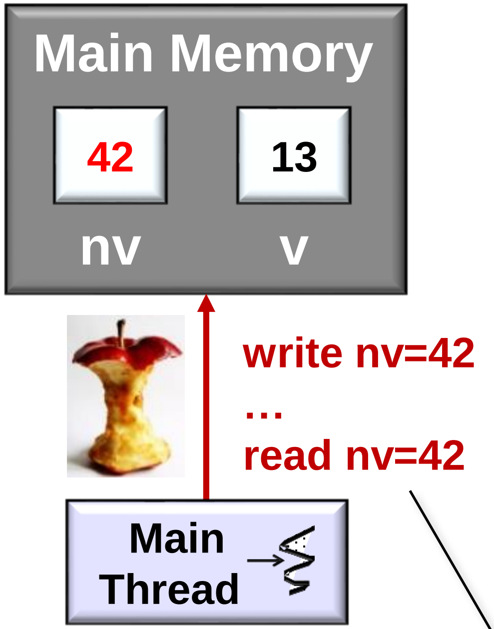
- Volatile ensures that changes to a variable are always consistent & visible to other threads atomically
  - Reads & writes go directly to main memory (not registers/cache) to avoid *read/write conflicts* on Java fields storing shared mutable data
  - Volatile reads/writes cannot be reordered
    - The Java compiler automatically transforms reads & writes on a volatile variable into atomic acquire & release pairs

Overview of Java Volatile Variables

- Volatile is *not* needed in sequential programs for several reasons

Overview of Java Volatile Variables

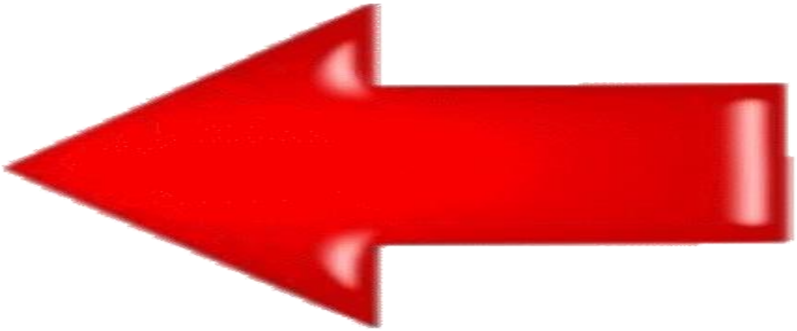
- Volatile is *not* needed in sequential programs for several reasons, e.g.
  - Reads & writes of (most) Java primitive variables are atomic



If the main thread writes a value to a non-volatile (nv) field the next read of that field will get that value

Overview of Java Volatile Variables

- Volatile is *not* needed in sequential programs for several reasons, e.g.
  - Reads & writes of (most) Java primitive variables are atomic
  - Although multiple-step operations are performed at the machine code level for variables of types long & double, these operations aren't interleaved in a single-threaded program



See docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7

Overview of Java Volatile Variables

- Volatile *is* needed in concurrent Java programs

Volatile variable

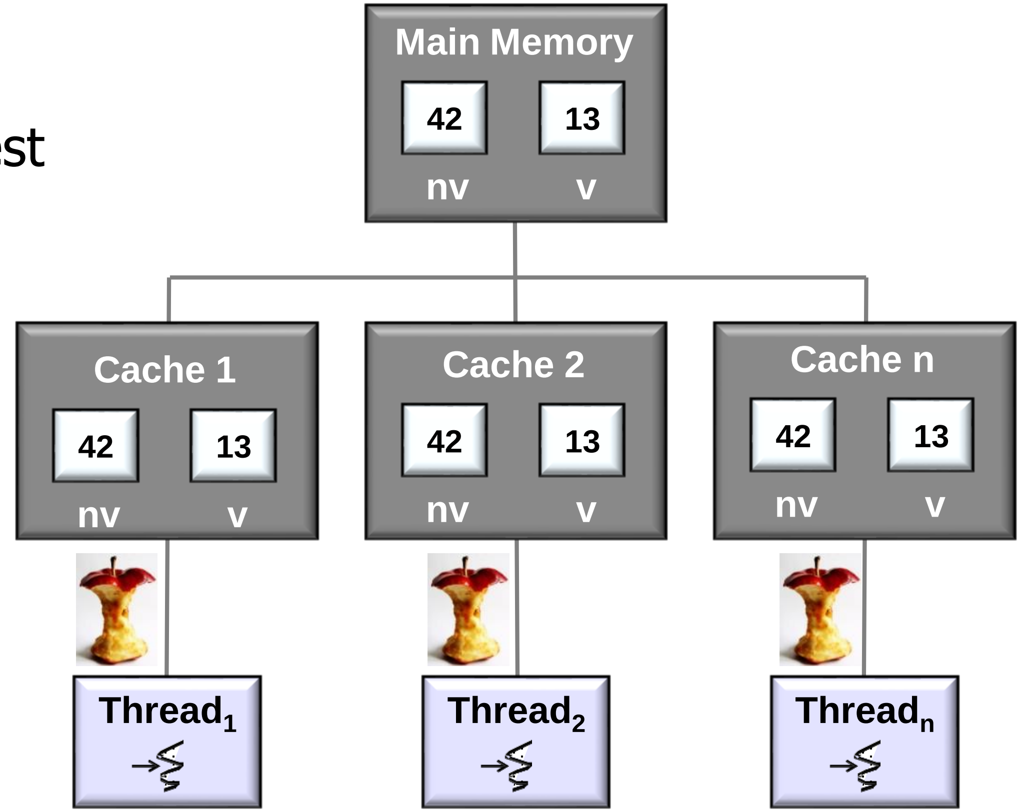
In computer programming, particularly in the C, C++, C#, and Java programming languages, a variable or object declared with the **volatile** keyword usually has special properties related to optimization and/or threading. Generally speaking, the **volatile** keyword is intended to prevent the compiler from applying certain optimizations which it might have otherwise applied because ordinarily it is assumed variables cannot change value "on their own."

The actual definition and applicability of the **volatile** keyword is often misconstrued in the context of the C language. Although C++, C#, and Java share the same keyword *volatile* from C, there is a great deal of difference between the semantics and usefulness of **volatile** in each of these programming languages.

See en.wikipedia.org/wiki/Volatile_variable

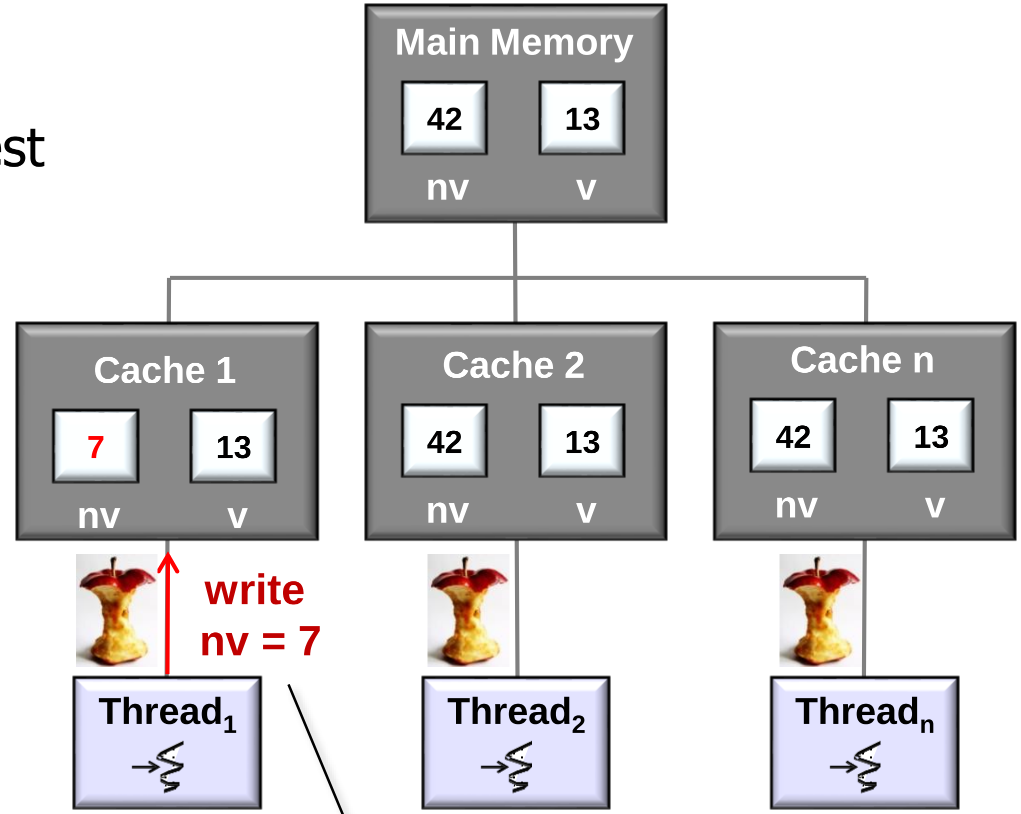
Overview of Java Volatile Variables

- Volatile *is* needed in concurrent Java programs
  - One thread may not see the latest value of a variable changed by another thread due to caching



Overview of Java Volatile Variables

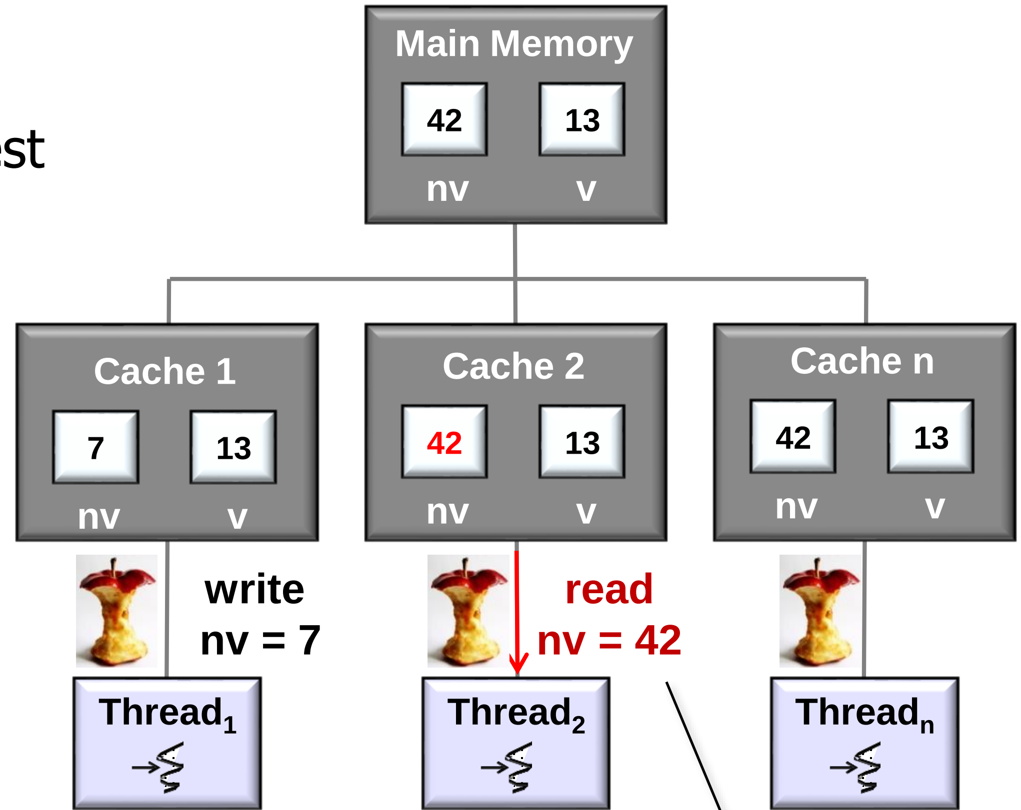
- Volatile *is* needed in concurrent Java programs
  - One thread may not see the latest value of a variable changed by another thread due to caching



Thread$_1$ writes a value to a non-volatile field **`nv`**, *which is cached locally in the core for efficiency*

Overview of Java Volatile Variables

- Volatile *is* needed in concurrent Java programs
  - One thread may not see the latest value of a variable changed by another thread due to caching



When Thread$_2$ subsequently reads the value of field **`nv`** *it gets a different result due to caching*

Overview of Java Volatile Variables

- Java defines the volatile keyword to address these problems

In Java [edit]

The Java programming language also has the `volatile` keyword, but it is used for a somewhat different purpose. When applied to a field, the Java qualifier `volatile` guarantees that:

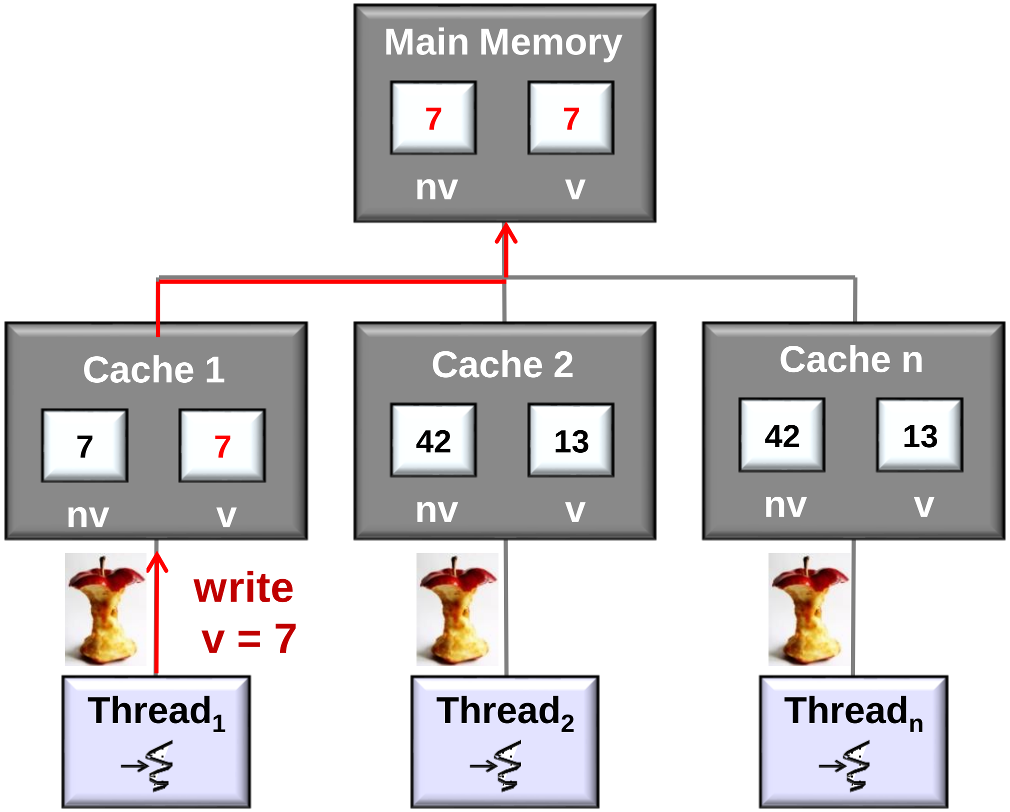
- In all versions of Java, there is a global ordering on the reads and writes to a volatile variable. This implies that every thread accessing a volatile field will read its current value before continuing, instead of (potentially) using a cached value. (However, there is no guarantee about the relative ordering of volatile reads and writes with regular reads and writes, meaning that it's generally not a useful threading construct.)
- In Java 5 or later, volatile reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.[9]

Using `volatile` may be faster than a lock, but it will not work in some situations.[citation needed] The range of situations in which volatile is effective was expanded in Java 5; in particular, double-checked locking now works correctly.[10]

See en.wikipedia.org/wiki/Volatile_variable#In_Java

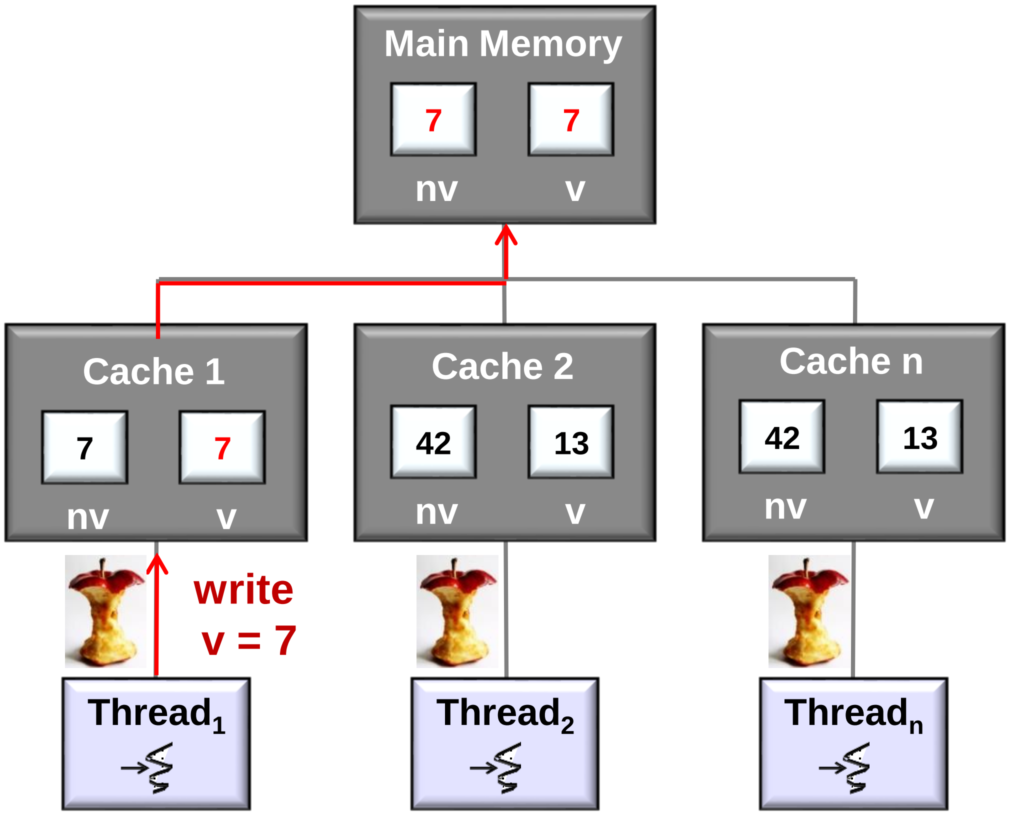
Overview of Java Volatile Variables

- Java defines the volatile keyword to address these problems
  - A value written to a volatile variable will *always* be stored in main memory



Overview of Java Volatile Variables

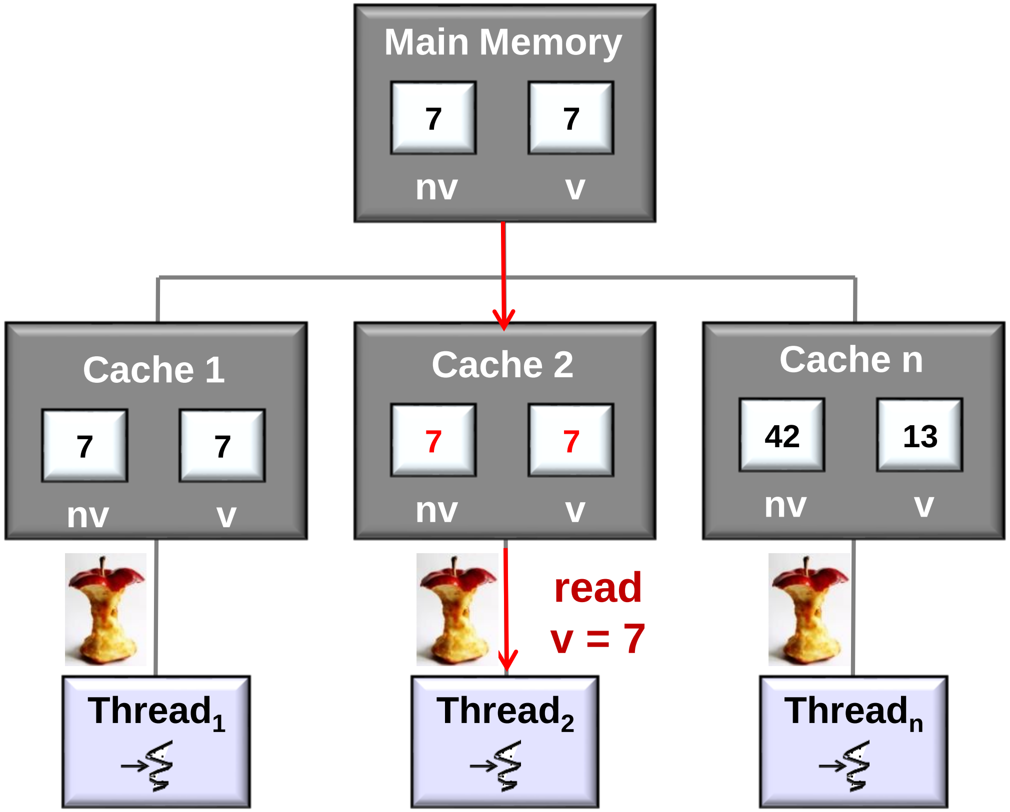
- Java defines the volatile keyword to address these problems
  - A value written to a volatile variable will *always* be stored in main memory
    - A volatile write "happens-before" all following reads of the same variable



See en.wikipedia.org/wiki/Happened-before

Overview of Java Volatile Variables

- Java defines the volatile keyword to address these problems
  - A value written to a volatile variable will *always* be stored in main memory
  - An access to a volatile variable will be read from main memory



volatile reads are cheap & volatile writes are cheaper than synchronized statements

Overview of Java Volatile Variables

- Volatile guarantees *atomicity*

```
volatile long foo;
final long A = 0xffffffffffffffffl;
final long B = 0;

new Thread(() -> {
  for (int i;; i++) {
    foo = i % 2 == 0 ? A : B;
}}).start();

new Thread(() -> {
  long fooRead = foo;
  if (fooRead != A && fooRead != B)
    System.err.println
     ("foo incomplete write "
      + Long.toHexString(fooRead));
 }).start();
```

If volatile is removed here then incomplete writes may occur (especially on 32 bit machines)

See stackoverflow.com/questions/3038203/is-there-any-point-in-using-a-volatile-long

Overview of Java Volatile Variables

- Volatile guarantees *atomicity*
  - Reads & writes are *atomic* for all variables declared volatile

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as `c++`, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using `volatile` variables reduces the risk of memory consistency errors, because any write to a `volatile` variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a `volatile` variable are always visible to other threads. What's more, it also means that when a thread reads a `volatile` variable, it sees not just the latest change to the `volatile`, but also the side effects of the code that led up the change.

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

Overview of Java Volatile Variables

- Volatile guarantees *atomicity*
  - Reads & writes are *atomic* for all variables declared volatile
  - Reads & writes are *always* atomic for Java references

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as `c++`, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using `volatile` variables reduces the risk of memory consistency errors, because any write to a `volatile` variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a `volatile` variable are always visible to other threads. What's more, it also means that when a thread reads a `volatile` variable, it sees not just the latest change to the `volatile`, but also the side effects of the code that led up the change.

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

Overview of Java Volatile Variables

- Volatile guarantees *visibility*

```
public class MyRunnable
                 implements Runnable {
  private volatile boolean
                 mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while (mIsStopped != true) {
      // a long-running operation
    }
    ...
```

See jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html

Overview of Java Volatile Variables

- Volatile guarantees *visibility*
  - If an action in thread T1 is *visible* to thread T2, the result of that action can be observed by thread T2

```
public class MyRunnable
                    implements Runnable {
  private volatile boolean
                      mIsStopped = false;

  public void stopMe() {
    mIsStopped = true; // T1 write
  }

  public void run() { // T2 read
    while (mIsStopped != true) {
      // a long-running operation
    }
    ...
```

volatile write is visible to "happens-after" reads

Overview of Java Volatile Variables

- Volatile guarantees *ordering*

```
public class MyRunnable
                implements Runnable {
  private volatile boolean
                  mIsStopped = false;

  public void stopMe() {
    mIsStopped = true;
  }

  public void run() {
    while (mIsStopped != true) {
      // a long-running operation
    }
    ...
```

See jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html

Overview of Java Volatile Variables

- Volatile guarantees *ordering*
  - *Ordering* constraints describe what order operations are seen to occur in different threads

```
public class MyRunnable
                  implements Runnable {
  private volatile boolean
                     mIsStopped = false;

  public void stopMe() {
    mIsStopped = true; // T1 write
  }

  public void run() { // T2 read
    while (mIsStopped != true) {
      // a long-running operation
    }
    ...
```

The write to **`mIsStopped`** in **`T1`** must happen-before the **`T2`** read completes

Overview of Java Volatile Variables

- Incrementing a volatile is *not* atomic

Thread$_1$	Thread$_2$		volatile value
initialized			0
read value		←	0
	read value	←	0
increase value by 2			0
	increase value by 1		0
write back	write back	→	2 or 1?

Overview of Java Volatile Variables

- Incrementing a volatile is *not* atomic
  - If multiple threads try to increment a volatile at the same time, an update might get lost

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

Consider using the java.util.concurrent.atomic package, which supports atomic increment/decrement & compare-and-swap (CAS) operations

End of Volatile Variables: Introduction