We might write a class in such a way that certain properties are held as invariants.

Methods can be written to adhere to preconditions and post conditions and have exceptions defined.

However, within a method, the class-wide invariant might not hold true.

So, what happens if a method is de-scheduled in the middle or if there are multiple threads running on multiple processors?
**UniqueList – Invariants, Pre/Post-Conditions**

Invariant for *UniqueList*: the list never contains duplicates.

Precondition for `add()` method: the value passed in should not exist in the list already.
Postcondition for `add()` method: the new value is added to the end of the list.
Exception handling for `add()` method: if the value passed in already exists in the list, throw an `IllegalArgumentException` and don’t add this duplicate item to the list.

Precondition for `swap()` method: the values passed in are valid positions in the list.
Postcondition for `swap()` method: the values in the two positions are exchanged.
Exception handling for `swap()` method: if the positions aren’t valid, throw an `IllegalArgumentException` and don’t try to swap.

---

**UniqueList – Is the class correct?**

Let’s go to the class code posted on the class website…
**UniqueList – Is the class thread-safe?**

Let’s run the tests posted on the class website…

---

**Synchronization / Concurrency Control**

How could we fix the **UniqueList** class to be thread-safe?

We could mark all methods as “synchronized” in the class (sometimes called the “monitor” pattern). This would make every method logically atomic. As long as there are no public data members, there’s no risk of things being visible while in an invalid state.

*However*, this would essentially remove any chance of performance improvements from concurrency that relies on many threads calling complex class methods.

In the case of **UniqueList** there really aren’t places where there is work to do inside that methods that doesn’t depend on the list itself, so it could be a valid approach.

However, other examples will benefit from finer lock granularity.
Finer-Grained Locking

In a larger example, we might look for data members that are mentioned in the class invariants and use their intrinsic locks when they are accessed.

Within a method we can look for critical sections where we might want to obtain a hold a lock over a sequence of related instructions.

Imagine a class that allows you to manipulate and use an RGB colored cubic 6-point spline. Any interactions with the color channels would need to be synchronized. Any interactions with the points used to create the spline would need to be synchronized. However, you could in theory have one thread dealing with colors while another was dealing with the spline points without them having to be sync’ed with each other.

Dealing with overlapping invariants

Let’s say we had a cubic spline defined by 6 points.

Let’s say we had five invariants about:
(P1,P2) (P2,P3) (P3,P4) (P4,P5) (P5,P6).

What could happen if we create five dummy objects to use to lock the pairs of points in each invariant?

- This is generally discouraged since in several places you would have one value (eg: P2) being guarded by multiple locks.