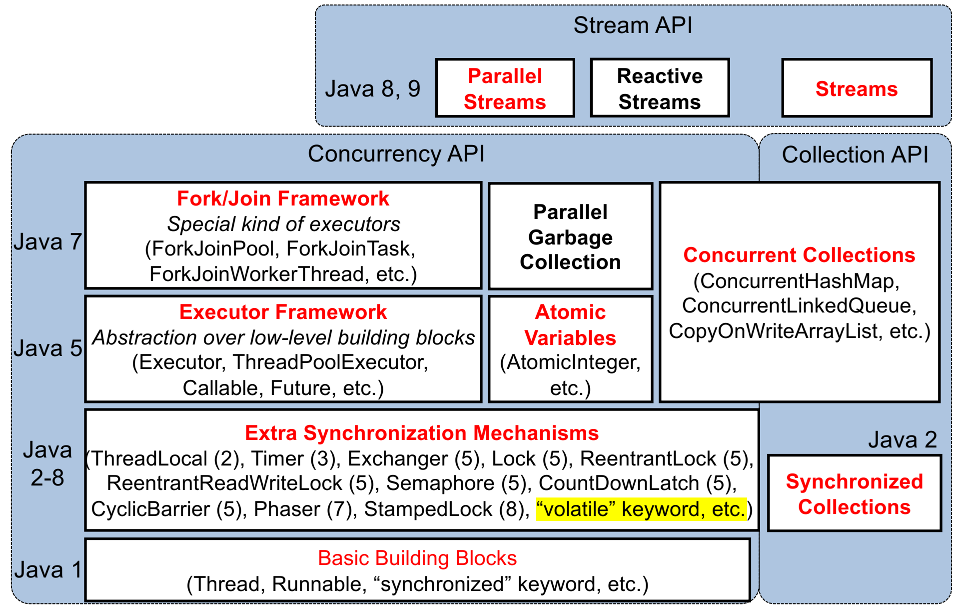
Volatile Variables

Concurrency API in Java



Recap: Thread Synchronization (Locking)

- You must do thread synchronization (locking) to guard a variable shared among threads.
  - Otherwise, race conditions can occur.

What is the "volatile" Keyword?

- You can skip thread synchronization to guard a shared variable in some cases by defining the variable as *volatile*.
  - No race conditions occur on the variable even though you never do thread sync.
    - Greater peace of mind regarding thread safety
    - Less coding/maintenance effort
    - Less overhead (higher performance)
      - Thread sync consumes some CPU time and memory space.

Recap: A Race Condition in Explicit Thread Termination

```
boolean done = false;

public void setDone(){
  done = true;
}

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

- A thread ("A": soon-to-be-terminated)
  - Has read the current value of **done**, which is false (in blue), but…
  - Has NOT applied the value (false) to a conditional (in yellow) yet
- A context switch occurs.
- Another thread ("B": terminator) calls **setDone()** to assign true to **done**.
- Another context switch occurs.
- The value change that "B" just made is NOT visible for "A".
  - "A" applies the value of **done** (false) to the conditional (in yellow) and runs green code instead of breaking out from the loop.

Thread-safe Code

- With thread sync

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

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

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

- With a volatile variable

```
• 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.
- A volatile variable makes code simpler and less error-prone.

Read on a Volatile Variable

```
volatile boolean done = false;

public void setDone(){
  done = true;
}

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

- A context switch can still occur in b/w
  - Reading the value of **done** (in blue) and
  - Applying it to the conditional (in yellow).
- Then, Thread "B" (terminator thread) may call **setDone()** to assign true to **done**.
- After another context switch, Thread "A" re-loads the most up-to-date value of **done** from the memory
  - because it's a volatile variable.
- A volatile variable makes a value change on a shared variable visible for all threads.

Write on a Volatile Variable

```
volatile boolean done = false;

public void setDone(){
  done = true;
}

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

- A "terminator" thread #1 called **setDone()**
  - Has loaded the value of true, but…
  - Has NOT assigned it to **done** yet.
- Then, a context switch occurs.
- Another "terminator" thread #2 calls **setDone()**
  - Completes an assignment of true to **done**
- After another context switch, thread #1 assigns true to **done**.
- No race conditions occur.
  - Because two threads (#1 & #2) assign the same value.

Special Effect on a Volatile Variable

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

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

  if(a==0)                    // 3 steps. It is thread-safe, even if a
                              // context switch occurs in b/w the 1st and 2nd
                              // steps, or in b/w the 2nd and 3rd steps, and
                              // another thread changes the value of "a"
                              // there.

  a + 1;                      // 3 steps: reading the value of "a", loading 1
                              // and summing up "a" and 1.
                              // This 3-step op is thread-safe, even if a
                              // context switch occurs in b/w the 1st and 2nd
                              // steps, or in b/w the 2nd and 3rd steps, and
                              // another thread changes the value of "a"
                              // there.
```

- This "special effect" is called "memory semantics" or "memory effect" of a volatile variable in Java API doc and other resources.


```
– long a;
  a = 10L;              // Requires 2 atomic steps because long is
                        // a 64-bit type. NOT thread-safe.
                        // A context switch can occur during the 10L
                        // value assignment. The first 32-bit half of
                        // value may be assigned by a thread, and the
                        // second half may be assigned by another
                        // thread.

– volatile long a;
                        // Thread-safe even if a context switch occurs
  a = 10L;              // during the 10L value assignment.
```

- This "special effect" is called "memory semantics" or "memory effect" of a volatile variable in Java API doc and other resources.

Limited Effectiveness

- A volatile variable does **NOT** eliminate all possible race conditions…

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

  b = a + 1;                  // These operations are NOT thread-safe.
  if(a+1>0)                   // The first 3 steps (in yellow) are
  println(a+1);               // thread-safe though.
                              // The 3rd step generates an intermediate
                              // state that is not volatile. It may NOT be
                              // in synch with the most up-to-date value of
                              // "a", if a context switch occurs in b/w the
                              // 3rd and 4th steps and another thread changes
                              // the value of "a" there.
```

- A volatile variable is effective only when no intermediate states/values are generated with it.

- A volatile variable is effective only when no intermediate states/values are generated with it.

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

  a = a + 1;            // 5 steps. Not thread-safe.
  a++;                  // The first 3 steps (in yellow) are
                        // thread-safe though.
                        // The 3rd step generates an intermediate
                        // state that is not volatile. It may NOT be
                        // in synch with the most up-to-date value of
                        // "a", if a context switch occurs in b/w the
                        // 3rd and 4th steps and another thread changes
                        // the value of "a" there.
```

- The “volatile” keyword imposes a strong/strict limitation in its use cases.
  - It works only for a variable whose value is used without any intermediate state.
    - Good: if(a>0), if(done), a=1
    - NOT good: if(a+1>0), if(!done), b=a+1, a=a+1
- Carefully use it only when you can live with this limitation.

- The keyword “volatile” works for all primitive types
  - Both 32-bit and 64-bit types
- However, it DOES NOT work for arrays and reference types.

When to Use Volatile Variables?

- So, what what use cases generate no intermediate states/values?
  - Probably, ***latch*** only.
    - Performs a single type of value changes
      - e.g. False → True, Non-zero → Zero, Negative → Non-negative, null to non-null, etc.
    - It is used in explicit thread termination.
      - The flag variable **`done`** in prior examples
        - » The state of `done` always changes in a unidirectional way: false → true.
        - » “true → false” never happen.
      - The flag **`done`** is a perfect example to be volatile effectively.

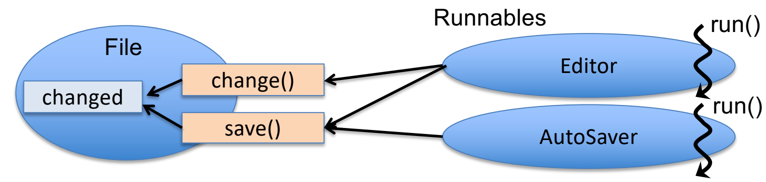
In Summary

- The “volatile” keyword is NOT a general-purpose, widely-applicable tool.
- Powerful only in some specific cases
  - In practice, assume it is useful only for implementing a latch in a thread-safe manner.
    - Very useful to implement flag-based thread termination and 2-step thread termination in a thread-safe manner.
- Note that there exist numerous projects that misuse “volatile” variables.

Exercise (Not HW)

- Declare a flag as a volatile variable in flag-based and 2-step thread termination schemes that you have implemented.
  - e.g., prime generation and factorization
  - As far as the flag is "volatile," you don't have to guard it with a lock.

Note: Do NOT use "volatile" for a Variable that allows Bi-directional Value Changes



- File
  - Has a boolean variable: `changed`
    - Initialized to be `false`.
  - `change()`
    - Changes the file's content.
    - Assigns `true` to `changed`.
  - `save()`
    - If `changed==false`, return;
    - If `changed==true`
      - Save the file's content.
      - Assign `false` to `changed`.

Note: Do NOT Use "volatile" to Implement *Concurrent Singleton*

- Not thread-safe

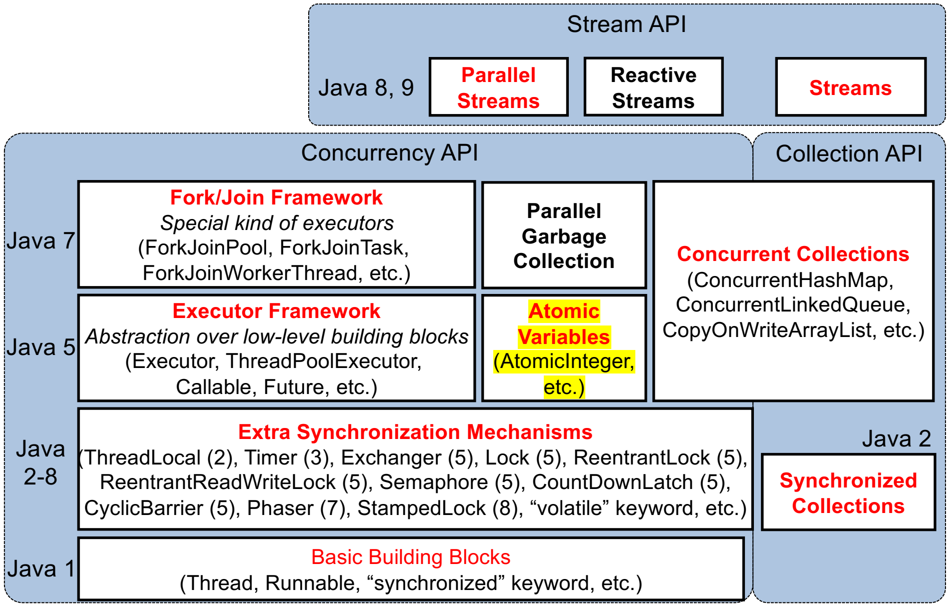
```
public class Singleton{
  private Singleton(){};
  private static Singleton instance = null;

  public static Singleton getInstance(){
    if(instance==null)
      instance = new Singleton();
    return instance;
  }
}
```

- Can we turn the *Singleton* class to be thread-safe by changing the `instance` variable to be volatile?
  - Its value changes unidirectionally: null to non-null.
- No, unfortunately,
  - Because it's a reference-type variable.

Atomic Data Structures

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 implemented to be thread-safe. They avoid race conditions.

Atomic Variables

- Serve as "better" volatile variables.
- Offer the "special" memory effects as volatile variables do.
- Provide additional support for atomic value changes.
  - get() has the memory effect of reading a volatile variable.
    - Returned value is guaranteed to be the most up-to-date whenever it is used.
  - set() has the memory effect of writing to a volatile variable.
    - Value assignment is atomic.
  - *read-and-update* methods (e.g., `xxxAndSet()` and `getAndXxx()`) have the memory effects of reading and writing to volatile variables.
- Highly recommended as far as they match your use cases.

AtomicBoolean

- Offers the methods to manipulate a single boolean value atomically (i.e., in a thread-safe manner).

  - ```
    boolean flag = false;
    if(flag)                    // 2 steps. Not thread-safe
    println(flag);              // 2 steps. Not thread-safe
    ```
  - ```
    volatile boolean vFlag = false;
    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 the memory effect of reading a volatile variable.
    - A context switch can occur right after `get()` returns, and another thread may change the boolean value.
    - However, `AtomicBoolean` guarantees that the most up-to-date value will be re-loaded, whenever it is used.

  - ```
    volatile boolean vFlag = false;
    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 a race condition can occur
                                // if a context switch happens in b/w
                                // the two ops (in yellow and blue) and
                                // vFlag is updated.
    ```
  - ```
    AtomicBoolean atomicFlag = new AtomicBoolean(false);
    atomicFlag.set(true)                      // Thread-safe
    atomicFlag.compareAndSet(true, false);    // Thread-safe!
    ```
  - `set()`: Thread-safe. Atomically sets a value.
  - *read-and-update* methods (e.g., `xxxAndSet()` and `getAndXxx()`) have the memory effects of both reading and writing volatile variables.

- ```
  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 variable 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(){
   while(true){
    if(done.get()) break;
    ... // do some work
    } } }
```

- Use **AtomicBoolean** or "volatile" to implement explicit thread termination in all future HWs.

- Exercise: Use **AtomicBoolean** in explicit thread termination in prior sample programs

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 int 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 the memory effect 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 most up-to-date value will be re-loaded, whenever it is used.

```
– volatile int 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 sets a value., with the memory effect of writing to 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 int i = 0;
  i = 1;                         // Thread-safe.

  if(i==0){i = 1;}               // NOT thread-safe.
                                 // Read and write ops are thread-safe
                                 // each, but a race condition can occur
                                 // if a context switch happens in b/w
                                 // the two ops (in yellow and blue) and
                                 // the variable is updated.

– 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 LE and returns the updated value.

- `IntUnaryOperator`: a general-purpose functional interface

- `updateFunction`: a LE that takes the current `int` value as a parameter, updates it and returns the updated value.
  - This update logic runs atomically

```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)-> ++i) );   // 11. Thread safe
atomicInt.incrementAndGet();                // 12. Thread safe
```

	Params	Returns
UnaryOperator<T>	T	T
IntUnaryOperator	int	int

```
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 = 1;}          // NOT thread-safe.
                           // Read and write ops are thread-safe
                           // each, but a race condition can occur
                           // if a context switch happens in b/w
                           // the two ops (in yellow and blue) and
                           // the variable is updated.
```

```
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 running a given LE and returns the updated value.

- `IntBinaryOperator`: a general-purpose functional interface

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

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

- This LE runs atomically (i.e., in a thread-safe manner)

	Params	Returns
BinaryOperator<T>	T, T	T
IntBinaryOperator	int, int	int

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

                              // Returns (0+10)/2
```

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

AtomicReference<V>

- Offers thread-safe methods to manipulate a reference type value atomically.
  - Note: "volatile" does NOT work for reference types.

```
– Car car = new Car(...);
  Car anotherCar = car;          // 2 steps. Not thread-safe.
  if(car!=null)                  // 3 steps. Not thread-safe.
```

```
– AtomicReference<Car> atomicCar =
        new AtomicReference<>(new Car(...)); // Thread-safe
  atomicCar.set(new Car(...));               // Thread-safe
  if(atomicCar.get()!= null)                 // Thread-safe.
```

  - Other useful methods
    - `compareAndSet(V,V)`, `updateAndGet(UnaryOperator<V>)`, `accumulateAndGet(V, BinaryOperator<V>)`, etc.
    - C.f. API documentation

Recap: *Concurrent Singleton*

- Guarantee that a class has only one instance.
  - Its static factory method is implemented to be thread-safe because it performs thread sync.

```
• public class ConcurrentSingleton{
   private ConcurrentSingleton(){};
   private static ConcurrentSingleton instance = null;
   private static ReentrantLock lock = new ReentrantLock();

   // Factory method to create or return the singleton instance
   public static Singleton getInstance(){
       lock.lock();
       try{
         if(instance==null){ instance = new ConcurrentSingleton(); }
         return instance;
       }finally{
         lock.unlock();
       }}}
```


Exercise (Not HW)

- Revise `ConcurrentSingleton`
  - Use `AtomicReference` to define `instance`
  - Take out `ReentrantLock`. No need to do thread sync!
  - Implement `getInstance()` in a thread-safe manner with `AtomicReference`. Use `compareAndSet()` or `updateAndGet()`

```
• public class ConcurrentSingleton{
    private ConcurrentSingleton(){};
    private static AtomicReference<ConcurrentSingleton> instance
      = new AtomicReference<>();

    public static AtomicReference<ConcurrentSingleton> getInstance(){
      ...
    } }
```

- Test it:

```
public static void main(String[] args){
  for(int i=0; i<10; i++){
    new Thread(
      ()->{Sys.out.println(ConcurrentSingleton.getInstance());}).start();
  }}
```