CSE 331

Mutation of Heap State

James Wilcox and Kevin Zatloukal

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 <u>no</u> 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 **<u>not modified</u>**
  - 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

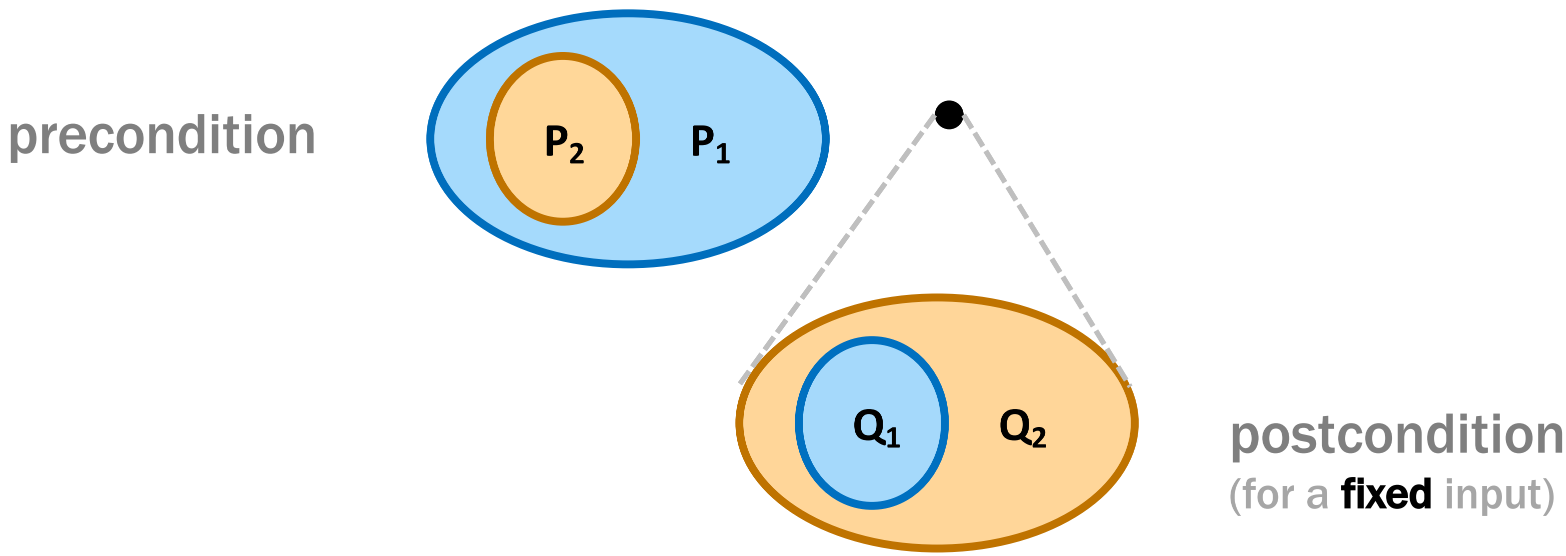
```
/**
 * Returns the number of common elements in both
 * A and B. Sorts A and B in the process.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @modifies A, B
 * @effects A is sorted and B is sorted
 * @returns the number of indexes i such that
 *      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?

Mutation in ADTs

Recall: Mutable vs Immutable ADTs

	Immutable	Mutable
observers	✅	✅
mutators	❌	✅
producers	✅	❌ (usually not)

- Sensible to pick one or the other
  - would be dangerous to provide both

    will see why later on

Recall: Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {

  // Returns the last element of the list (O(1) time)
  // @requires obj != nil
  // @return last(obj)
  int getLast();

  // Returns the object as a regular list of items.
  // @return obj
  List getList();
```

observer (getLast)

observer (getList)

Recall: Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {
  …
  /**
   * Returns a new list with x in front of this list.
   * @return x :: obj
   */
  FastList cons(int x);
```

producer

- How do we make this a mutator?

Specifying a Mutable FastList

```
/**
 * A mutable list of integers that can retrieve the
 * last element in O(1) time.
 */
interface MutableFastList {
  …
  /**
   * Adds x to the front of this list.
   * @modifies obj
   * @effects obj = x :: obj_0
   */
  void cons(int x);
```

- Changes obj to have x at the beginning

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return x */
  double getX();

  /** @return y */
  double getY();
```

- Abstract state *is* a pair (x, y)
  - i.e., we have $(x, y) := \text{obj}$
  - so, we can refer to "x" and "y"

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x^2 + y^2)^(1/2) */
  double getR();

  /** @return arctan(y/x) */
  double getTheta();
```

- Imperative specifications
  - code may or may not actually do these calculations
  - `PolarPoint` just returns the value in a field

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x + dx, y + dy) */
  Point shiftBy(double dx, double dy);
```

- How do we make this a mutator?

Specifying a Mutable Point

```
/** Represents a mutable (x, y) point in 2D space. */
interface MutablePoint {

  /**
   * Moves the point right by dx and up by dy
   * @modifies obj
   * @effects obj = (x_0 + dx, y_0 + dy)
   */
  void shiftBy(double dx, double dy);
```

Recall: Immutable Queue

- A queue is a list that can *only* be changed two ways:
  - add elements to the front
  - remove elements from the back

```
// List that only supports adding to the front and
// removing from the end
interface NumberQueue {

  // @return len(obj)
  int size();

  // @return [x] ++ obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

Which method(s) change in a mutable version?

```
class DequeueParts {
  public final NumberQueue Q;
  public final int x;
}
```

Mutable Queue

```
// @return [x] ++ obj
NumberQueue enqueue(int x);
```

- How do we make this mutable?

```
// @modifies obj
// @effects obj = [x] ++ obj_0
void enqueue(int x);
```

Mutable Queue

```
// @requires len(obj) > 0
// @return (x, Q) with obj = Q ++ [x]
DequeueParts dequeue();
```

- How do we make this mutable?

```
// @modifies obj
// @effects obj_0 = obj ++ [x]
// @return x
int dequeue();
```

Mutable Queue

- Note the symmetry between these operations:

```
// @modifies obj
// @effects obj = [x] ++ obj_0
void enqueue(int x);

// @modifies obj
// @effects obj_0 = obj ++ [x]
// @return x
int dequeue();
```

Which one of these is declarative?

Converting Between Mutators and Producers

- We can transform between these in general
  - assume that "T" is our interface

```
// @return f(obj, x)
T produce(int x);
```

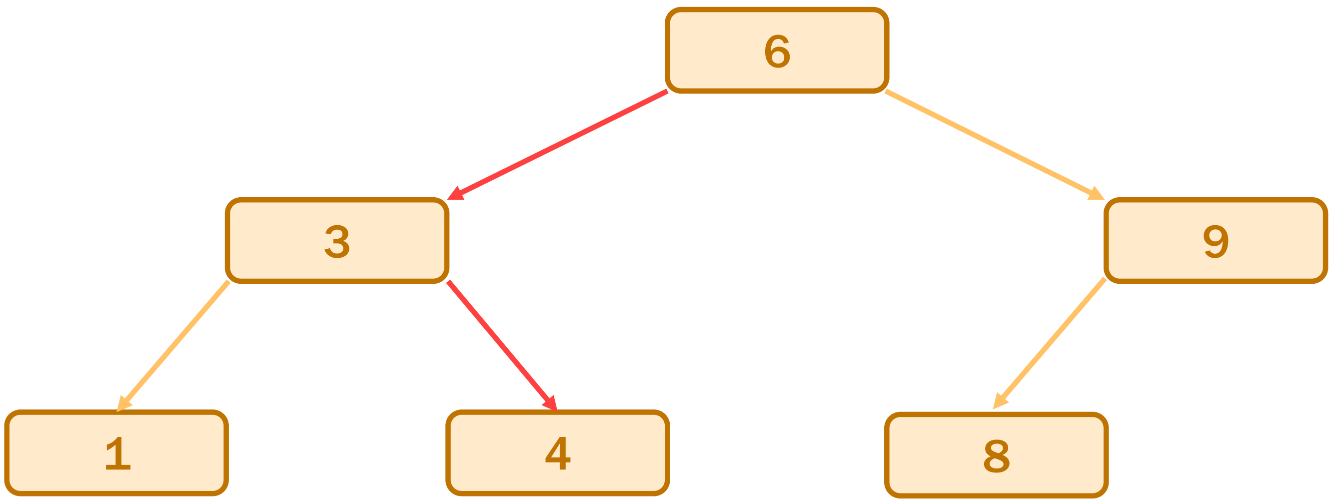
1. change return type
2. change @return expression into @effects obj = expression

```
// @modifies obj
// @effects obj = f(obj_0, x)
void mutate(int x);
```

Aliasing

Recall: Binary Search Trees

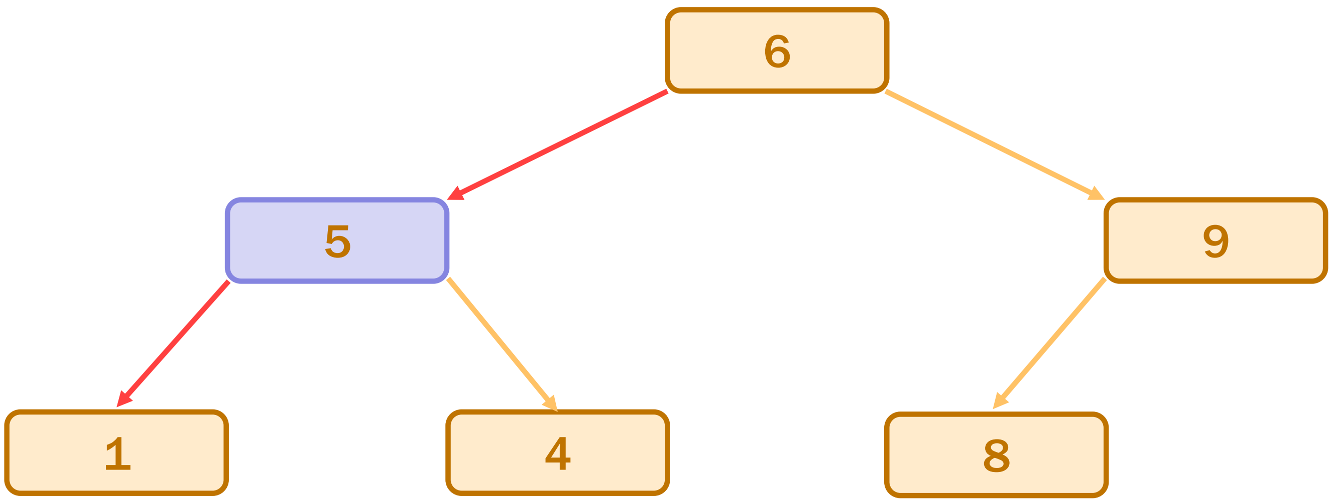
- Consider the following tree
  - searching for "4" proceeds as follows:



- 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

- <u>EJ 17</u>: 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

- Not enough just to declare it "`private`"

```
class MyClass {
  // RI: vals is sorted
  private String[] vals;
  …
  public String[] values() {
    return this.vals;
  };
```

this is "representation exposure"
we wil treat it as a bug

  – anyone can get an alias by calling `values()`

- "`private`" is a clue that aliases might be bad

Easy Ways to Stay Safe

2. Do not allow aliases

(a) do not hand out aliases yourself

- return copies instead

```
class MyClass {
  // RI: vals is sorted
  private String[] vals;
  …
  public String[] values() {
    return this.vals;                       // unsafe!
    return Arrays.copyOf(this.vals,   // make a copy
        this.vals.length);
  };
```

Easy Ways to Stay Safe

2. Do not allow aliases

(b) make a copy of anything you want to keep

– does not matter if the caller mutates the original

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

  …

  // @requires A is sorted
  public MyClass(String[] A) {
    this.vals = A;                      // unsafe!
    this.vals = Arrays.copyOf(A,        // make a copy
        A.length);
  };
```

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 <u>one</u> reference to the object (no aliases)

- For 331, mutable aliasing across files is a <u>bug</u>!
  - 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)

Using List

- Same issue arises with `List` as with arrays

```
class MyClass {
  // RI: vals is sorted
  private List<String> vals;

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

  – since a List is mutable, we cannot create aliases

Another Alternative

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

```
class MyClass {
  // RI: vals is sorted
  private List<String> vals;

  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)

Module Design

Module Design

"Designing modules is the heart of software design."

— Michael Ernst

- In Java, a "module" is a file or a top-level class
- Module design is an enormous subject
  - can look for many properties such as decomposability, composability, understandability, continuity, isolation
- We will keep things simpler…

Module Design

- Modules should have
  - high cohesion
  - low coupling

- Cohesion: the parts go together
  - they all serve one purpose or represent one concept
  - examples: an ADT, java.util.Arrays
  - non-example: one class for sorting, drawing, & printing
  - primarily about the specification

Module Design

- Modules should have
  - high cohesion
  - low coupling

- Coupling: the parts only understandable together
  - must learn both to understand either
  - example: an immutable ADT
  - non-example: a mutable ADT that allows aliases

    must understand how all aliases are used to know if it's correct
  - primarily about the implementation
  - will see another non-example next time..

Coupling Is Bad

- Coupling makes the code less understandable
  - truth for both humans and AI
  - highly coupling becomes "spaghetti code"
  - often shows up as a "god class"
- Coupling makes the code hard to change
  - all the interrelated parts may require changes
- Coupling creates potential for painful debugging
  - bugs in one piece can cause failures in another
  - e.g., any misuse of an alias can break use by any other alias

Subclasses

Subclasses

- Subclassing is a means of sharing code
  - subclass gets parent fields & methods (unless overridden)

```
class Product {
  private String name;
  private int price;
  public String getName() {return name; }
  public int getPrice() { return price; }
}

class SaleProduct extends Product {
  private float discount;
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Subclasses

- Subclassing is a surprisingly dangerous feature
- Subclassing tends to break modularity
  - creates tight coupling between super- and sub-class
  - often see the “fragile base class” problem
    changes to super class often break subclasses
- Let’s see some examples…

Example 1: Tight Coupling

```
class Product {
  private int price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return getPrice() < p.getPrice();
  }
}

class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

– looks okay so far...

Example 1: Tight Coupling

```
class Product {
  private int price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return this.price < p.price;
  }
}
```

Made it faster by eliminating a method call!

```
class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

What's wrong?

Oops! Broke the subclass

Example 2: Tight Coupling

```
class InstrumentedHashSet extends HashSet<Integer> {
  private static int count = 0;

  public boolean add(Integer e) {
    count += 1;
    return super.add(e);
  }

  public boolean addAll(Collection<Integer> c) {
    count += c.size();
    return super.addAll(c);
  }

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

– **what could possibly go wrong?**

Example 2: Tight Coupling

```
InstrumentedHashSet S = new InstrumentedHashSet();
System.out.println(S.getCount());  // 0
S.addAll(Arrays.asList(1, 2));
System.out.println(S.getCount());  // 4?!?
```

- what does this print?

- **What is printed depends on `HashSet`'s `addAll`:**
  - **if it calls `add`, then this prints 4**
  - **if it does not call `add`, then this prints 2**
- **Also possible to be dependent on *order* of calls**

Subclassing Creates Tight Coupling

- Creates tight coupling between super- and sub-class
- Example 1: super-class needs to know about subclass
  - direct field access in parent breaks subclass
- Example 2: subclass needs to know about super-class
  - subclass dependent on which methods call each other
- But wait… There's more!

Example 3: Tight Coupling

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    addToLists(job.getName(), job.getTime());
    total += job.getTime();
  }

  protected addToLists(String name, int time) {
    names.add(name);
    times.add(time);
  }
}
```

Example 3: Tight Coupling

```java
// Makes sure no task is too large compared to rest
class BalancedWorkList extends WorkList {
  protected addToLists(String name, int time) {
    if (times.size() <= 3 || 2*time < total)
      super.addToLists(name, time);  // okay
    } else {
      throw new ImbalancedWorkException(name, time);
    }
  }
}
```

- prevents item from being added if too big
- (also: this subclass is not a subtype!)

Example 3: Tight Coupling

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    int time = job.getTime();  // just one call
    total += time;
    addToLists(job.getName(), time);
  }
}
```

RI not true in method call

- reordering the updates breaks the subclass!
- subclass is using `total` that includes the new job

Example 3: Tight Coupling

- RI can be false in calls to non-public methods
  - only needs to hold at end of the public method

- Requires extra care to get it right
  - method is tightly coupled with the ones that call it
  - needs to know what is true in those methods
    - not enough to just know the RI

- Hard for multiple people to communicate this clearly
  - can be okay when it's all your code
  - very error prone when methods are written by others

Subclassing Creates Tight Coupling

- Creates tight coupling between super- and sub-class
  - direct field access can break subclass
  - subclass dependent on which methods call each other
  - subclass dependent on *order* of method calls
  - subclass can be called when RI is false

- Often see the "fragile base class" problem

- Subclassing is a surprisingly dangerous feature!
  - up to you to verify subclass method specs are stronger
  - up to you to prevent tight coupling

Subclassing is Best Avoided

- EJ 19: either design for subclassing or prohibit it
  - from Josh Bloch, author of (much of) the Java libraries
- We haven’t used subclassing in our ADTs
  - we used interfaces and implemented them with classes
  - these problems are the main reason why we avoided it
- Subclassing is not necessary anyway
  - we have other ways to share code
  - EJ 18: prefer composition to inheritance

Equality

Equity of User-Defined Types

- For any type, useful to know which are "the same"

- Java "==" is not useful on records:

```
new Integer(1) == new Integer(1)  // false!
```

  - this is "reference equality"
  - tells you if they refer to the same object in memory

- Checking if the fields are the same is also wrong
  - different concrete states can have same abstract state

Storing a List In Two Parts

```
// Stores a list, split in two parts.
class ListPair implements List {

  // AF: obj = this.front ++ this.back
  private List front;
  private List back;
```

– three ways of representing the same abstract state:

front	back	front ++ back
[1, 2]	[]	[1, 2]
[1]	[2]	[1, 2]
[]	[1, 2]	[1, 2]

– same abstract states should be considered equal!

Recall: HW3

```
/**
 * Represents an immutable collection of integers.
 *
 * Clients can think of a set as a list of integers. However, they can only ask
 * if an integer is present or not. The order of the integers does not matter.
 * The number of times that an integer appears in the list does not matter.
 */
public interface IntSet {
  /**
   * 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);

  /**
   * Creates and returns a new list containing n as well as all of obj.
   * @param n the number to add to the new list.
   * @returns n :: obj
   */
  public IntSet add(int n);
```

The abstract state allows **duplicates**, but clients can't tell.

Equality on Sets

- Suppose our concrete representation is:

```
// RI: this.list has no duplicates
// AF: obj = this.list
private List list;
```

- Method `add` returns a different list than the spec
  – spec says add(1) on [1] returns [1, 1]
  – if the code add a second 1, abstract state is still [1]

- Need "equal" that says these states are "the same"
  – two abstract states are equal if they contain the same values

equal(L, R) := true **iff** contains(x, L) = contains(x, R) for any x

Equality

- Often useful / necessary to define your own `equal`
  - check if references point to records that are "the same"

- Sensible definition should act like "=" in math:

  1. $\text{equal}(a, a) = T$ for any $a : A$ — reflexive

  2. $\text{equal}(a, b) = \text{equal}(b, a)$ for any $a, b : A$ — symmetric

  3. if $\text{equal}(a, b)$ and $\text{equal}(b, c)$, then $\text{equal}(a, c)$ for any … — transitive

  - (311 alert: this is an "equivalence relation")
  - Java has two more rules for `Object.equal`

Java Equals

- **Jave requires the following parts:**

1. `a.equals(a) = true`
2. `a.equals(b) == b.equals(a)`
3. `a.equals(b) and b.equals(c) means a.equals(c)`
4. `a.equals(null) = false` — asymmetric with null
5. `a.equals(b)` cannot change value unless `a` or `b` is mutated — consistency

Equals in Java

- Every class inherits an `equals` method
  - this implements reference equality

```
public class Object {
  public boolean equals(Object o) {
    return this == o;
  }
}
```

- Make your own `equals` by overriding it:

```
public class MyClass {
  public boolean equals(Object o) {
    // … new code here …
  }
}
```

Example: Duration

- **Define** Duration **to be an amount of time in seconds**
  - **one representation stores separate minutes and seconds**

$$\textbf{type}\ \text{Duration} = \{\text{min} : \mathbb{Z}, \text{sec} : \mathbb{Z}\}\ \ \text{with}\ \ 0 \le \text{sec} < 60$$

  - **second part is a rep invariant**

- **Can define equality on** Duration **this way:**

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\})\ \ :=\ \ (\text{m} = \text{n})\ \text{and}\ (\text{s} = \text{t})$$

  - **true iff these are the same amount of time**

    (wouldn't be true without the invariant)

Example: Duration

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \quad := \quad (m = n) \text{ and } (s = t)$$

- **Does this have the required properties?**
  - reflexive

$$\begin{aligned}
&\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: m, sec: s}\}) \\
&\quad = (m = m) \text{ and } (s = s) && \textbf{def of } \text{equal} \\
&\quad = T \text{ and } T \\
&\quad = T
\end{aligned}$$

proof by calculation
that it holds for any record

  - symmetric

$$\begin{aligned}
&\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \\
&\quad = (m = n) \text{ and } (s = t) && \textbf{def of } \text{equal} \\
&\quad = (n = m) \text{ and } (t = s) \\
&\quad = \text{equal}(\{\text{min: n, sec: t}\}, \{\text{min: m, sec: s}\}) && \textbf{def of } \text{equal}
\end{aligned}$$

Example: Duration

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \quad := \quad (m = n) \text{ and } (s = t)$$

- Does this have the required properties?
  - reflexive yes
  - symmetric yes
  - transitive also yes (but a little long for a slide)

- Good evidence that this is a reasonable definition

Non-Example: “==” in JavaScript

```
0 == "0"      true
0 == ""       true
0 == " "      true
```

- **Which property fails?**
  - transitivity: `"" != " "`

- **Good evidence that this is <u>not</u> a reasonable definition**

Example: Duration in Java

```
// Represents an amount of time measured in seconds
class Duration {
  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  private int min;
  private int sec;

  public boolean equals(Duration d) {
    return this.min == d.min && this.sec == d.sec;
  };
```

- **What is wrong with this?**
  - **it doesn't override** `equals(Object)`

Example: Duration in Java

```
// Represents an amount of time measured in seconds
class Duration {
  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  private int min;
  private int sec;

  public boolean equals(Object o) {
    return this.min == o.min && this.sec == o.sec;
  };
```

- What is wrong with this?
  - it doesn't compile

Example: Duration in Java

```
// Represents an amount of time measured in seconds
class Duration {
  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  private int min;
  private int sec;
  public boolean equals(Object o) {
    if (!(o instanceof Duration))
      return false;

    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
  }
```

- Correct and idiomatic Java

Example: NanoDuration

- **Suppose a subclass also measures nanoseconds**

```
class NanoDuration extends Duration {
  // min: number (inherited)
  // sec: number (inherited)
  private int nano;
  …
```

- **How should we define `equal`?**

Example: NanoDuration

```
class NanoDuration extends Duration {
  // min: number (inherited)
  // sec: number (inherited)
  private int nano;
  public boolean equals(Object o) {
    if (!(o instanceof NanoDuration)) {
     return false;

    NanoDuration n = (NanoDuration) o;
    return this.min === n.min &&
           this.sec === n.sec &&
           this.nano === n.nano;
  }
```

symmetry

- **Which property does this lack?**

Example: NanoDuration

```
Duration d = new Duration(2, 10);
NanoDuration n = new NanoDuration(2, 10, 300);

System.out.println(n.equals(d));  // false
System.out.println(d.equals(n));  // true!
```

– `NanoDuration` **is only equal to other** `NanoDuration`**s**

– `Duration` **can be equal to a** `NanoDuration`
  if they have the same minutes and seconds

Example: NanoDuration

```
class NanoDuration extends Duration {

  public boolean equals(Object o) {
    if (!(o instanceof Duration))
     return false;

    if (!(o instanceof NanoDuration)) {
      Duration d = (Duration) o;
      return this.min == d.min && this.sec == d.sec;
    } else {
      NanoDuration n = (NanoDuration) o;
      return this.min === d.min &&
        this.sec === d.sec && this.nano === d.nano;
    }
  };
```

No! It lacks transitivity

- Fixes symmetry! all good now?

Example: NanoDuration

```
NanoDuration n1 = new NanoDuration(2, 10, 300);
Duration d = new Duration(2, 10);
NanoDuration n2 = new NanoDuration(2, 10, 400);

System.out.println(n1.equals(d)); // true
System.out.println(d.equals(n2)); // true
System.out.println(n1.equals(n2));// false!
```

– **transitivity requires** `n1` **to equal** `n2` **(but it doesn't)**

Example: NanoDuration

- Can fix this instead as follows:
  - have both agree that `Duration` ≠ `NanoDuration`

```
class Duration {
  …
  public boolean equals(Object o) {
    if (!(o instanceof Duration) ||
        (o instanceof NanoDuration))
     return false;

    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
  }
}
```

- This is arguably the most sensible answer…

Example: NanoDuration

- Should have spelled out the abstract states:

```
// Represents an amount of time in nanoseconds
class NanoDuration extends Duration {
  // RI: 0 <= sec < 60 and 0 <= nano < 10000
  // AF: obj = 60,000,000 * this.min +
  //           1,000,000 * this.sec +
  //           this.nano
  private int nano;
}
```

- Abstract states of the two types are different
  - time in seconds vs nanoseconds
  - two different types of things should not be equal

Duration and NanoDuration

- We fixed it… but at what cost?

- `Duration` and `NanoDuration` are tightly coupled
  - the two classes are tightly intertwined

- This usually happens with subclasses
  - saw several different ways they are interdependent
  - very hard to avoid coupling between subclasses

    EJ 19: either design for subclassing or prohibit it
  - better to simply not use it

    find other ways to share code (e.g., shared utility functions etc.)

HashCode in Java

- **Java has another method called** `hashCode`

```
public int hashCode();
```

- **Should override** `hashCode` **and** `equals` **together**
  - **almost certainly a bug to only override** `equals`

Java HashCode

- Java has another method called `hashCode`
  - provided to make `HashMap` etc. work

```
public int hashCode();
```

- Its spec has the following requirements:

1. `a.hashCode()` cannot change value unless `a` is mutated

   self-consistency

2. `a.equals(b) means a.hashCode() == b.hashCode()`

   consistent with equals

   when equals changes, so does hashCode

Equals & HashCode in Java

- Every class inherits a `hashCode` method

```
public class Object {
  public int hashCode() {
    // … consistent with reference equality …
  }
}
```

- When you override `equals`, also override `hashCode`
  - almost certainly a bug to only override `equals`

```
public class MyClass {
  public int hashCode() {
    // … something consistent with new equality …
  }
}
```