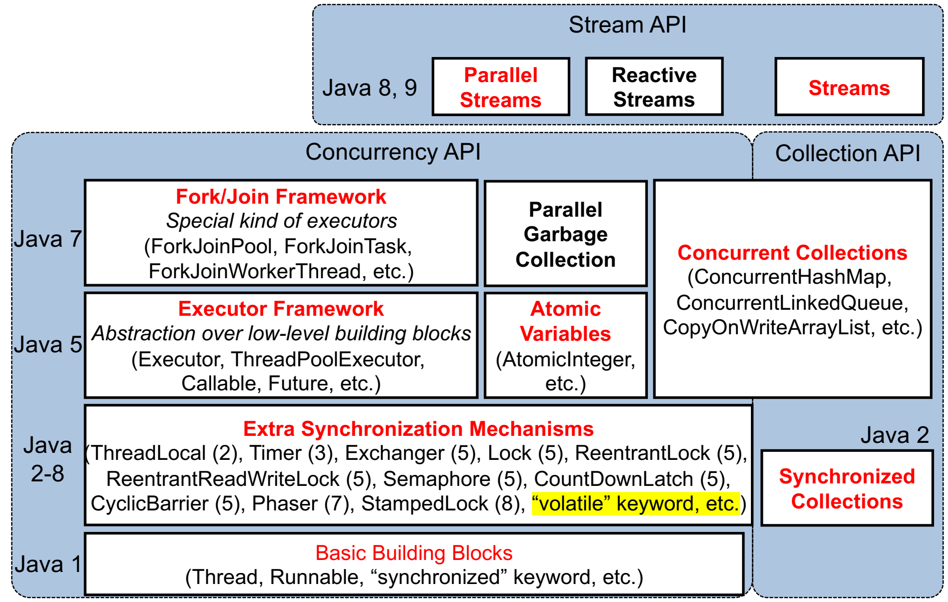
Volatile Variables in Java

Concurrency API in Java



What is the "volatile" Keyword?

- You must do thread synchronization (locking) to share a variable among threads.
- You can skip locking to share a *volatile* variable among threads <u>in some cases</u>.
  - No locking means…
    - No worry about thread safety issues
      - Particularly, race conditions
    - Less overhead (better performance)
      - Locking consumes some time and resources.

Recap: Race Condition

- Explicit thread termination

```
boolean done = false;

public void setDone(){
  done = true;
}

public void run(){
  while( true )
    if( done ) break;
    // Do some work
}
```

- A thread ("A")has read the current value of "done" but has not checked if it is true or false.
- A context switch occurs.
- A race condition occurs if
  - Another thread ("B") calls setDone() to assign true to "done."
  - This state update is not visible for Thread "A".
    - It still believes done==false and does not immediately break out from the loop.

Thread-safe Code

- Without "volatile"

```
• boolean done = false;
• ReentrantLock lock =
      new ReentrantLock();

• public void setDone(){
   lock.lock();
   done = true;
   lock.unlock();
  }

• public void run(){
    while( true ){
        lock.lock();
        if( done ) break;
        // Do some work
        lock.unlock(); }
  }
```

- With "volatile"

```
• volatile boolean done = false;

• public void setDone(){
    done = true;
  }

• public void run(){
    while( true )
      if( done ) break;
      // Do some work
  }
```

- Both versions are thread-safe.
- "volatile" makes code simpler and less error-prone.

- With "volatile"

```
volatile boolean done = false;

public void setDone(){
  done = true;
}

public void run(){
  while( true )
    if( done ) break;
    // Do some work
}
```

- A context switch can occur in b/w
  - Reading the value of "done"
  - Judging if it is true or not.
- The most up-to-date value of "done" is loaded from the memory in the 2nd step
  - In case another thread has called setDone() in between the 2 steps
- No race conditions occur.
  - "Volatile" makes a state change on a shared variable visible for threads.

- With "volatile"

```
volatile boolean done = false;

public void setDone(){
  done = true;
}

public void run(){
  while( true )
    if( done ) break;
    // Do some work
}
```

- A context switch can occur in between
  - Loading the true value
  - Assigning it to "done"
- All threads will assign true to "done."
  - There are no other possible state changes.
- Threads do not generate race conditions.

Limited Effectiveness/Usefulness

- A "volatile" variable is guaranteed to have the most up-to-date value whenever it is read.

```
– volatile int a;
                         // These multi-step ops are all thread-safe,
  int b = a;             // even if a context switch occurs in between
  if(a==0){...}          // 2 different steps and another thread changes
  println(a);            // the value of "a" there.
```

- However, it does NOT eliminate all possible race conditions.

```
– a + 1;                 // 3 steps. Thread-safe.

  b = a + 1;             // These operations are NOT thread-safe.
  if(a+1>0)              // The first 3 steps are thread-safe though.
  println(a+1);          // The 3rd step generates an intermediate
                         // state. Race conditions can occur in b/w the
                         // 3rd and the 4th step.
```

- "volatile" is effective only in the read operations that have no intermediate state.

- "volatile" is effective only in the write operations that have no intermediate state.

```
– volatile int a;
  a = 1;              // 2 steps. A context switch can occur in
                      // between the 2 steps, but no race conditions
                      // occur here.

  a = a + 1;          // 5 steps. There is an intermediate state.
  a++;                // Not thread-safe.
```

- Use a "volatile" variable only when you can live with these limitations/constraints.
- Do NOT use "volatile" for arrays.

When to use Volatile Variables?

- Despite these limitations, when can we take advantage of volatile variables?
- ONLY when a value assignment to a shared variable does not depend on the current value.
- Any data structures that perform this kind of value assignments?
  - Probably, ***latch*** only.
    - A data structure that performs a single type of state changes
      - e.g. False → True
    - Often used to terminate threads.
      - c.f. "done" variable in prior examples
        - » The state of "done" always changes in a unidirectional way: false → true
        - » "true → false" never happen.

In Summary…

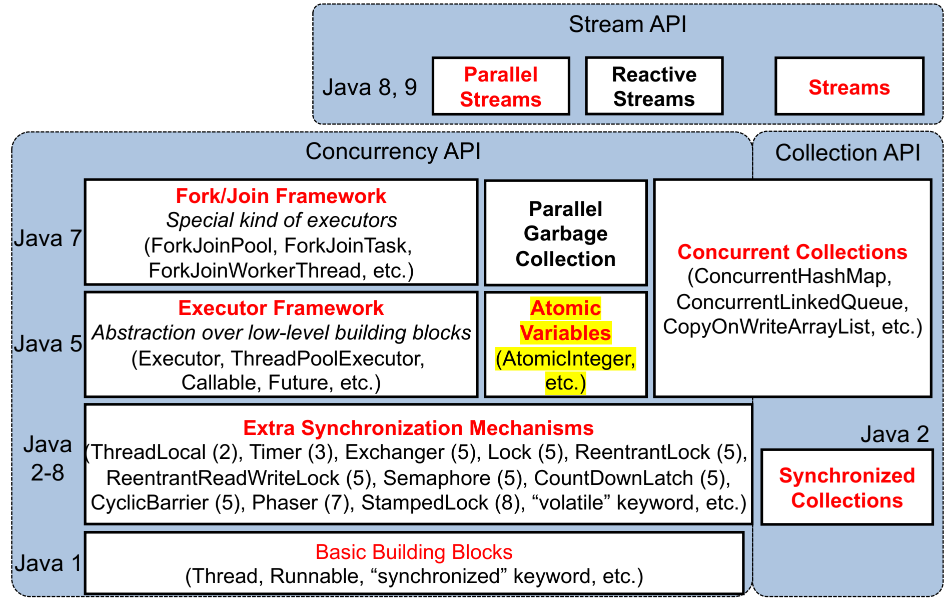
- "Volatile" is NOT a general-purpose, widely-applicable threading tool.
- Powerful only in some specific cases
  - In practice, assume it is useful only for simplifying the implementation of a latch.
    - Very useful to implement flag-based thread termination and 2-step thread termination.

Exercise (Not HW)

- Define a flag as a volatile variable in a 2-step thread termination scheme that you have implemented for a prior HW.

Atomic Data Structures in Java

Concurrency API in Java



`java.util.concurrent.atomic` Package

- Offers thread-safe classes to manipulate single variables.
  - `AtomicBoolean,`
  - `AtomicInteger, AtomicIntegerArray`
  - `AtomicLong, AtomicLongArray`
  - `AtomicReference<V>, AtomicReferenceArray<E>`
  - `DoubleAccumulator, DoubleAdder`
  - `LongAccumulator, LongAdder`
  - `...`

`java.util.concurrent.atomic` Package

- Offers thread-safe classes to manipulate single variables.
  - `AtomicBoolean,`
  - `AtomicInteger, AtomicIntegerArray`
  - `AtomicLong, AtomicLongArray`
  - `AtomicReference<V>, AtomicReferenceArray<E>`
  - `DoubleAccumulator, DoubleAdder`
  - `LongAccumulator, LongAdder`
  - `...`
- All of their public methods are thread-safe. They avoid race conditions
  - with a special CPU instruction, called Compare-and-Swap (CAS) instruction, rather than doing thread synchronization
  - Very efficient.

Atomic Variables

- Serve as "better" volatile variables.
- Offer the same memory semantics as volatile variables, but with additional support for atomic state updates.
  - get() has the memory effects of reading a volatile variable.
    - Returned value is guaranteed to be the most up-to-date whenever it is used.
  - set() has the memory effects of writing to a volatile variable.
  - *read-and-update* methods (e.g., xxxAndSet() and getAndXxx()) have the memory effects of both reading and writing volatile variables.
- Highly recommended as far as they match your use cases.

-
```
volatile boolean vFlag;
vFlag = true;                // 2 steps. Thread-safe.
if(vFlag){ int i=10; }       // Thread-safe.
if(vFlag){ vFlag = false; }  // NOT thread-safe.
                             // Read and write ops are thread-safe
                             // each, but if a context switch occurs
                             // in between the two ops, a race
                             // condition can occur.
```
-
```
AtomicBoolean atomicFlag = new AtomicBoolean(false);
atomicFlag.set(true)                     // Thread-safe
atomicFlag.compareAntSet(true, false);   // Thread-safe!
```
- `set()`: Thread-safe. Atomically returns the current value, with memory effects of reading a volatile variable.
- *read-and-update* methods (e.g., `xxxAndSet()` and `getAndXxx()`) have the memory effects of both reading and writing volatile variables.

AtomicBoolean

- Offers thread-safe methods to manipulate a single boolean value atomically (i.e., in a thread-safe manner).
  -
```
boolean flag;
if(flag)                        // 2 steps. Not thread-safe
printlne(flag);                 // 2 steps. Not thread-safe
```
  -
```
volatile boolean vFlag;
if(vFlag)                       // 2 steps. Thread-safe
println(vFlag);                 // 2 steps. Thread-safe
```
  -
```
AtomicBoolean atomicFlag = new AtomicBoolean(false);
if(atomicFlag.get())            // 2 steps. Thread-safe
println(atomicFlag.get());      // 2 steps. Thread-safe
```
  - `get()`: Thread-safe. Atomically returns the current value, with memory effects of reading a volatile variable.
    - A context switch can occur right after `get()` returns, and another thread may change the value.
    - However, `AtomicBoolean` guarantees that the returned value of `get()` will be updated and the most up-to-date, whenever it is used later on.

-
```
public final boolean compareAndSet(boolean expect,
                                   boolean update)
```
  - Atomically sets the "update" (new) value if the current value is equal to the "expect" value.
  - Returns true if successful.
  - Returns false if the current value was not equal to the "expect" value.
  - `atomicFlag.compareAndSet(true, false);     // Thread-safe`
    - Sets false if the current value is true, and returns true
    - Keeps the current value if it is false, and returns false

AtomicBoolean for Thread Termination

- **AtomicBoolean** can be used to implement a flag in flag-based and 2-step thread termination schemes

```
class CancelableRunnable
 implements Runnable {

 boolean done = false;
 ReentrantLock lock;

 public void setDone(){
  lock.lock();
  done = true;
  lock.unlock();
 }

 public void run(){
  while(true){
   lock.lock();
   if(done) break;
   lock.unlock();
   ... // do some work }}}
```

```
class CancelableRunnableWithAtomicBoolean
 implements Runnable {

 AtomicBoolean done =
   new AtomicBoolean(false);

 public void setDone(){
   done.set(true);
 }

 public void run(){
   boolean temp = false;
   while(true){
    temp = done.get();
    if(done.get()) break;
    ... // do some work
    } } }
```

Exercise (Not HW)

- Define a flag as an **AtomicBoolean** variable in a 2-step thread termination scheme that you have implemented for a prior HW.

AtomicInteger

- Offers thread-safe methods to manipulate a single integer value atomically.
  - ```
    int i = 0;
    int j = i;                    // 2 steps. Not thread-safe.
    if(i==0)                      // 3 steps. Not thread-safe.
    ```
  - ```
    volatile i = 0;
    int j = i;                    // 2 steps. Thread-safe.
    if(i==0)                      // 3 steps. Thread-safe.
    ```
  - ```
    AtomicInteger atomicInt = new AtomicInteger(0);
    int j = atomicInt.get();  // Thread-safe. j==0
    if(atomicInt.get()==0){...}  // Thread-safe.
    ```
  - `get()`: Thread-safe. Atomically returns the current value, with memory effects of reading a volatile variable.
    - A context switch can occur right after `get()` returns, and another thread may change the value.
    - However, `AtomicInteger` guarantees that the returned value of `get()` will be updated and the most up-to-date, whenever it is used later on.

- ```
  volatile i = 0;
  i = 1;                      // Thread-safe.
  int j = i + 1;              // 5 steps. NOT thread-safe.
  i++;                        // 5 steps. NOT thread-safe.
  ```
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.set(1);                      // Thread-safe.
  j = atomicInt.incrementAndGet();       // Thread-safe. j==2
  atomicInt.incrementAndGet();           // Thread-safe.
  ```
- `set()`: Thread-safe. Atomically returns the current value, with memory effects of reading a volatile variable.
- *read-and-update* methods (e.g., `xxxAndSet()`, `xxxAndGet()` and `getAndXxx()`) have the memory effects of both reading and writing volatile variables.
  ```

– `volatile i = 0;`

```
i = 1;                      // Thread-safe.
if(i==0){i = 1;}            // NOT thread-safe.
                            // Read and write ops are thread-safe
                            // each, but if a context switch occurs
                            // in between the two ops, a race
                            // condition can occur.
```

– `AtomicInteger atomicInt = new AtomicInteger(0);`

```
atomicInt.compareAndSet(0, 1);     // Thread-safe.
```

- Other *read-and-update* methods include:
  - `decrementAndGet(), addAndGet(int), getAndSet(int), ...`
  - `updateAndGet(IntUnaryOperator), accumulateAndGet(int, IntBinaryOperator)`

– `updateAndGet(IntUnaryOperator updateFunction)`

- *Atomically* updates the current integer value with the result of running a given lambda expression, and then returns the updated value.
- `IntUnaryOperator`: a general-purpose functional interface
- `updateFunction`: a lambda expression that takes the current `int` value as a parameter, updates it and returns the updated value.
  - This update logic runs atomically (i.e., in a thread-safe manner)
-
```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)->++i) );   // 11. Thread safe
atomicInt.incrementAndGet();               // 12. Thread safe
```

	Params	Returns	Example use case
UnaryOperator<T>	T	T	Logical NOT (!)

-
```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)->++i) );   // 11. Thread safe
```

- Why ++1? Just in case, note that:
  -
```
int i = 0;
i++;          // i==1
```
  -
```
int i = 0;
int x = 0;
int x = i++; // i==1, x==0
```
  -
```
int i = 0;
int y = 0;
int y = ++i; // i==1, y==1
```

-
```
volatile i = 10;
if(i > 0){i = 0;}        // NOT thread-safe.
                         // Read and write ops are thread-safe
                         // each, but if a context switch occurs
                         // in between the two ops, a race
                         // condition can occur.
```

-
```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)->{ if(i > 0){i = 0;}
                                   return i; } );

                         // Thread safe. The lambda expression
                         // is executed in a thread-safe manner.
```

– `accumulateAndGet(int, IntBinaryOperator)`
  - *Atomically* updates the current `int` value with the result of applying a given function, and then returns the updated value.

- `IntBinaryOperator`: a general-purpose functional interface

- ```
  atomicInt.accumulateAndGet(initValue,
                             (result, currentVal)-> ... );
  ```

- ```
  int result = initValue;
  result = accumulate(result, currentVal);
  ```

- Takes a lambda expression as the second parameter.
  – The body of `accumulate()` is expressed in the LE, which runs atomically
  – c.f. `Stream.reduce()`

	Params	Returns	Example use case
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)

- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.accumulateAndGet(10,
                             (result, currentVal)->
                                 (currentVal+result)/2 );
  ```

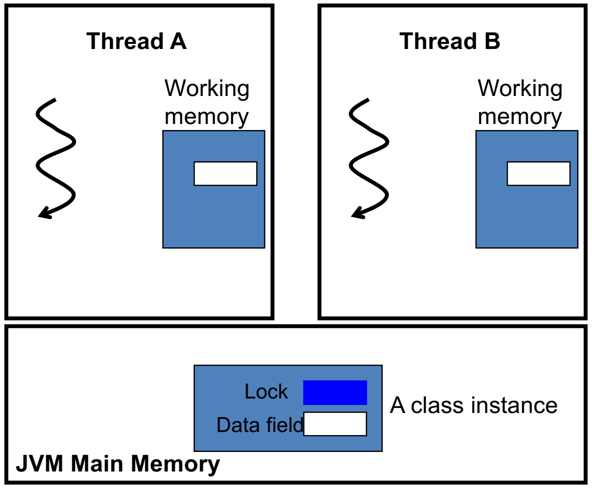
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.accumulateAndGet(
      0, (result, currentVal)->{
          if(currentVal >= result) return currentVal;
          else if return result; }));
  ```

Appendix:
The *Volatile* Keyword and JVM Memory Model

Memory Management in JVM

- A Java Virtual Machine (JVM) manages memory space for a Java program(s), following JVM Memory Model.

JVM Memory Model (Java 5~)



Main memory

Shared by all threads

Working (local) memory

Per-thread space

Not accessible/visible by the other threads

A local copy of each data field

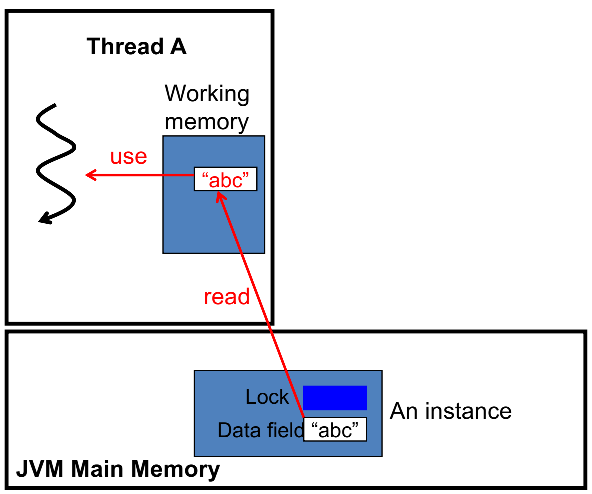
Local variables

JVM Actions

- JVM implements a set of *JVM actions* to manage the main memory space and local memory spaces.
- JVM actions (Java bytecode instructions)
  - *read*, *write*, *use*, *assign*
  - *lock*, *unlock*
  - These are all atomic.

Read Operation (Single Threaded)

```
System.out.println(variable);
```



When thread A reads a value for the first time…

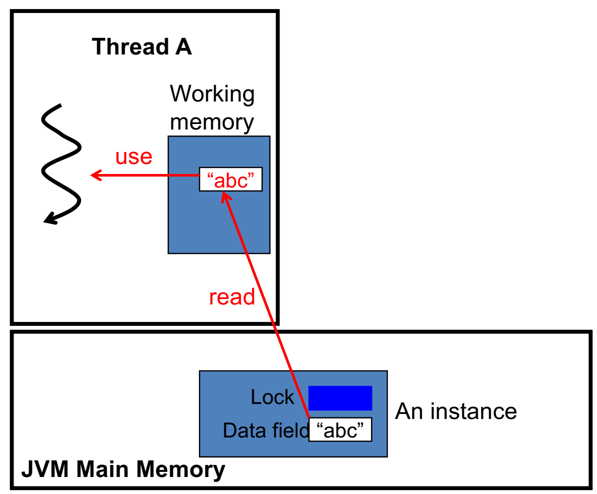
(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

Read Operation (Single Threaded)

```
if(variable.equals(...))
```



When thread A reads a value for the first time…

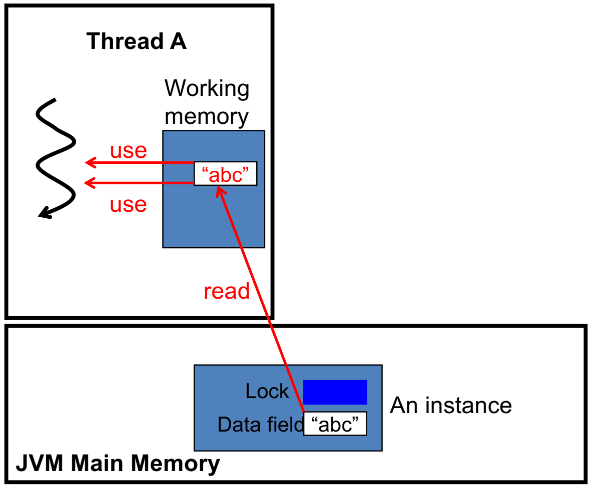
(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

Read Operation (Single Threaded)

```
if(variable.equals(...)){          // First read operation
  System.out.println(variable); } // Second read operation
```



When thread A reads a value for the first time…

(1) Read the value from the main memory

(2) Store the value in the local working memory.

Always Read-Use for the first read operation.

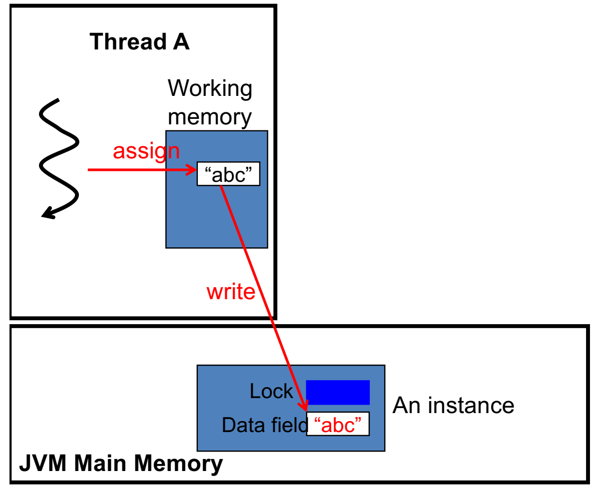
When thread A accesses the same variable later…

Use OR Read-Use

It's up to JVM implementations, but often, Use only because it's faster.

Write Operation (Single Threaded)

```
variable = "abc";
```



When thread A assigns a value to a data field…

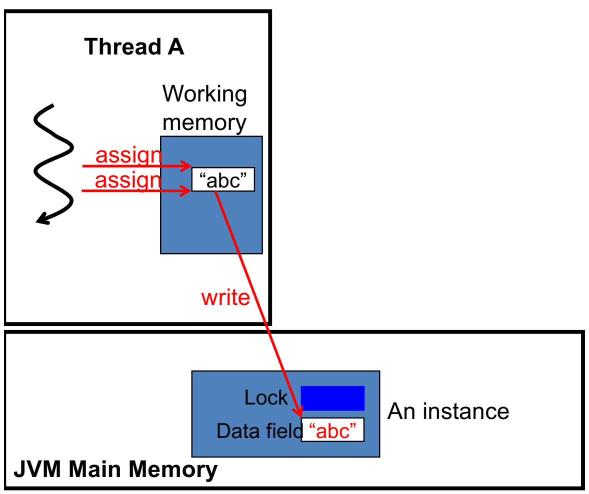
(1) Assign the value to a data field in the local memory.

(2) Copy it to the main memory.

Always Assign-Write for the first assignment operation.

Write Operation (Single Threaded)

```
variable = "abc";   // First write operation
variable = "cde";   // Second write operation
```



When thread A assigns a value to a data field…

(1) Assign the value to a data field in the local memory.

(2) Copy it to the main mem.

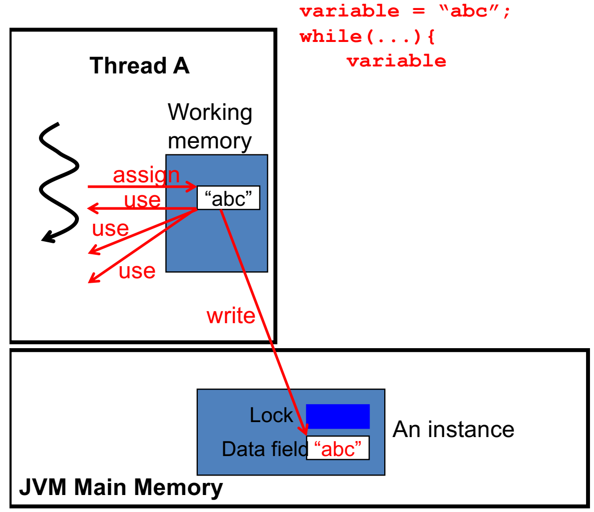
Always Assign-Write for the first assignment operation.

It is not predictable when (how soon) "write" occurs. Maybe immediately after "assign," but some interval may exist.

If thread A repeatedly assigns different values to the the same data field, it makes sense to do: Assign, Assign, … , Write

Write Operation (Single Threaded)

```
variable = "abc";
while(...){
    variable
```



When thread A assigns a value to a data field…

(1) Assign the value to a data field in the local memory.

(2) Copy it to the main mem.

Always Assign-Write for the first assignment operation.

It is not predictable when (how soon) "write" occurs. Maybe immediately after "assign," but some interval may exist.

If thread A repeatedly assigns different values to the the same data field, it makes sense to do: Assign, Assign, … , Write

If thread A often read the value, it makes sense to do: Assign, Use, Use, Use, … , Write

Thread Synchronization and Memory Synchronization

- When a thread runs atomic code with a lock held…
  - `public void setDone(){`

```
public void setDone(){
    lock.lock()
    // atomic code;
    lock.unlock();   }
```

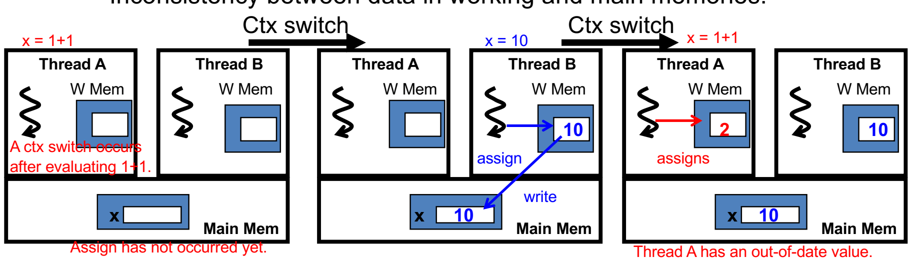
- JVM does two things:
  - Thread synchronization
    - Allows only one thread to enter and run atomic code at a time.
      - All the other threads are blocked.
  - Memory synchronization
    - Synchronizes the most up-to-date value in between a local memory and the main memory.

Race Condition

- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

Race Condition

- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

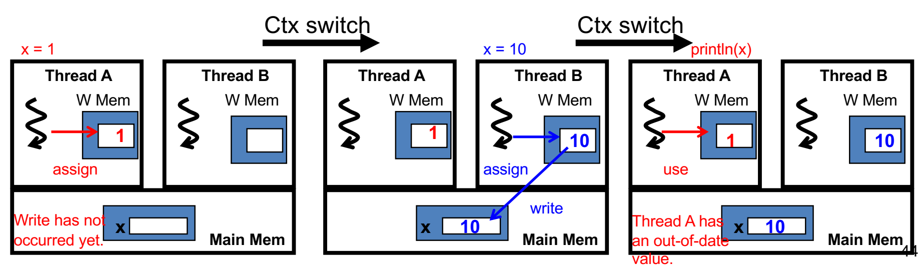


A race condition can occur due to a failure of thread synchronization.

Race Condition

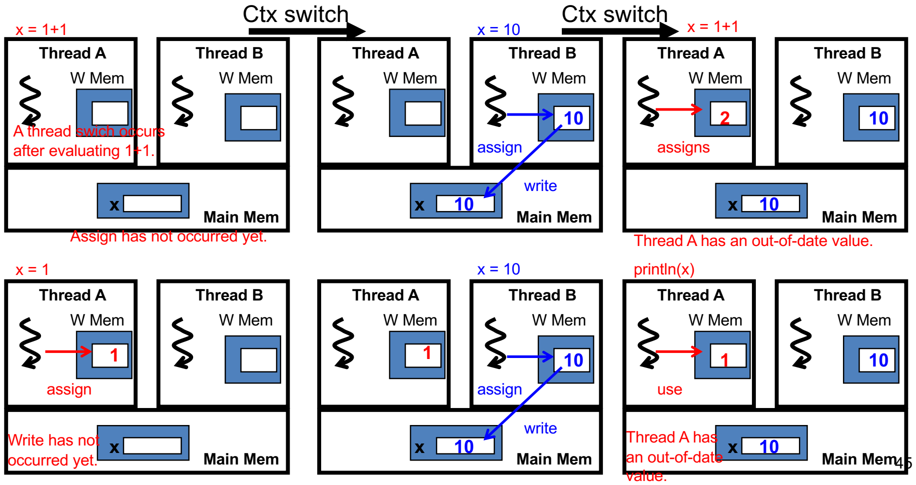
- A race condition can occur due to
  - Failure of thread synchronization and/or
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.

A race condition can occur due to a failure of memory synchronization. Threads are synchronized in this case (by chance).



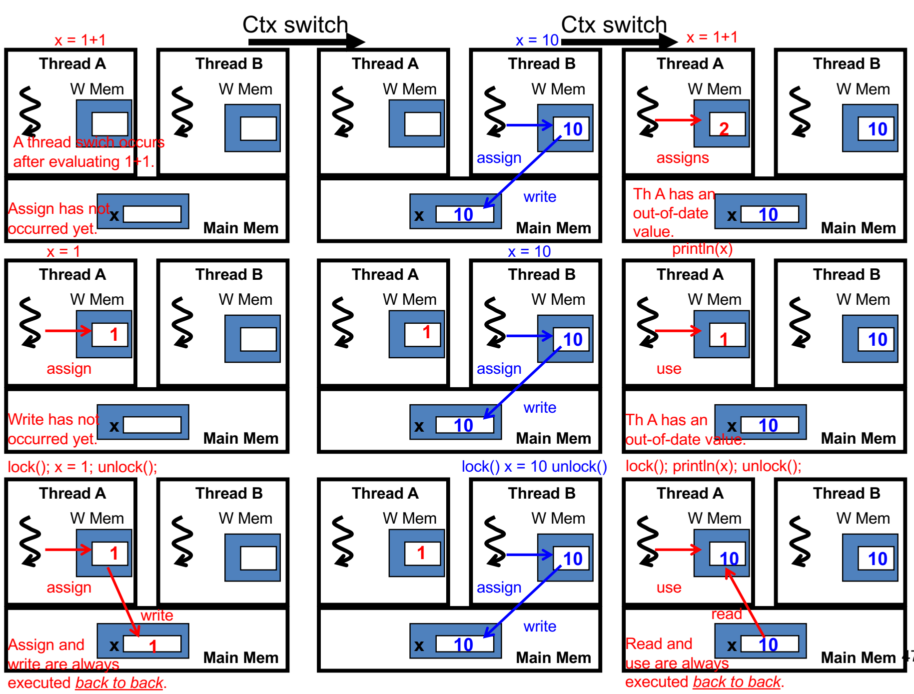
Race Condition

- A race condition can occur due to
  - Failure of thread synchronization
  - Failure of memory synchronization
    - Inconsistency between data in working and main memories.



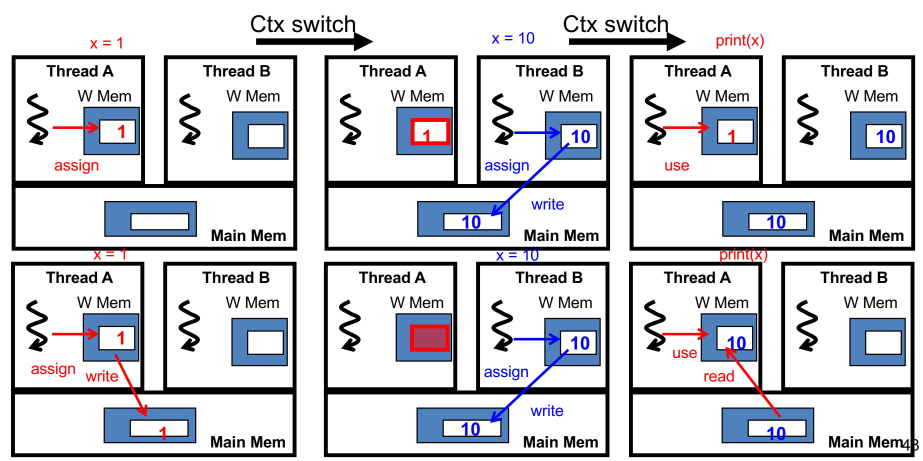
Race Conditions and `ReentrantLock`

- You need both thread and memory synchronization to prevent race conditions.
- `ReentrantLock` does both.
  - Thread synchronization
    - Allows only one thread to enter and run atomic code at a time.
      - All the other threads are blocked.
  - Memory synchronization
    - Synchronizes the most up-to-date value in between a local memory and the main memory.
      - Destroys working memory upon entering atomic code
      - Flushes working memory to the main memory upon existing atomic code

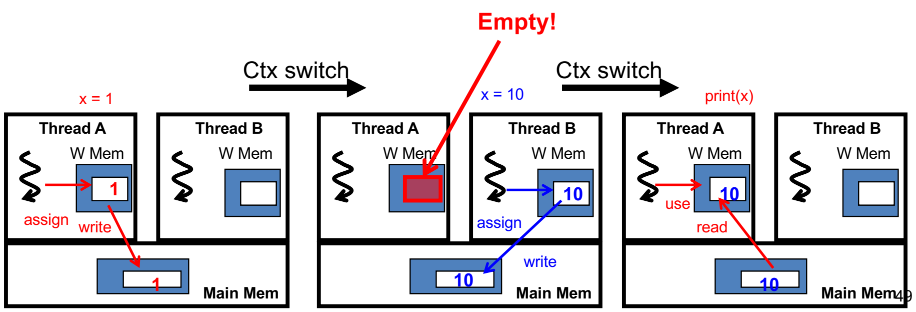


Volatile Variables

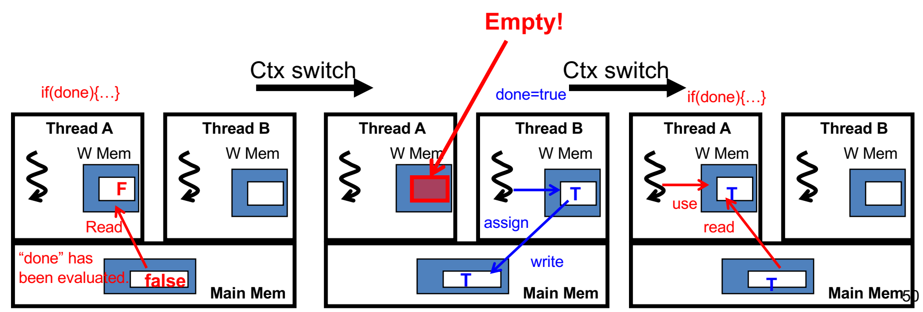
- When a thread uses a a volatile variable…
  - Memory synchronization is guaranteed.



- Write always follows Assign immediately.
  - They are always paired.
- Read always occur before Use.
  - They are always paired.
- A volatile variable's value is *NOT persistent* (*i.e., volatile*) across context switches.



- Write always follows Assign immediately.
  - They are always paired.
- Read always occur before Use.
  - They are always paired.
- A volatile variable's value is *NOT persistent* (*i.e., volatile*) across context switches.



- No need to use locking to perform memory synchronization for a volatile variable.
  - No locking → less overhead
- A volatile variable NEVER synchronizes threads that access the variable.
  - "Volatile" is not a thread sync tool. It's for a memory sync tool.
- ***The volatile keyword works when you don't need to synchronize threads but need to synchronize memory.***

When to use Volatile Variables?

- When a value to write into a shared variable does not depend on the current value.
- Latch
  - A data structure that performs a single type of state changes
    - e.g. False → True
  - Often used to terminate threads.
    - c.f. "done" variable in prior examples

An Example Latch

```
volatile boolean done = false;
```

```
public void setDone(){
  done = true;}
```

```
public void run(){
  while(true)
    if(done) break;
    counter++;
}
```

- The state of “done” always changes in a unidirectional way: false → true
  - “true → false” never occur.


```
volatile boolean done = false;
```

```
public void setDone(){
  done = true;}
```

```
public void run(){
  while(true)
    if(done) break;
    counter++; }
```

- No need to surround the if statement with lock() and unlock().
- Thread sync is not performed.
  - A context switch can occur in between
    - Evaluating the “done” variable and
    - Applying the current value in “done” to the if statement
- Memory sync is performed.
  - The most up-to-date value of “done” is applied to the if statement.


```
volatile boolean done = false;
```

```
public void setDone(){
  done = true;}  // NO NEED TO SURROUND THIS WITH LOCK() and UNLOCK()
```

```
public void run(){
  while(true)
    if(done) break;
    counter++; }
```

- Thread sync is not performed.
  - A context switch can occur in between evaluating the value of “true” and assigning it “done.”
  - All threads will assign “true” to “done.”
    - No other possible state changes.
  - Writer threads do not generate race conditions.
- Memory sync is performed.
  - The value of “true” must be copied to the main memory once the assignment (“done=true”) is completed.

Syntactic Difference

- Without “volatile”

```
boolean done = false;
```

```
ReentrantLock lock =
    new ReentrantLock();
```

```
public void setDone(){
 lock.lock();
 done = true;
 lock.unlock();
}
```

```
public void run(){
  while( true ){
      lock.lock();
      if( done ) break;
      // Do some work
      lock.unlock(); }
}
```

- With “volatile”

```
volatile boolean done = false;
```

```
public void setDone(){
  done = true;
}
```

```
public void run(){
  while( true )
    if( done ) break;
    // Do some work
}
```

Limited Effectiveness/Usefulness

- A "volatile" variable is guaranteed to have the most up-to-date value whenever it is read.

```
– volatile int a;
                         // These 2-step operations are all thread-safe,
  int b = a;             // even if a context switch occurs in between
  if(a==0){...}          // the 2 steps and another thread changes the
  println(a);            // value of "a" there.
```

- However, it is not a silver bullet to eliminate all possible race conditions.

```
– a + 1;                 // 3 steps. Thread-safe.
  b = a + 1;             // These 4-step operations are NOT thread-safe.
  if(a+1>0){...};        // The first 3 steps are thread-safe though. A
                         // race condition can occur in between the 3rd
                         // and 4th steps.

  println(a+1)           // 4 steps. Not thread-safe.
```

- "volatile" is effective only in the read operations that have no intermediate state.

- A "volatile" variable NEVER eliminate race conditions in write operations on it.

```
– volatile int a;
  a = 1;               // 2 steps. A race condition can occur if a
                       // context switch occurs in between the 2 steps
                       // and another thread changes the value of "a"

  a = a + 1;           // 4 steps. NOT thread-safe
  a++;
```

- Use a "volatile" variable only when you can live with these race conditions.

- Do NOT use "volatile" for arrays.

A Concurrency Bug in Jetty

- Jetty
  - An open source web implementation in Java
    - http://jetty.codehaus.org/jetty/
- A bug report (March 2010)

```
– // Jetty 7.1.0,
  // org.eclipse.jetty.io.nio,
  // SelectorManager.java, line 105
  private volatile int _set;
  public void register(SocketChannel channel, Object att){
      int s = _set++;      // This part is NOT thread-safe
      ...
  }
  public void addChange(Object point){
      synchronized (_changes){
          ...
      }
  }
```

In Summary…

- NOT a general-purpose, widely-applicable threading tool

- Powerful only in some specific cases
  - In practice, assume it is useful only for simplifying the implementation of a latch.
    - Useful to implement flag-based thread termination and 2-step thread termination.