Atomics

Without the requirement for explicit synchronization, atomic variables offer a quick and effective way to carry out thread-safe operations.

Problem with non-atomic operations

Conventional non-atomic operations like increment and decrement operations are not **thread-safe**, meaning they can result in incorrect values being stored if several threads try to access the same variable simultaneously.

Several threads may simultaneously access and alter the same variables in multi-threaded systems, resulting in race conditions and other synchronization problems.

Consider a counter program example in which several threads are attempting to increase a single shared counter variable.
A bad counter class that is not thread-safe might look something like this:

```
public class BadCounter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

The problem with this implementation is that the increment() method is not atomic.

In order to update a variable's value, we have first to read the current value from memory, update the read value, and then write the updated value back to memory. If this is not done atomically, two threads may concurrently execute increment() and end up overwriting one another's modifications.

Depending on the timing of the threads, the count can wind up being 1 instead of 2 if the initial count is 0 and two threads use increment() simultaneously.

Atomic operations to the rescue

Using atomic variables, you can carry out thread-safe operations without explicit synchronization.

All operations on atomic variables are guaranteed to be thread-safe.

An **atomic operation** is an operation that is performed as a single, indivisible unit of work.

In other words, an atomic operation is performed entirely or not at all.

A thread gets a **lock** on an atomic variable when it performs an operation to prevent concurrent modification by other threads.

This ensures that the atomic variable's value is always accurate and consistent.

Let's consider the previous example of the basic counter. To make this counter thread-safe, we can use an atomic variable instead of a regular integer.

Here's an example of a thread-safe counter using AtomicInteger:

```
import java.util.concurrent.atomic.AtomicInteger;

public class GoodCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }
}
```

In this case, the incrementAndGet() method of the AtomicInteger class is an atomic operation. When a thread calls this method, it acquires a lock on the count variable to ensure that no other threads can modify it at the same time.

This guarantees that the count variable is always incremented correctly, even when multiple threads access it simultaneously.

Atomics API

The Java java.util.concurrent.atomic package provides a set of classes that implement atomic operations on variables of various types. These classes are:

- AtomicBoolean: provides atomic operations on a boolean value.
- AtomicInteger: provides atomic operations on an int value.
- AtomicLong: provides atomic operations on a long value.
- AtomicReference: provides atomic operations on an object reference.

Each of these classes provides a set of methods for performing atomic operations on their corresponding types. Let's focus on the AtomicInteger class and explore its methods:

- **get()** returns the current value of the AtomicInteger.

```
// create an AtomicInteger with an initial value of 0
AtomicInteger atomicInt = new AtomicInteger(0);

// get the current value of the AtomicInteger
int currentValue = atomicInt.get();

System.out.println(currentValue); // = 0
```

- **getAndAdd(int delta)** adds delta to the current value of the AtomicInteger and returns the old value.

```
int oldValue = atomicInt.getAndAdd(10);

System.out.println(atomicInt); // = 10
System.out.println(oldValue); // = 0
```

- **getAndDecrement()** decrements the current value of the AtomicInteger by 1 and returns the old value.

```
oldValue = atomicInt.getAndDecrement();

System.out.println(atomicInt); // = 9
System.out.println(oldValue); // = 10
```

- **getAndIncrement()** increments the current value of the AtomicInteger by 1 and returns the old value.

```
oldValue = atomicInt.getAndIncrement();

System.out.println(atomicInt); // = 10
System.out.println(oldValue); // = 9
```

- **getAndSet(int newValue)** sets the AtomicInteger to newValue and returns the old value.

```
oldValue = atomicInt.getAndSet(42);

System.out.println(atomicInt); // = 42
System.out.println(oldValue); // = 10
```

- **addAndGet(int delta**) adds delta to the current value of the AtomicInteger and returns the new value.

```
int newValue = atomicInt.addAndGet(5);

System.out.println(atomicInt); // = 47
System.out.println(newValue); // = 47
```

- **decrementAndGet()** decrements the current value of the AtomicInteger by 1 and returns the new value.

```
newValue = atomicInt.decrementAndGet();

System.out.println(newValue); // = 46
```

- **incrementAndGet()** increments the current value of the AtomicInteger by 1 and returns the new value

```
newValue = atomicInt.incrementAndGet();

System.out.println(newValue); // = 47
```

- **compareAndSet(int expect, int update)** atomically sets the value to update if the current value is equal to expect.

```
boolean success = atomicInt.compareAndSet(10, 20);

System.out.println(atomicInt); // = 47
System.out.println(success); // = false
```

- **weakCompareAndSet(int expect, int update**) atomically sets the value to update if the current value is equal to expect.

The main difference between compareAndSet() and weakCompareAndSet() is in their memory semantics.

The compareAndSet() method provides a strong memory consistency guarantee, meaning that any changes made by one thread before calling it will be visible to any later calls to it by other threads.

On the other hand, weakCompareAndSet() guarantees less memory consistency.

It only ensures that modifications made by one thread before the call to weakCompareAndSet() are visible to subsequent calls to weakCompareAndSet() by the same thread.

This approach makes no promises about the visibility of modifications made by other threads.

- set(int newValue) sets the value to newValue.

```
atomicInt.set(50);

System.out.println(atomicInt); // = 50
```

- lazySet(int newValue): the value is eventually updated to newValue, but this does not guarantee that any thread executing get() will see the new value. The order of operations may also be changed. This absence of ordering guarantees might result in certain performance advantages. The operation can be done more **rapidly** and with less overhead than other atomic operations that offer ordering guarantees since there is no need to synchronize with other threads.

Optimistic vs. pessimistic locking

The compareAndSet(expectedValue, newValue) method mentioned above can be used in a pattern known as **optimistic locking**.
In this pattern, several threads can access and alter a shared resource concurrently, without requiring a full lock or blocking other threads.

The basic idea of optimistic locking is to read the resource's current value, perform some computations or modifications on that value, and then attempt to set the modified value back into the resource atomically
using compareAndSet().

If the compareAndSet() operation succeeds, the alteration is finished, and the thread can proceed. If it fails, another thread has already modified the resource, and the thread must address the failure somehow.

Here's an example of how you can implement this pattern:
AtomicInteger sharedResource = new AtomicInteger(0);

```
while (true) {
    int currentValue = sharedResource.get();
    int modifiedValue = computeNewValue(currentValue); //do some work on the
obtained value

    if (sharedResource.compareAndSet(currentValue, modifiedValue)) {
        // The modification was successful, we can continue
        break;
    } else {
        // Another thread modified the shared resource in the meantime,
        // we must start over from the beginning
        continue;
    }
}
```

In this example, the loop will continue until the compareAndSet operation succeeds, meaning that the modification was successful and the thread can continue.

The compareAndSet() function makes use of the low-level **Compare-And-Swap** (CAS) operation of the CPU.

This operation allows the writing of a value to a memory address, but only if the current value at that location matches an expected value.

The CAS operation is frequently used in concurrent programming to perform atomic operations, such as compareAndSet().

In contrast, **pessimistic locking** is when a thread obtains a **full lock** on a shared resource before accessing or altering it.

Pessimistic locking is more costly and can cause contention and blocking of other threads, but it gives better consistency guarantees and may be more appropriate in some instances.

In terms of performance, optimistic locking is generally faster and more scalable than pessimistic locking since it avoids the overhead of acquiring and releasing locks.

Nevertheless, optimistic locking might cause conflicts and retries, slowing down the system's overall throughput if the conflict rate is significant.

Finally, the choice between optimistic and pessimistic locking should depend on the application's unique needs and the desired balance of performance and consistency.