CSE 331
Software Design & Implementation

Autumn 2025

Section 5 – Mutable ADTs

Administrivia

- HW5 released tonight - **Due @ 6pm Friday**

Externally Documenting ADTs - Review

- JavaDoc comments in interface use “tags” to describe what ADT methods do in terms of the **abstract state**

```
/**
 * High level description of what function does
 * @param a What "a" represents + any conditions
 * @requires Rules about multiple params and Abstract State (obj)
 * @returns Detailed description of return value
 * @throws Condition when errors will be thrown
 */
```

- For mutable ADTs, will have 2 additional tags to describe “mutator” methods

```
 * @modifies states what could be mutated by function (obj)
 * @effects Detailed description of guaranteed changes
```

Specification Strength Mutation - Review

- Adding more to `@returns` **increases** specification strength

- Adding more to `@effects` **increases** specification strength

- Adding more to `@modifies` **decreases** specification strength
    - Note that this is not a promise or guarantee

Review: ADT Correctness

- To prove a method of a *mutable* ADT correct, we need to:
  - show the spec's postcondition holds
  - show the RI holds
- In other words, these make up the postcondition to prove

- For *immutable* ADTs
  - only need to prove RI is established by the constructor

    because fields don't change, we know RI will remain true

- AF doesn't need to be proven (it always holds when RI holds)

    it is just a given relationship between fields and obj to use in other proofs

    think of it like a fact :)

Testing - Review

- **Statement Coverage**
  - Test every executable statement reachable by an allowed input
- **Branch Coverage**
  - For every conditional, test all branches for allowed inputs
- **Loop Coverage**
  - Every loop/recursive call must be tested on 0, 1, any 2+ iterations for allowed inputs
- **Exhaustive Testing**
  - Test all possible inputs for functions with <= 10 allowed inputs

- **Mutable ADTs**
  - Must also test that mutated values were updated correctly

Task 1 - Two Sides of The Same Join

```
/** method:
 * Join the two given lists into a single one
 * @requires first /= null, second /= null
 * ...
 */
public static List<Integer> join(List<Integer> first, List<Integer> second);
```

specifications:

```
@modifies first                                          // Spec B
@return first ++ second

@modifies first, second                                  // Spec C
@return first ++ second

@modifies first                                          // Spec D
@effects first = first_0 ++ second
@return first_0 ++ second

@modifies first, second                                  // Spec E
@effects first = first_0 ++ second
@return a list
```

Task 1 - Two Sides of The Same Join

a) Fill in the following table explaining the relationships between each pair of specifications. Write an "S" for if the spec on left (the row) is stronger than the name on top (the column), a "W" if it is weaker, and a "—" if they are incomparable.

	A	B	C	D	E
A	X				
B		X			
C			X		
D				X	
E					X

Task 1 - Two Sides of The Same Join

a) Fill in the following table explaining the relationships between each pair of specifications. Write an "S" for if the spec on left (the row) is stronger than the name on top (the column), a "W" if it is weaker, and a "—" if they are incomparable.

	A	**B**	**C**	**D**	**E**
A	X	S	S	—	—
B	W	X	S	W	—
C	W	W	X	W	—
D	—	S	S	X	S
E	—	—	—	W	X

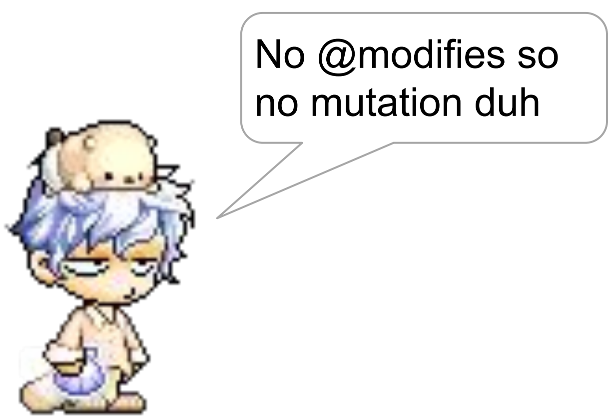
Task 1 - Two Sides of The Same Join

b) Not every combination of `@modifies`, `@effects`, and `@return` behaviors appearing in the specifications on the previous page would be sensible. For example, consider the following specification:

```
@effects first = first_0 ++ second
@return second
```

What is wrong with this specification? Why shouldn't we use it?

The lack of a `@modifies` means the specification promises not to modify anything. However, the `@effects` states that it *will* modify `first`. These statements are contradictory.



Task 2 - Test, Ice, Compression, Elevation

a) First, consider a version of `join`, which does not mutate either argument:

```
public static List<Integer> join(List<Integer> first, List<Integer> second) {
    List<Integer> newList = new ArrayList<>();
    newList.addAll(first);
    newList.addAll(second);
    return newList;
}
```

Fill in the missing parts of the following JUnit test for this version of `join`.

```
    @Test
    public void testJoin() {
        List<Integer> list1 = Arrays.asList(new int[] { 1, 2 });
        List<Integer> list2 = Arrays.asList(new int[] { 3, 4 });
        assertEquals(_Arrays.asList(new int[] 1, 2, 3, 4)__, join(list1, list2));

        List<Integer> list3 = Arrays.asList(new int[] { 1 });
        List<Integer> list4 = Arrays.asList(new int[] { 2, 3, 4 });
        assertEquals(_Arrays.asList(new int[] 1, 2, 3, 4)__, join(list3, list4));
    }
```

Task 2 - Test, Ice, Compression, Elevation

b) Next, consider the following version of `join`, which mutates `first` and does not return anything.

```
/**
 * @modifies first
 * @effects first = first_0 ++ second
 */
public static List<Integer> join(List<Integer> first, List<Integer> second) {
    first.addAll(second);
}
```

Rewrite the JUnit test above to use this new definition of `join` on the same inputs as above.

Task 2 - Test, Ice, Compression, Elevation

```
@Test
public void testJoin() {
    List<Integer> list1 = Arrays.asList(new int[] { 1, 2 });
    List<Integer> list2 = Arrays.asList(new int[] { 3, 4 });
    join(list1, list2);
    assertEquals(Arrays.asList(new int[] {1, 2, 3, 4}), list1);

    List<Integer> list3 = Arrays.asList(new int[] { 1 });
    List<Integer> list4 = Arrays.asList(new int[] { 2, 3, 4 });
    join(list3, list4);
    assertEquals(Arrays.asList(new int[] {1, 2, 3, 4}), list3);
}
```

Task 2 - Test, Ice, Compression, Elevation

c) This version should be longer than before. Why is that the case?

Since the function doesn't return the answer (it modifies the list instead), you have to put the call to `join` on a different line than the call to `assertEquals`.

Task 2 - Test, Ice, Compression, Elevation

d) Finally, consider the version of `join`, which modifies both `first` and `second`.

```
public static List<Integer> join(List<Integer> first, List<Integer> second) {
    while (!second.isEmpty()) {
      first.add(second.get(0));
      second.remove(0);
    }
    return first;
}
```

Rewrite the JUnit test again to properly test this new definition of `join`.

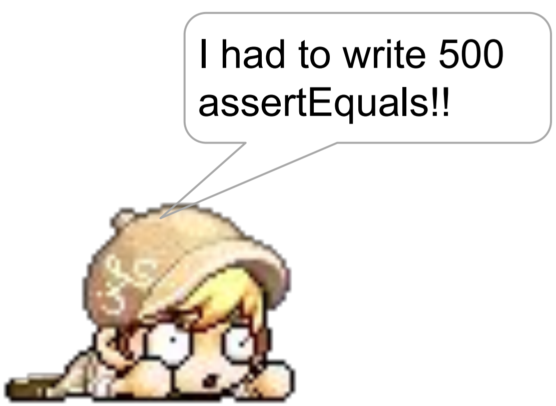
Task 2 - Test, Ice, Compression, Elevation

```
@Test
public void testJoin() {
    List<Integer> list1 = Arrays.asList(new int[] { 1, 2 });
    List<Integer> list2 = Arrays.asList(new int[] { });
    assertEquals(Arrays.asList(new int[] {1, 2 }), join(list1, list2));
    assertEquals(Arrays.asList(new int[] {1, 2 }), list1);
    assertEquals(Arrays.asList(new int[] {}), list2);

    List<Integer> list3 = Arrays.asList(new int[] { 3 });
    assertEquals(Arrays.asList(new int[] {1, 2, 3 }), join(list1, list3));
    assertEquals(Arrays.asList(new int[] {1, 2, 3 }), list1);
    assertEquals(Arrays.asList(new int[] {}), list3);

    List<Integer> list4 = Arrays.asList(new int[] { 4, 5, 6 });
    assertEquals(Arrays.asList(new int[] {1, 2, 3, 4, 5, 6}),
        join(list1, list4));
    assertEquals(Arrays.asList(new int[] {1, 2, 3, 4, 5, 6}), list1);
    assertEquals(Arrays.asList(new int[] {}), list4);
}
```

Task 2 - Test, Ice, Compression, Elevation

e) This version should be the longest yet. What are the reasons for that?

We now have to test not only the return value or one mutated object but that both (two!) of the lists were properly updated. Furthermore, since the implementation uses a loop, we also have to add an additional test case to get loop coverage.



The next problem concerns the following ADT:

```
/**
 * Represents a **mutable** collection of integers.
 *
 * Clients can think of a set as a list of integers that contains no duplicates.
 * The order of the integers is important and the "pop" operation promises
 * to remove the first element in the list.
 */
public class MutableIntSet {
  /**
   * Determines whether n is in the list.
   * @param n the number to look for in the list
   * @returns contains(n, obj), where
   *     contains(n, nil)    := false
   *     contains(n, m :: L) := true           if m = n
   *     contains(n, m :: L) := contains(n, L) if m /= n
   */
  public boolean contains(int n);

  /**
   * Adds n to the list if not already present.
   * @param n the number to add to the new list.
   * @modifies obj
   * @effects obj = add(n, obj_0), where
   *     add(n, L) := L       if contains(n, L)
   *     add(n, L) := n :: L  if not contains(n, L)
   */
  public void add(int n);

  /** Removes and returns the first element in the collection. .... */
  public int pop();
}
```

Task 3 - Good News and Add News

Answer the following questions about the specification of `MutableIntSet`. Assume that `T` is an instance of this class whose abstract state is $1 :: 2 :: 3 ::$ nil.

a) Would `T.add(3)` actually change `obj`? If not, why is that allowed when it says `@modifies obj`.

`@modifies` says that add *may* or *can* modify `obj` but it is not a promise that it does so. For example, in this case we know `obj` would not be modified (via its spec) since the list already contains 3.

b) Now, consider a call `T.add(4)`. Explain how the operation of `MutableIntSet.add` differs from that of `IntSet.add` from Homework 3.

`IntSet.add` returns $n ::$ obj, whereas `MutableIntSet.add` actually changes the abstract state (`obj`) into that value.

Task 3 - Good News and Add News

c) What is the abstract state of T after the following code[1]:

```
T.add(4);
T.add(2);
T.add(0);
```

The resulting state would be $0 :: 4 :: 1 :: 2 :: 3 ::$ nil.

d) Write a specification for the method pop. It should return the head of the list and change the abstract state to be the tail of the list.

```
/**
 * Removes and returns the first element in the collection.
 * @requires len(obj) /= 0
 * @modifies obj
 * @effects obj_0 = n :: obj
 * @returns n
 */
public int pop();
```

Attendance QR code

Happy (early) Halloween!