Where we are

- Basics of Reasoning about code
- Coming up
  - **Specification**: What are we supposed to build?
  - **Design**: Abstraction. Which designs are “better”?
  - **Implementation**: Building code to meet a specification
  - **Testing**: Systematically finding problems
  - **Debugging**: Systematically fixing problems
  - **Maintenance**: How does the artifact adapt over time?
- **Documentation**: What do we need to know to do these things? How/where do we write that down?

Class Interface

class MutableList<T : Comparable> {
  var count : Int { ... }
  func get(index: Int) -> T { ... }
  func set(index: Int, to value: T) -> T { ... }
  func append(_ t : T) { ... }
  ...

  static func isSubsequence(_ part : MutableList<T>, of list: MutableList<T>) -> Bool {
    ...
  }
}
Just Read The Code

static func isSubsequence(_ part : MutableList<T>, of list: MutableList<T>) -> Bool {
    var partIndex = 0
    for element in list {
        if element == part.get(partIndex) {
            partIndex += 1
            if partIndex == part.count {
                return true
            }
        } else {
            partIndex = 0
        }
    }
    return false
}

Just Read The Comments

// Check whether part appears as a contiguous subsequence
// of list.
static func isSubsequence(_ part : MutableList<T>, of list: MutableList<T>) -> Bool {
    var partIndex = 0
    for element in list {
        if element == part.get(partIndex) {
            partIndex += 1
            if partIndex == part.count {
                return true
            }
        } else {
            partIndex = 0
        }
    }
    return false
}

Write Appropriate Specification

// Check whether part appears as a contiguous subsequence
// of list.

• Document Caveats
  // * If list is empty, always returns false
  // * Results may be unexpected if partial matches
  // can happen right before a real match; e.g.,
  // (1,2,1,3) will not be identified as a
  // sub sequence of (1,2,1,2,1,3).

• Or Replace with More Detailed Behaviour
  // This method scans “list” from beginning
  // to end, building up a match for “part”, and
  // resetting that match every time that...

Write Better Code... (And Spec)

// Returns true iff there exist possibly empty
// sequences A, B where
// list = A : part : B
// and “:” is sequence concatenation.
static func isSubsequence(_ part : MutableList<T>, of list: MutableList<T>) -> Bool {
    ...
}
Swift Comments

```swift
/**
   Returns the first index where the specified value appears in the collection.

   After using `index(of:)` to find the position of a particular element in a
   collection, you can use it to access the element by subscripting. This
   example shows how you can modify one of the names in an array of students.

   var students = ["Ben", "Ivy", "Jordell", "Maxime"]
   if let i = students.index(of: "Maxime") {
     students[i] = "Max"
   }
   print(students)
   // Prints ["Ben", "Ivy", "Jordell", "Max"]

   - Parameter element: An element to search for in the collection.
   - Returns: The first index where element is found, or nil
   - Declared In: Array

   */

func index(of element: Element) -> Int? {

    Parameters
    element: An element to search for in the collection.

    Returns
    The first index where element is found, or nil.

    Declared in: Array

    */
```

CS326 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

   */

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
}
```
CS326 Specification Pieces

- **Precondition**: constraints that hold before the method is called (if not, all bets are off)
  - **Requires**: spells out any obligations on client

- **Postcondition**: constraints that hold after the method is called (if the precondition held)
  - **Modifies**: lists objects that may be affected by method; any object not listed is guaranteed to be untouched
  - **Effects**: gives guarantees on final state of modified objects
  - Standard "Returns" tag
  - Standard "Throws": lists possible exceptions and conditions under which they are thrown (won't worry about for now)

CS326 Specifications

```swift
/**
... 
**Requires**: self and other are the same size
**Modifies**: self
**Effects**: the ith element of other is added to the ith element of self
*/
func add(_ other: MutableList<Int>) {
    for i in 0..<count {
        set(i, get(i) + list2.get(i))
    }
}
```

CS326 Specifications

```swift
/**
... 
**Requires**: list1 and list2 are the same size
**Modifies**: none
**Effects**: none
- Parameter list1: ...
- Parameter list2: ...
- Returns: A list of the same size as the parameters, where the ith element is the sum of the ith elements of list1 and list2
*/
static func pointwiseSum(_ list1 : MutableList<Int>,
    _ list2 : MutableList<Int>) -> MutableList<Int> {
    let result = MutableList<Int>()
    for i in 0..<list1.count {
        result.append(list1.get(i) + list2.get(i))
    }
    return result
}
```

CS326 Specifications

```swift
/**
... 
**Requires**: ??
**Modifies**: ??
**Effects**: ??
*/
func uniquify() {
    for i in 0..<count-1 {
        if get(i) == get(i+1) {
            remove(i)
        }
    }
}
```
Satisfaction of a Specification

• Let M be an implementation and S a specification

• \textbf{M satisfies S} if and only if
  – Every behavior of M is permitted by S

• If M does not satisfy S, either (or both!) could be “wrong”
  – Usually better to change the program than the spec

Comparing Specifications

• Specification \textbf{S1 is weaker than S2}, if for all M,
  \[ M \text{ satisfies } S_2 \implies M \text{ satisfies } S_1 \]

• A weaker specification gives greater freedom to the implementer

Which is Weaker? A or B?

```swift
func index(of element: Element) -> Int? {
    for i in 0..<count {
        if get(i) == element {
            return i
        }
    }
    return nil
}
```

\textbf{Specification A}

– requires: value occurs in self
– returns: \text{i such that } get(i) = value

\textbf{Specification B}

– requires: value occurs in self
– returns: \text{smallest } i \text{ such that } get(i) = value

\textbf{Which is Weaker? A or C?}

```swift
func index(of element: Element) -> Int? {
    for i in 0..<count {
        if get(i) == element {
            return i
        }
    }
    return nil
}
```

\textbf{Specification A}

– requires: value occurs in self
– returns: \text{i such that } get(i) = value

\textbf{Specification C}

– returns: \text{i such that } get(i) = value, or nil if value is not in self
Weakening a Specification

- Promise Less
  - Weaker Postcondition
    - Returns clause easier to satisfy
    - More objects in modifies clause
    - Effects clause easier to satisfy
    - Fewer specific exceptions
- Ask more of client
  - Stronger Precondition
    - Requires clause harder to satisfy

(Strengthening: The Opposite)

Which is Better?

- Stronger does not always mean better!
- Weaker does not always mean better!
- Strength of specification trades off:
  - Usefulness to client
  - Ease of simple, efficient, correct implementation
  - Promotion of reuse and modularity
  - Clarity of specification itself
- “It depends”

Two Representations of Points

```java
class Point {
    public float x;
    public float y;
    public float r;
    public float theta;
}
```

Point ADT

```java
public class Point {
    // A 2-d point exists in the plane, ...
    public var x : Double
    public var y : Double
    public var r : Double
    public var theta : Double
    // ... can be created, ...
    public init() // new point at (0,0)
    public init(points : Set<Point>) // centroid
    // ... can be moved, ...
    public func translate(dx: Double, dy: Double)
    public func scaleAndRotate(dx: Double, dTheta: Double)
}
```
Abstract Data Type = Objects + Ops

Poly: Overview and Abstract State

/**
   A Poly is an immutable polynomial with integer coefficients. A typical Poly is
   \( c_0 + c_1 \cdot x + c_2 \cdot x^2 + \ldots \)
*/
public class Poly {

Poly: Creators

    // **Effects**: makes a new Poly = 0
    public init() {

    // **Requires**: n >= 0
    // **Effects**: makes a new Poly = coeff \cdot x^n
    public init(c: Int, n: Int) {

(Note: full specs omitted to save space; style might not be perfect either – focus on main ideas.)

Poly: Observers

    // The degree of self, ie largest exponent with a non-zero coefficient, or 0 if self = 0.
    public var degree : Int

    /**
       **Requires**: d >= 0
       - Returns: The coefficient of the term of self whose exponent is d.
    */
    public func coefficient(for d: Int) -> Int
### Poly: Producers

```swift
/// - Returns: self + q, as a Poly
public func add(_ q: Poly) -> Poly

/// - Returns: self * q, as a Poly
public func mul(_ q: Poly) -> Poly

/// - Returns: -self
public func negate() -> Poly
```

```swift
let p = Poly(2,4)
let q = p.mul(p)
let r = q.negate()
```

### Aside: Operator Overloading

```swift
/// - Returns: p + q
static public func +(p: Poly, q: Poly) -> Poly

/// - Returns: p * q
static public func *(p: Poly, q: Poly) -> Poly

/// - Returns: -p
static public prefix func -(p: Poly) -> Poly
```

```swift
let p = Poly(2,4)
let q = p * p
let r = -q
```

### IntSet: Overview, Abs State, Creator

```swift
/// Overview: An IntSet is a mutable, unbounded set of integers. A typical IntSet is \{ x_1, \ldots, x_n \}.
class IntSet {

/// **Effects**: makes a new IntSet = {}  
public init()
```

### IntSet: Observers

```swift
/// - Returns: true if and only if element in self  
public func contains(element: Int) -> Bool

/// Number of elements in the set  
public var count: Int

/// - Returns: Some element of self.  
/// - Throws: EmptyError if self is empty  
public func choose() throws -> Int
```
IntSet: Mutators

```swift
/// **Modifies**: self
/// **Effects**: self_post = self_pre U { element }
public func add(_ element : Int)

/// **Modifies**: self
/// **Effects**: self_post = self_pre - { element }
public func remove(_ element : Int)
```

Stronger and Weaker Specifications

- Weaker specification:
  - Implementer: Easier to satisfy (more implementations satisfy it)
  - Client: Harder to use (fewer guarantees)

- Stronger specification:
  - Implementer: Harder to satisfy
  - Client: Easier to use (more guarantees, more predictable, can make more assumptions)