ADTs

structure Stack :> sig
  type 'a stack
  val create: 'a stack
  val isEmpty: 'a stack -> bool
  val push: 'a -> 'a stack -> 'a stack
  ...
end = struct ... end

• ADT: hidden representation
• Only access through (implementor-provided) operations
"Exposed" ADTs

(define empty-stack '())
(define (empty? a-stack) (equal? a-stack '()))
(define (push v a-stack) (cons v a-stack))
...

• Client can access representation.
• Only "politeness" prevents this.
• Still useful to organize thinking, make intentions manifest
ADT design process

• Identify abstractions
• Identify operations on abstractions

• Key addition of OO: inheritance...
OOP design process

• Identify abstractions
• Identify operations on abstractions
• Factor into subclass/superclass relationships
  • Common operations across many classes
    -> make into superclasses, inherit

• Factoring is ongoing, iterative process
• Good OO programmers constantly refactor
• Frameworks: libraries that "pre-factor" functionality needed by many clients in a given application domain
When to inherit?

- Inheritance to express kind-of relationships
  - An IconButton is a kind of a Button.
  - Common interface

- Inheritance to reuse code/implementation
  - Stack might inherit from Array
  - Less desirable than organizing for interfaces
    - For long run reuse, factor for interfaces, not implementation.
    - Otherwise, may later find that interface is not exactly suitable.
Concrete vs. abstract

- **Concrete class:**
  - Intended to be instantiated, used directly

- **Abstract class:**
  - Intended to provide common interface or implementation for subclasses
  - Do not instantiate directly
  - In statically typed languages, typically declare abstractness explicitly
  - In Smalltalk, define methods that send `self subclassResponsibility`
Leaf vs. interior

• Rule of thumb: only "leaf" classes should be concrete
• i.e., do not inherit from concrete classes
  • Often later discover that concrete class is not exactly what one wants; but you can't alter it, because the instances depend on behavior
  • Instead, create abstract class and inherit from that
• E.g., do not inherit FancyIconButton directly from IconButton; instead, define AbstractIconButton and inherit both IconButton and FancyIconButton from that.
Factoring exercise: collections

Array          at:, at:put:, first, last
String         at:, at:put:, from:to:, first, last
Set            put:
Bag            put:, count:
Dictionary     at:put:
Interval       from:to:
LinkedList     head, tail, at:, at:put:, first, last
DoublyLinkedList
                head, tail, at:, at:put:, first, last
all collections:
    do:, contains:, any:ifAbsent:, filter:
What is a framework?

• **A**: A library that
  ● Provides functionality for writing applications in a particular **domain**
  ● Is designed to be **extended by the client** (in the OO world, usually by **subclassing** some framework class)
Framework examples

• Graphical user interface (GUI)

• Domain-specific functionality: drawing, widgets (buttons, input fields, etc.), input event loop

• Hook for client extension: user might subclass Button and override mouseDown method, draw method, etc.
Framework examples

• Web application servers

  • Domain-specific functionality: network connections, request parsing, database queries

• Hooks for client extension:
  • defines abstract RequestHandler class, with
  • handleRequest method (default sends empty reply) that is overridden by client.
Framework examples

- Unit testing

- Domain-specific functionality: for sending messages to an object, capturing return values, comparing to expected return value, and recording/presenting results

- Hook for user extension: testCase class with runTest method (default does nothing; user subclasses, and overrides to run tests).