In the face of concurrent, modifying access, the state of the object may be modified by concurrent callers. Forbid Concurrent Modification: Permit Concurrent Modification: 1 - Access Permissions Programmers specify if/how references may be aliased, and whether modification can occur. There are 5 permission kinds:

- **Immutable** - Reference is not subject to concurrent modification! Share means other modifying refs. exist. Reference is subject to concurrent modification.
- **Full** - Reference is subject to concurrent modification and acquisition of the state of the object.
- **Unique** - Share ref. (requires "Full"). Assume in "Smart"
- **Pure** - Copy ref. (requires "Full"). Assume in "Smart"

Other more familiar examples:
- java.util.List (remove is smart method)
- java.util.Map.Entry (containsKey is smart method)
- java.util.Scanner (readLine is smart method)
- java.util.concurrent.atomic.AtomicReference

Can we statically, modularly ensure correct protocol usage?

- **Immutable** - Reference is not subject to concurrent modification!
- **Full** - Reference is subject to concurrent modification and acquisition of the state of the object.
- **Unique** - Share ref. (requires "Full"). Assume in "Smart"
- **Pure** - Copy ref. (requires "Full"). Assume in "Smart"

**Solution in a Nutshell**

1. Programmers specify Access Permissions which fall into 5 categories: Permanent, Modifiable, Immutable, Unique, and Empty.
2. Method implementations need to check if a permission is satisfied before proceeding.
3. Invariants are used to ensure that modifications are made to the correct state.

**Solution: Access Permissions Modularly Denote Concurrent Modification**

- **Author**: Beckman, Nels E.
- **Title**: Modular Typestate Checking in Concurrent Java Programs
- **Date**: SEFM 2005.

**Example: Finding Race Condition**

```java
def pass() {
    new OnePlaceBuffer<String>() {
        @Share
        @FalseIndicates("Full")
        @Pure
        boolean isFull() {
            if (b.isFull()) {
                synchronized = { b }
                return true;
            }
            else if (b.isEmpty()) {
                synchronized = {}   synchronized = {}
                return false;
            }
        }
        void put(String item) {
            synchronized = { b }
            b.put(item);
        }
        T get() {
            synchronized = { b }
            return b.get();
        }
    }
    pass(OnePlaceBuffer<String>())
    .start();
}
```

**Example: Verifying Corrected Code**

```java
def pass() {
    new OnePlaceBuffer<String>() {
        @Share
        @FalseIndicates("Full")
        @Pure
        boolean isFull() {
            if (b.isFull()) {
                synchronized = { b }
                return true;
            }
            else if (b.isEmpty()) {
                synchronized = {}   synchronized = {}
                return false;
            }
        }
        void put(String item) {
            synchronized = { b }
            b.put(item);
        }
        T get() {
            synchronized = { b }
            return b.get();
        }
    }
    pass(OnePlaceBuffer<String>())
    .start();
}
```