ADT operations and mutation

Creators/Producers

- Creators: return new ADT values (e.g., Java constructors). Effects, not modifies
- Producers: ADT operations that return new values

Mutators: Modify a value of an ADT

Observers: Return information about an ADT

**Mutable** ADTs: creators, observers, and mutators
**Immutable** ADTs: creators, observers, and producers
Three examples

A primitive type as an (immutable) ADT
An immutable type as an ADT
A mutable type as an ADT
Primitive data types are ADTs

**int** is an immutable ADT:
- creators: 0, 1, 2, ...
- producers: + - * / ...
- observer: `Integer.toString(int)`

Peano showed we can define **int** with only one creator
Would this be a good programming language design choice? Why might we want to do this?
A Poly is an immutable polynomial with integer coefficients. A typical Poly is $	ext{c}_0 + \text{c}_1\text{x} + \text{c}_2\text{x}^2 + \ldots$

Overview:
- Always state whether mutable or immutable
- Define an abstract model for use in operation specifications
  - Often difficult and always vital!
- Appeal to math if appropriate
- Give an example (reuse it in operation definitions)
In all ADTs, the state in specifications is abstract, not concrete
(coefficients above refer to specification, not implementation.)
Poly: creators

// effects: makes a new Poly = 0
public Poly()

// effects: makes a new Poly = cx^n
// throws: NegExponent if n < 0
public Poly(int c, int n)

Creators
New object, not part of pre-state: effects, not modifies
Overloading: distinguish procedures of same name by parameters (Example: two Poly constructors)

Footnote: slides omit full JavaDoc comments to save space; style might not be perfect either – focus on main ideas
Poly: observers

// returns: the degree of this,
// i.e., the largest exponent with a
// non-zero coefficient.
// Returns 0 if this = 0.
public int degree()

// returns: the coefficient of the term
// of this whose exponent is d
public int coeff(int d)
Notes on observers

Observers
  Used to obtain information about objects of the type
  Return values of other types
  Never modify the abstract value
  Specification uses the abstraction from the overview
  this
    The particular Poly object being accessed
    The target of the invocation
    Also known as the receiver

Poly x = new Poly(4, 3);
int c = x.coeff(3);
System.out.println(c);  // prints 4
Poly: producers

// returns: this + q (as a Poly)
public Poly add(Poly q)

// returns: the Poly = this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()
Notes on producers

Operations on a type that create other objects of the type
Common in immutable types like `java.lang.String`
  `String substr`ing(int offset, int len)
No side effects
  Cannot change the abstract value of existing objects
IntSet, a mutable datatype: overview and creator

// 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 IntSet()
IntSet: observers

// returns: true if x ∈ this
// else returns false
public boolean contains(int x)

// returns: the cardinality of this
public int size()

// returns: some element of this
// throws: EmptyException when size()==0
public int choose()
IntSet: mutators

// modifies: this
// effects:  this\text{post} = this\text{pre} \cup \{x\}
public void add(int x)

// modifies: this
// effects:  this\text{post} = this\text{pre} - \{x\}
public void remove(int x)
Notes on mutators

Operations that modify an element of the type
Rarely modify anything other than this
    Must list this in modifies clause (if appropriate)
Typically have no return value
    (sometimes return “old” value that was replaced)
Mutable ADTs may have producers too, but that is less common
Quick recap

The examples focused on the abstract specification – with no connection at all to a concrete implementation.

To connect them we need the abstraction function (AF) that maps values of the concrete implementation of the ADT into abstract values in the specification.

The representation invariant (RI) ensures that values in the concrete implementation are well-defined – i.e., the RI must hold for every element in the domain of the AF.
The abstraction function is a function

Why do we map concrete to abstract and not vice versa?

It’s not a function in the other direction.

E.g., lists \([a,b]\) and \([b,a]\) each represent the set \(\{a, b\}\)

It’s not as useful in the other direction.

We can manipulate abstract value through abstract operations
Brief example

Abstract stack with array and “top” index implementation

new() 0 0 0

stack = <>

push(17) 17 0 0

stack = <17>

pop() 17 -9 0

stack = <17>

Abstract states are the same
stack = <17> = <17>

Concrete states are different
< [17, 0, 0], top=1 >
≠
< [17, -9, 0], top=1 >

AF is a function
AF^{-1} is not a function

CSE 331 Autumn 2011
Benevolent side effects

Different implementation of member:

```java
boolean member(Character c1) {
    int i = elts.indexOf(c1);
    if (i == -1)
        return false;
    // move-to-front optimization
    Character c2 = elts.elementAt(0);
    elts.set(0, c1);
    elts.set(i, c2);
    return true;
}
```

Move-to-front speeds up repeated membership tests
Mutates rep, but does not change *abstract* value
  AF maps both reps to the same abstract value
- Creating the concrete object must establish the representation invariant.
- Every concrete operation must maintain the rep invariant.
- Creating the abstraction object must establish the abstraction function.
- Every abstract operation must maintain the AF to provide consistent semantic meaning to the client.
- If things are right, either red arrow above will give the same result.
Writing an abstraction function

The **domain**: all representations that satisfy the rep invariant

The **range**: can be tricky to denote

For mathematical entities like sets: easy

For more complex abstractions: give names to fields or derived values

AF defines the value of each “specification field”

“derived specification fields” more complex

The overview section of the specification should provide a way of writing abstract values

A printed representation is valuable for debugging
ADTs and Java language features

Java classes
- Make operations in the ADT public
- Make other ops and fields of the class private
- Clients can only access ADT operations

Java interfaces
- Clients only see the ADT, not the implementation
- Multiple implementations have no code in common
- Cannot include creators (constructors) or fields

Both classes and interfaces are sometimes appropriate
- Write and rely upon careful specifications
- Prefer interface types instead of specific classes in declarations (e.g., List instead of ArrayList for variables and parameters)
Representation exposure redux

- Hiding the representation of data in the concrete implementation increases the strength of the specification contract, making the rights and responsibilities of both the client and the implementer clearer.
- Defining the fields as `private` in a class is not sufficient to ensure that the representation is hidden.
- *Representation exposure* arises when information about the representation can be determined by the client.
Is Line mutable or immutable?
It depends on the implementation!
   If Line creates an internal copy: immutable
   If Line stores a reference to p1, p2: mutable
Lesson: storing a mutable object in an immutable collection can expose the representation
A half-step backwards

Why focus so much on invariants (properties of code that do not – or are not supposed to – change)?
Why focus so much on immutability (a specific kind of invariant)?

Software is complex – invariants/immutability etc. allow us to reduce the intellectual complexity to some degree. That is, if we can assume some property remains unchanged, we can consider other properties instead. Simplistic to some degree, but reducing what we need to think about in a program can be a huge benefit.