AtomicReferenceArray

Guide to working with AtomicReferenceArray.

Overview

The official documentation of the `AtomicReferenceArray` states that it is an array of object references, that each one, may be updated atomically. However, it is easy to get confused with all the other array constructs and `AtomicReferenceArray`, so we'll go through one by one to clarify the differences. Also, according to the Java specification, assignments in Java except for primitive types `long` and `double` are atomic, which may confuse the reader as to what does *atomically updating* an element of an `AtomicReferenceArray` instance means?

Remember to draw a distinction between assignment and update of an atomic variable. Assignment involves changing the reference (memory location) pointed to by an atomic variable e.g.

```
// first assignment
AtomicReference<Object> atomicReference = new AtomicReference<>(null);

// assigning a different AtomicReference object
atomicReference = new AtomicReference<>(Long.class);
```

Updating atomically, implies that the value of an atomic variable holds is changed based on an expected value the variable currently holds e.g.

```
// first assignment
AtomicReference<Object> atomicReference = new AtomicReference<>(null);

// atomic update
atomicReference.compareAndSet(null, Long.class);
```

The atomic update has consequences for memory visibility and establishes the happens-before relationship as we'll learn shortly. In contrast, an assignment operation may not establish a *happens-before* relationship if the variable isn't marked `volatile`.

Array Reference

First up, we'll discuss what is an array reference. There is a distinction between an array reference and the elements of the array. Array is a sequence of values and the values are the elements of the array, all of the same type. The array is created using the `new` operator as shown below:

```
Long[] myLongs = new Long[10];
```

The `new` operator, allocates memory on the heap for the array and returns the address of the memory allocation into the variable `myLongs`. The variable `myLongs` isn't the array; rather the variable holds the address of the memory location where the elements of the array live. The memory location's address is stored in the variable `myLongs` and the variable `myLongs` is said to be a reference to the array.

This distinction between array reference and the array is important because one could update the array reference in a thread-safe manner but the elements of the array may still be updated in a thread-unsafe manner. Consider the following methods:

```
Long[] myLongs = new Long[10];

// method is thread-safe
void increaseArraySize() {
   synchronized (this) {
      Long[] newArray = new Long[myLongs.length * 2];

      // copy old array elements into new array
      for (int i = 0; i < myLongs.length; i++)
         newArray[i] = myLongs[i];

      // update the reference
      myLongs = newArray;
   }
}

// method isn't thread-safe
void set(int index, Long newVal) {
   myLongs[index] = newVal;
}
```

Note that the array reference is being manipulated in a thread-safe manner but the elements of the array are not.

Difference between array and volatile array

Another source of confusion is the following snippet:

```
Long[] myLongs = new Long[10];
volatile Long[] volatileMyLongs = new Long[10];
```

The reference `myLongs` is prone to getting cached by a processor whereas the reference `volatileMyLongs` is always read and written from main memory. However, the elements for both the arrays can get cached by a processor and multiple threads working on the array without appropriate synchronization constructs can see stale values for array elements in addition to thread-safety issues. Going back to our example, let's say we modify the `increaseArraySize()` method as follows:

```
void increaseArraySize() {
   Long[] newArray = new Long[myLongs.length * 2];

   // copy old array elements into new array
   for (int i = 0; i < myLongs.length; i++)
       newArray[i] = myLongs[i];

   // update the reference
   myLongs = newArray;
}
```

If only the main thread ever invokes the `increaseArraySize()` method, then to avoid the locking overhead, we may remove the `synchronized` block from the method. However, without marking `newLongs` as `volatile`, it could happen that the main thread creates a new array and updates the address in the local cache and never in the main memory. Consequently, other threads never see the new array and keep working on the old one. It could also happen that other threads have the reference cached and never retrieve the latest reference from the main memory since `myLongs` isn't marked volatile. Either way, marking the reference i.e. the variable `myLongs` volatile solves the memory visibility issue. However, marking `myLongs` as `volatile` has no effect on the array elements. Consider a thread that executes the following snippet:

```
Long cachedObject = myLongs[0];
```

The `Long` object at zero-th index isn't `volatile` and will suffer from memory visibility issues caused by caching, even though the reference to the array has been marked `volatile`.

Also, bear in mind that `volatileMyLongs` reference is volatile, and `volatileMyLongs` is **NOT** an array of volatile `Long` objects.

Difference between volatile array and `AtomicReferenceArray`

The next question we'll address is the difference among the three statements in the following snippet:

```
volatile Long[] volatileMyLongs = new Long[10];
AtomicReferenceArray<Long> atomicReferenceArray = new AtomicReferenceArray<>(10);
volatile AtomicReferenceArray volatileAtomicReferenceArray = new AtomicReferenceArray(10);
```

The reference `volatileMyLongs` is volatile but the reference `atomicReferenceArray`, which points to an object of type `AtomicReferenceArray` on the heap isn't. The elements of both the arrays can be updated atomically, since they are references and the Java specification mandates atomic assignments for references. However, the array elements of `volatileMyLongs` can suffer from memory visibility issues such as being cached in the processor's local memory and any updates to them will not establish the *happens-before* relationship with other variables observable to a thread. In the case of the `atomicReferenceArray` variable, the individual elements at each index are always updated in the main memory and a stale value is never observed by any thread. Furthermore, manipulating a single element of `atomicReferenceArray` also establishes the *happens-before* relationship with other variables in the scope.

The variable `volatileAtomicReferenceArray` reference, itself and its array elements don't suffer from memory visibility issues. If the reference is updated after initialization using the following snippet, the change will be visible to all threads:

```
 volatileAtomicReferenceArray = newReference;
```

A similar update as above to `atomicReferenceArray` variable can potentially be cached and not observable by all threads without using synchronization constructs.

Difference between `AtomicReferenceArray` and array of `AtomicReference`-s

The astute reader would question the difference between instanting an array of `AtomicReference`-s vs instantiating an object of `AtomicReferenceArray`. Consider the following snippet:

```
      AtomicReferenceArray<Long> atomicReferenceArray = new AtomicReferenceArray<Long>(100);
      AtomicReference[] arrayOfAtomicReferences = new AtomicReference[100];
```

Functionally, the above two are equivalent, however, the instance of `AtomicReferenceArray` occupies less memory than an array of `AtomicReference`-s of equivalent size. The memory saving can become significant when the size of the array is in the thousands or millions.

Summary of differences

Snippet	Is reference assignment atomic?	Is update atomic?	Reference update establishes *happens-before* relationship?	Element update establishes *happens-before* relationship
`Long[] myLongs = new Long[10];`	Yes	No	No	No
`volatile Long[] volatileMyLongs = new Long[10];`	Yes	No	Yes	No
`AtomicReferenceArray<Long> atomicReferenceArray`	Yes	Yes	No	Yes
`volatile AtomicReferenceArray volatileAtomicReferenceArray`	Yes	Yes	Yes	Yes

Example

In the example below, we create an instance of `AtomicReferenceArray<Long>` typed on the `Long` class. Next, we run fifteen threads and each one of them attempts to initialize each element of the array with a `Long` object. The reference of the `Long` object is stored in the array element. No matter how many times we run the program, each element of the array should only be initialized by one thread.

Java

Press ⌃ + ⏎ to interact

main.java

Saved

```java
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicReferenceArray;

class Demonstration {

    public static void main( String args[] ) throws Exception {

        int arrayLength = 10;
        AtomicReferenceArray<Long> atomicReferenceArray = new AtomicReferenceArray<>(arrayLe
        ExecutorService executor = Executors.newFixedThreadPool(15);

        try {
```

Run

Back lesson
AtomicReference

Mark As Completed

Next
AtomicReferenceFieldUpdater