Announcements

- Due Today!
  - Lab 1
  - HW 2
- Due Thursday!
  - HW 3
- Due next week
  - HW 4 (on today’s material)
- Lab 2 on Thursday
  - Writing our first iOS app
  - No prelab

IntSet Spec

```swift
/// Overview: An IntSet is a mutable, unbounded set of integers.
/// A typical IntSet is { x1, ..., xn }.
class IntSet {
    /// **Effects**: makes a new IntSet = {}
    public init() {
    }
    /// **Effects**: self_post = self_pre U { element }
    public func add(_ element : Int) -> Bool {
    }
    /// **Effects**: self_post = self_pre - { element }
    public func remove(_ element : Int) {
    }
```

Overview

Abstract State

Creator

Observer

Mutators

CS326 Method Specifications

```swift
/**
  ...
  **Requires**: none (can omit in this case)
  **Modifies**: self
  **Effects**: Changes the first occurrence of oldValue to newValue
  - Parameter oldValue: element to replace.
  - Parameter newValue: what to replace it with.
  - Returns: The first index where oldValue is found, or nil if it does not occur in the list.
*/
func replace(_ oldValue: T, with newValue: T) -> Int? {
    for i in 0..<count {
        if get(i) == oldValue {
            set(i, to: newValue)
            return i
        }
    }
    return nil
}
```

CS 326
Representation Invariants and Abstraction Functions

Stephen Freund
ADTs and Specs

IntSet Implementation. Ok?

```swift
class IntSet {
    private var elems = [Int]()

    public func contains(_ element: Int) -> Bool {
        return elems.contains(element)
    }

    public func add(_ element: Int) {
        elems.append(element)
    }

    public func remove(_ element: Int) {
        if let index = elems.firstIndex(of: element) {
            elems.remove(index)
        }
    }
}
```

IntSet Rep Invariant

```swift
let s = IntSet()
s.add(3)
s.remove(3) // assert !s.contains(3)
```
Rep Invariant for ADT

Client

/// ...
/// A typical IntSet
/// is \{ x_1, \ldots, x_n \}.
class IntSet {

Implementer
class IntSet {
  var elems : [Int]
  ...
  [1, 1, 1] [1, 4, 3] [2, 1] [1, 2] [1, 3, 4] [3, 4, 1]
  [1, 1, 2] [1, 3, 4, 3]
}

Representation Invariant: RI(self) = \{ self.elems has no duplicates \}

Rep Invariant Must Be Preserved

Client

implementer

[1, 1, 1]
[2, 1]
[1, 2]
[1, 3, 4]
[3, 4, 1]
[1, 1, 2]
[1, 3, 4, 3]

object.add(3)

client

RI(Object) ✔
RI(Object') ✔

add(3) ->
elems.append(3)

[1, 3]
[1, 3, 3]

RI(Object) ✔
RI(Object') ✗
**IntSet Rep Invariant**

```swift
class IntSet {
  // Rep Invariant: elems has no duplicates
  private var elems: [Int]()

  public func contains(_ element: Int) -> Bool {
    return elems.contains(element)
  }

  public func add(_ element: Int) {
    if !contains(element) {
      elems.append(element)
    }
  }

  public func remove(_ element: Int) {
    if let index = elems.index(of: element) {
      elems.remove(index)
    }
  }
}
```

**Rep Invariant Must Be Preserved**

- init
- ...
- balance ≥ 0
- balance = \( \text{Sum} \{ \text{tamount} \mid \text{t in transactions} \} \)

**Another Example**

```swift
class Account {
  var balance: Int
  var transactions: [Transaction]
...
}
```

Real-world constraints:
- balance ≥ 0
- balance = \( \text{Sum} \{ \text{tamount} \mid \text{t in transactions} \} \)

Implementation-related constraints:
- forall t in transactions, tcompletionDate != nil
Checking the Rep Invariant

Rule of thumb: check on entry and on exit (why?)

public func remove(_ element : Int) {
    checkRep()
    if let index = elems.firstIndex(of: element) {
        elems.remove(index)
    }
    checkRep()
}

// Verify that elems contains no duplicates.
private func checkRep() {
    for i in 0..<elems.count {
        assert elems.firstIndex(of: elems[i]) == i
    }
}

IntSet V2

class IntSet {
    // Rep Invariant: elems has no duplicates
    private let elems = MutableList<Int>()

    public func contains(_ element: Int) -> Bool {
        checkRep()
        return elems.contains(element)
    }

    public func add(_ element: Int) {
        checkRep()
        if (!contains(element) {
            elems.append(element)
        }
        checkRep()
    }

    public func remove(_ element: Int) {
        checkRep()
        elems.remove(element)
        checkRep()
    }

    /// Returns: A list containing the members of self
    public func getElements() -> MutableList<Int> {
        return elems
    }
}

class MutableList<T> {
    var count : Int
    func get(index: Int) -> T
    func set(index: Int, to: T)
    func append(_ e: T)
    func contains(_ e: T) -> Bool
}

let s = IntSet()
s.add(5)
let elems = s.getElements()
elems.add(5)
s.remove(5)
assert !s.contains(5)

IntSet V3

class IntSet {
    // Rep Invariant: elems has no duplicates
    private let elems = MutableList<Int>()

    public func contains(_ element: Int) -> Bool {
        checkRep()
        return elems.contains(element)
    }

    public func add(_ element: Int) {
        checkRep()
        if (!contains(element) {
            elems.append(element)
        }
        checkRep()
    }

    public func remove(_ element: Int) {
        checkRep()
        elems.remove(element)
        checkRep()
    }

    /// Returns: A list containing the members of self
    public func getElements() -> MutableList<Int> {
        return elems
    }
}
**private Is Not Enough**

Client  ADT

```
 IntSet

MutableList

...

...```

---

**Solution 1: Copy In. Copy Out.**

(assume Point is a mutable ADT)

```java
public class Point {
    var x : Int
    var y : Int
}
```

```java
public class Line {
    private var s : Point
    private var e : Point

    public init(s : Point, e : Point) {
        self.s = Point(s.x, s.y)
        self.e = Point(e.x, e.y)
    }

    public var start : Point {
        return Point(self.s.x, self.x.y)
    }

    ...
}
```

---

**Shallow Copy**

What’s the bug (assuming Point is a mutable ADT)?

```java
class PointSet {
    private var points = MutableList<Point>()

    public getElements() <- MutableList<Point> {
        let result = MutableList<Point>()
        for p in points {
            result.append(p)
        }
        return result
    }
```
**Shallow Copy**

What’s the bug (assuming `Point` is a mutable ADT)?

```java
class PointSet {
    private var points = [Point]()
    public getElements() -> [Point] {
        return points
    }
}
```

**Deep Copy**

```java
class PointSet {
    private var points = MutableList<Point>()
    public getElements() -> MutableList<Point> {
        let result = MutableList<Point>()
        for p in points {
            result.append(Point(x: p.x, y: p.y))
        }
        return result
    }
}
```

**Solution 2: Immutable ADTs**

(immutable `Point`)

```java
public class Line {
    private let s: Point
    private let e: Point

    public init(s: Point, e: Point) {
        self.s = s
        self.e = e
    }

    public var start: Point {
        return self.s
    }
}
```

**Deep Copy Not Needed**

(assuming `Point` is an immutable ADT)

```java
class PointSet {
    private var points = MutableList<Point>()
    public getElements() -> MutableList<Point> {
        let result = MutableList<Point>()
        for p in points {
            result.append(p)
        }
        return result
    }
}
```
Deep Copy Not Needed

(assuming Point is a immutable ADT)

class PointSet {
    private var points = [Point]()
    public getElements() -> [Point] {
        return points
    }
}

Immutability and Design

• Advantages
  – Aliasing does not matter
  – No need to make copies with identical contents
  – Rep invariants cannot be broken
• Sometimes requires different/awkward design

public class MutableLine {
    func move(dx: Int, dy: Int) {
        self.s = Point(self.s.x + dx, self.s.y + dy)
        self.e = Point(self.e.x + dx, self.e.y + dy)
    }
}

Abstract vs Concrete State of ADT

Client
  /// ...
  /// A typical IntSet
  /// is { x1, ..., xn }.
  class IntSet {

Implementer
  class IntSet {
    var elems : [Int]
    ...

  Abstraction Function: AF(self) = { x | x is contained in self.elems. }
IntSet.add(3)

self_post =
self_pre U (3)

{1, 2}

Method
Specification

{1, 2, 3}

Method
Implementation

if !elems.contains(3) {
    elems.append(3)
}

Abstract Equivalence

[ ]

{1}

[1, 3, 7]

{} add(1) [1] add(3) [1, 3] add(7) [1, 7, 3]

Benevolent Side Effects

/// - **Modifies**: *still nothing*
/// - **Returns**: true if and only if element in self
public func contains(_ element: Int) -> Bool {
    if let index = elems.index(of: element) {
        elems.swapAt(0, index)
        return true
    } else {
        return false
    }
}
Benevolent Side Effects

/// - **Modifies**: still nothing
/// - Returns: true if and only if element in self
public func contains(_ element: Int) -> Bool {
    if let index = elems.index(of: element) {
        elems.swapAt(0, index)
        return true
    } else {
        return false
    }
}

Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Leverage math structures when possible

/**
 * A Polygon is a plane figure that is bounded by a finite chain of at least 3 line segments closing in a loop, eg:
 * (x0,y0)→(x1,y1), (x1,y1)→(x2,y2), ..., (xn,yn)→(x0,y0).
 * where (x0,y0)→(x1,y1) denotes a line segment.
 */
class Polygon {
    // AF(self) = { pts[i]→pts[i+1] | i in 0..<pts.count-1 }
    U { pts[pts.count-1]→pts[0] }
    // Rep Inv: points.count >= 3
    var pts: [Point]
}

/**
 * A point (x,y) on the Cartesian plan.
 */
class Point {
    // AF(self) = x is self.x, y is self.y
    let x: Double
    let y: Double
}

Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Introduce names for pieces of abs state
  – often obvious and match public properties and observers

/**
 * Specification Properties**:
 * - x: horizontal coordinate
 * - y: vertical coordinate
 */
class Point {
    // AF(self) = x is self.x, y is self.y
    let x: Double
    let y: Double
}
Writing AFs

• Domain: all concrete values satisfying Rep Inv.
• Range: Introduce names for pieces of abs state

```/**
A URL represents the location of a resource on the network.
***/

**Specification Properties**:
- `protocol`: either http or https
- `hostname`: name of computer holding the resource
- `path`: location of the resource on the host

*/```

class URL {
    // AF(self): let "protocol://hostname/path" = urlString
    let urlString: String
```

Make Abstract State Printable

• Domain: all concrete values satisfying Rep Inv.
• Range: Introduce names for pieces of abs state
• Write description property to show abs state:

```class Point : CustomStringConvertible {
    // AF(self): x is self.x, y is self.y
    let x: Double
    let y: Double

    var description : String {
        return "(\(x), \(y))"
    }
```

```let p = Point(3,4)
    print(pt="(p")
```

Data Abstraction Summary

• **Rep Invariant**: Which concrete values represent abstract values?
• **Abstraction Function**: Which abstract value does a concrete value represent?

• Both are needed to reason about whether a module's implementation satisfies the specification