CSE 331

Mutation and Data Abstraction

James Wilcox

with thanks to Kevin Zatloukal for many slides

331 So Far…

- Saw how to implement ADTs without mutation
- Introducing more mutation going forward
  - core idea is that mutation makes things harder
- Introduced **local variable** mutation last time
  - causes **some** difficulty for *implementers*

    need to reason line-by-line for any variable that is mutated
  - causes **no** difficulty for *clients*

    they literally cannot tell the difference

When we mutate objects and arrays...

- Objects and arrays are "heap" data
  - can still be in use after the call returns
- Mutation of heap data is different
  - clients can see that mutation occurred!
- So, we must also update specifications
  - need to explain any possible mutation that may happen

    by default, nothing is being mutated
  - higher likelihood of potential bugs

    miscommunication between programmers is a common cause
  - these will be harder to debug

Plan for today

Learn how to specify heap mutation for clients

1. Mutation in simple functions (revisit Topic 1)
2. Mutation in ADTs (revisit Topic 3)

Mutation of Arguments

Recall: Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- Item 1 is the precondition
  - explained in `@param` and `@requires`

- Items 2-3 are the postcondition
  - explained in `@return` and `@throws`

Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- The postcondition can also include mutation
  - client will see that something argument was changed
  - explained in `@modifies` and `@effects`

Describing Mutation in Specifications

- List anything that may change in `@modifies`
  - anything not listed is assumed **not modified**
  - no `@modifies` means nothing is mutated

- Results of the mutation listed in `@effects`
  - promises about the state when the call returns
  - no `@effects` means any change is possible

```
// @modifies A
// @effects all entries of A set to zero
void clear(int[] A)
```

Example 1

```
/**
 * Changes the first instance of v in A to w
 * @param A The list to look in. Must be non-null
 * @param v The value to look for
 * @param w The value to replace the first v with
 * @modifies A
 * @effects changes A[i] to w, where i is the
 *      smallest index with A[i] = v, and leaves
 *      A[j] unchanged for all j != i
 * @throws NotFound if no such index i exists
 */
void changeFirst(List<Integer> A, int v, int w)
```

Recall: Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @return A ++ B
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B)
```

How would we change this to mutate instead?

Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @modifies A
 * @effects A = A_0 ++ B
 */
void concat(List<Integer> A, List<Integer> B)
```

We are now using Floyd logic in the spec!

What about a version that modifies B instead?

Is there any scenario where *both* arguments are modified?

Example 3

```
/**
 * Returns the number of common elements in both
 * A and B. Sorts A and B in the process.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 *
 *
 *
 *
 */
int commonElems(List<Integer> A, List<Integer> B)
```

How should we specify this?

Example 3

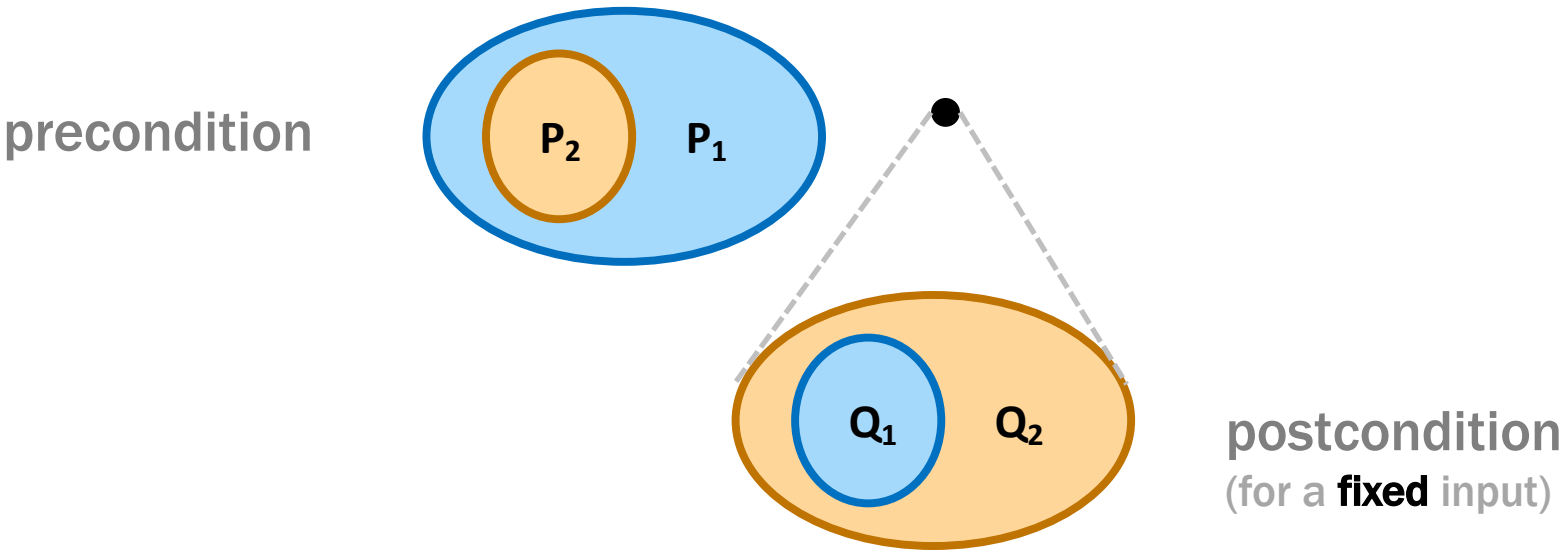
```
/**
 * Returns the number of common elements in both
 * A and B. Sorts A and B in the process.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @modifies A, B
 * @effects A is sorted and B is sorted
 * @returns the number of indexes i such that
 *       A[i] also appears in B somewhere
 */
int commonElems(List<Integer> A, List<Integer> B)
```

Recall: Comparing Specifications

- Specification S_1 is stronger than S_2...
  - whenever is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2
- Changing from S_2 to S_1 (strengthening)...
  - cannot break any clients!
  - client works with any implementation satisfying S_2 and that includes anything satisfying S_1
- But what does this mean...
  - in terms of precondition and postcondition

Recall: Comparing Specifications

- Specification S_1 is stronger than S_2 if it has…
  - a weaker precondition and the same postcondition
  - a stronger postcondition and the same precondition
  - (or both)



Comparing Specifications With Mutation

- Specification S_1 is stronger than S_2 if it has…

- A stronger postcondition:
  - adds more to `@returns`
  - adds more to `@effects`
  - removes from `@modifies`

    promise is **not** to modify anything not listed

- A weaker precondition:
  - no change here

Example 4

```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S1
// @modifies A, B
// @effects A is sorted and B is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S2
// @modifies A, B
// @effects
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_1 relate to S_2?

Example 5

```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S3
// @modifies A, B
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S4
// @modifies A
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_3 relate to S_4?

Example 5

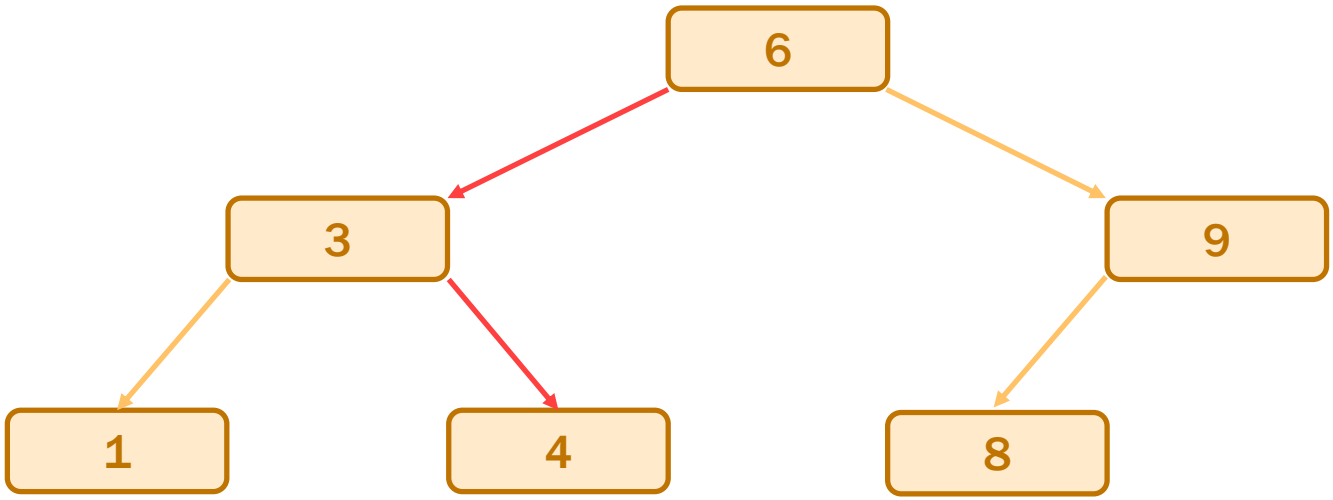
```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S1
// @modifies A, B
// @effects A is sorted and B is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S4
// @modifies A
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_1 relate to S_4?

Aliasing

Recall: Binary Search Trees

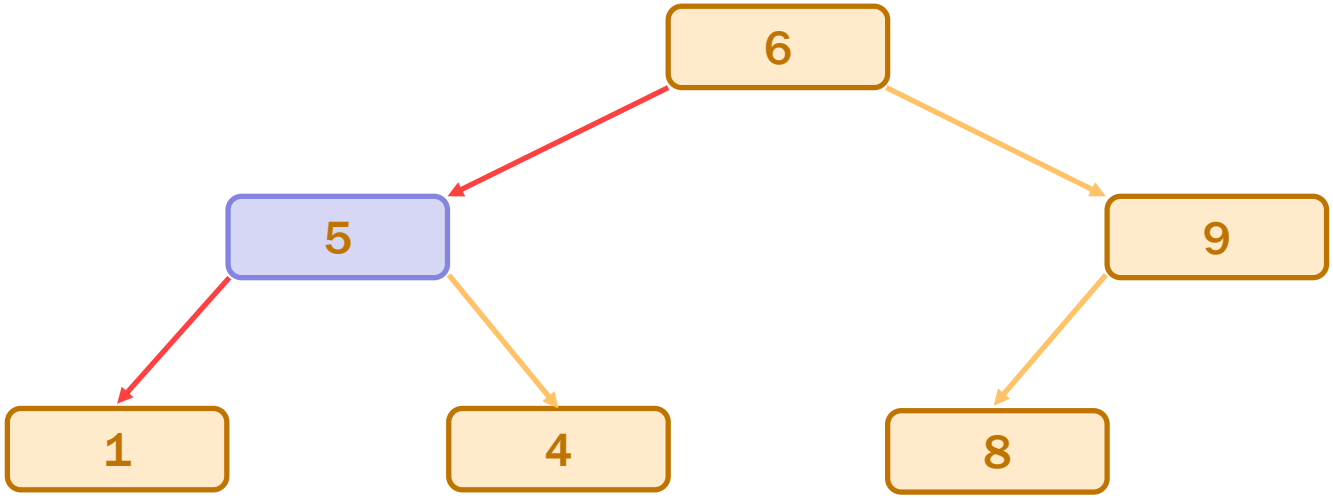
- Consider the following tree
  - searching for "4" proceeds as follows:



- Suppose someone changed "3" into "5"...

Recall: Binary Search Trees

- Suppose someone changed "3" into "5"...
  - now this happens when we search for "4":



  - It can no longer be found!

    Doesn't crash. It's just not found.
  - Problem doesn't occur on the line with the change

Scary Bugs

- Do not fear crashes
  - often no debugging at all

    get a stack trace that tells you exactly where it went wrong

- Do fear unexpected mutation
  - failure will give you no clue what went wrong

    will take a long time to realize the BST invariant was violated by mutation
  - bug could be almost anywhere in the code

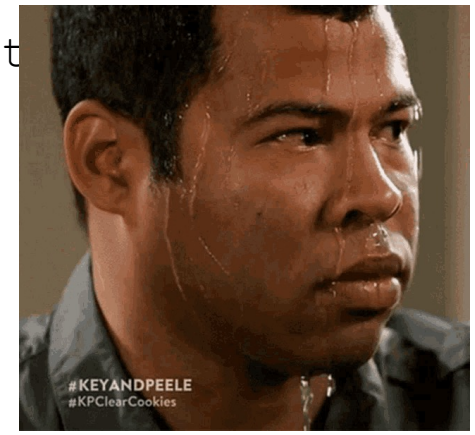
    anyone who mutates a `TreeNode` could have caused it
  - could take *weeks* to track it down

Another Example

```
class Name {
  private String first;
  private String last;

  public String toString() {
    return first + " " + last;
  }

  public void capitalize() {
    this.first = first.substring(0, 1).toUpperCase()
        + first.substring(1);
    this.second = second.substring(0, 1).t
        + second.substring(1);
  }
}
```

Somewhere else...

```
Map<Name, Integer> M;
```



Even Worse in C/C++

- C/C++ strings are mutable
  - commonly used as map keys
  - this sort of bug is still very common

- Java strings are immutable
  - was hugely controversial at the time

    in retrospect, it was clearly a good idea
  - other mutable types can still be used as keys

Aliases

- Extra references to an object are called "aliases"
  - possible for any reference type
- Aliases are fine when objects are *immutable*
  - we don't care if someone else reads the data
  - we only care if someone mutates it
- Aliases are scary when objects are **mutable**...
  - creates the potential for failures far from bugs
  - that means painful debugging

Mutable Heap State

- "With great power, comes great responsibility"
  - Uncle Ben

- With aliases to mutable heap state:
  - gain efficiency in some cases
  - must keep track of every alias that could mutate that state

    any alias, anywhere in the *entire* program could cause a bug

- EJ 17: minimize mutability in classes

Easy Ways to Stay Safe

1. Do not mutate heap state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases…
   - create the state in your constructor and **don't share** it

```
class MyClass {
  // RI: vals is sorted
  private String[] vals;

  public MyClass() {
    this.vals = new String[10];  // only reference
    …
  }
```

Easy Ways to Stay Safe

1. Do not use mutable state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases to *mutable* state
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

ensures only one reference to the object (no aliases)

- For 331, mutable aliasing across files is a bug!
  - gives other parts the ability to break your code
  - we will stick to these simple strategies for avoiding it

An Advanced (Two-Stage) Approach

- Mutable object has only one reference (owner)
  - one reference that is allowed to use & mutate it
- Object is eventually "frozen", making it immutable
  - no longer necessary to track ownership
- Example: Java's `StringBuilder` vs `String`
  - `StringBuilder` is mutable (be careful!)
  - `StringBuilder.toString` returns the value as a `String`
  - `String` is immutable

Rules of Thumb

Client Side

1. Data is small
   - anything on screen is O(1)
2. Aliasing is common
   - UI design forces modules
   - data is widely shared

Rule: avoid mutation

- create new values instead
- performance will be fine

Server Side

1. Data is large
   - efficiency maters
2. Aliasing is avoidable
   - you decide on modules
   - data is not widely shared

Rule: avoid aliases

- do not allow aliases to your data
- hand out copies not aliases
- (good enough for us in 331)

Another Alternative

- With `List`, a third option is sometimes used:

```
public List<String> values() {
  return Collections.unmodifiableList(this.vals);
};
```

  - throws an exception when mutators are called
  - runs in $O(1)$ time instead of $O(n)$ to copy

Can this change break the client?

Another Alternative

- This can break clients
  - this works with a copy

```
MyClass m = …;
List<String> list = m.values()
list.add("another");
```

  - but not with `UnmodifiableList`

- Specification must make clear the behavior
  - how do the two options relate?

Another Alternative

- These two are incomparable
  - they have differing behavior
  - client can work with one but not the other and v.v.
- How is this possible when both return `List`?
  - the unmodifiable list does not implement `List`!

    the spec doesn't let you throw on any call to `add`
  - this is a terrible idea

    but occasionally necessary in extreme circumstances
- Really these are different return types
  - would be better to make then different interfaces

Unmodifiable View

- Unmodifiable list is a "view" of the underlying list
- It changes whenever the underlying list changes
  - updates to that list show up in the view immediately
  - it is not a copy of the data at that point
- This can lead to difficult bugs
  - do not use such a view as a key in a map
  - any alias to it can mutate it at any point

Unmodifiable View

- Why would someone do this?
- Like most CS bugs, it is for performance
  - we all know that $O(1)$ is better than $O(n)$
- But most client uses are $O(n)$ anyway!
  - client probably wants to loop through the list
  - in that case, there is no $O(..)$ gain to
- We will stick to immutable or copying (no aliases)

Calling Methods in Floyd Logic

Reasoning about Function Calls

- Not too difficult if the function is…
  1. defined for all inputs
  2. defined imperatively
  3. no arguments are mutated

Reasoning about Function Calls

- For the simplest case only…
- Forward reasoning rule is

$\{\{ P \}\}$

```
x = Math.sin(a);
```

$\{\{ P[x \mapsto x_0] \text{ and } x = \sin(a) \}\}$

```
// @param x
// @return sin(x)
double sin(double x)
```

- Backward reasoning rule is

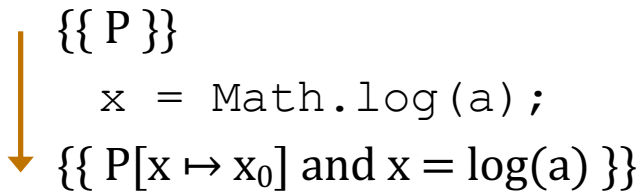
$\{\{ Q[x \mapsto \sin(a)] \}\}$

```
x = Math.sin(a);
```

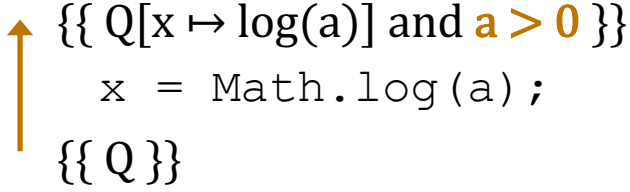
$\{\{ Q \}\}$

Reasoning about Function Calls

- **Preconditions must be checked**
  - not valid to call the function on disallowed inputs
- **Forward reasoning rule is**



- **Backward reasoning rule is**



Must also check $a > 0$

```
// @param x with x > 0
// @return log(x)
double log(double x)
```

Reasoning about Function Calls

- Applies to functions we define with imperative specs

```
// @param n a non-negative integer
// @returns square(n), where
//        square(0) := 0
//      square(n+1) := square(n) + 2n + 1
public int square(int n) {..}
```

- Reasoning is the same. E.g., forward rule is

{{ P }}

```
x = square(n);
```

{{ $P[x \mapsto x_0]$ and $x = \text{square}(n)$ }}

Must also check that `n` is non-negative

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ _____ }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

inverting operation gives $r^2 = x + 2$

c.f. to when $\mathrm{r} = r_o + 1$ and $\mathrm{P}(r_o)$

here we have $\mathrm{r} = \sqrt{r_o}$ and $r_o = x + 2$

second fact is *already* solved for r_o
so we can substitute right into left
instead of left into right

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ x ≥ 0 and r = √(x + 2) + 1 }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

$$\begin{aligned} r &= x + 2 \\ &\geq 0 + 2 \qquad \text{since } x \geq 0 \\ &\geq 0 \end{aligned}$$

this looks good

what did we forget?

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ ____________ }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________________ }}
    double r = x + 2;
    {{ √r + 1 = √(x + 2) + 1 and r ≥ 0 }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
```

$\{\{ x \geq 0 \}\}$

$\{\{ \sqrt{x+2} + 1 = \sqrt{x+2} + 1 \text{ and } x + 2 \geq 0 \}\}$

check this implication

```
    double r = x + 2;
```

$\{\{ \sqrt{r} + 1 = \sqrt{x+2} + 1 \text{ and } r \geq 0 \}\}$

```
    r = Math.sqrt(r);
```

$\{\{ r + 1 = \sqrt{x+2} + 1 \}\}$

```
    r = r + 1;
```

$\{\{ r = \sqrt{x+2} + 1 \}\}$

```
    return r;
}
```

$x \geq 0$ implies $x + 2 \geq 0$

Reasoning about ADT Calls

- ADT methods are calls involving abstract states
- Forward reasoning rule is

$\{\{ P \}\}$

```
L = L.cons(x);
```

$\{\{ P[L \mapsto L_0] \text{ and } L = x :: L_0 \}\}$

```
// @return x :: obj
FastList cons(int x)
```

L is a mathematical list

- Backward reasoning rule is

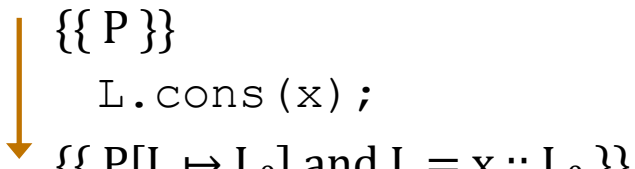
$\{\{ Q[L \mapsto x :: L] \}\}$

```
L = L.cons(x);
```

$\{\{ Q \}\}$

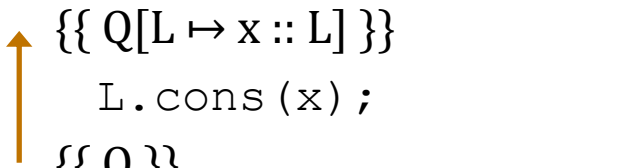
Reasoning about ADT Calls

- Very little changes with mutators…
- Forward reasoning rule is




```
// @modifies obj
// @effects obj = x :: obj0
void cons(int x)
```

- Backward reasoning rule is



`@effects` says it is an assignment
so we get an identical result

Example Calls with Declarative Specs

```
// @returns x such that x >= a and x >= b
public int max(int a, int b) {..}
```

- **Forward reasoning rule is**

{{ P }}
  x = **max**(a, b);
{{ P[x ↦ x_0] and x ≥ a and x ≥ b }}

- **Backward reasoning rule is**

{{ a > 0 }}
  x = **max**(a, b);
{{ a > 0 and 2x ≥ a + b }}

Must check that x ≥ a and x ≥ b **implies** 2x ≥ a + b

Function Calls with Declarative Specs

```
// @requires P2                  -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
public int f(int a, int b) {..}
```

- **Forward reasoning rule is**

{{ P }}
  x = f(a, b);
{{ P[x ↦ x_0] and R }}

Must also check that P implies P_2

- **Backward reasoning rule is**

{{ Q_1 and P_2 }}
  x = f(a, b);
{{ Q_1 and Q_2 }}

Must also check that R implies Q_2

Q_2 is the part of postcondition using "x"