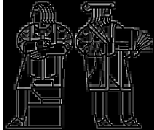
6.170 Lecture 7
Abstract Data Types



John Chapin
MIT EECS
2-14-2000

Outline

1. What is an ADT?
2. How to specify an ADT
   - immutable
   - mutable
3. Things to know about the ADT methodology

Part 1: What is an ADT?

Liskov:

"An abstraction is a many-to-one map. It abstracts from irrelevant details, describing only those details that are relevant to the problem at hand. "

Procedural abstraction:

abstracts from the details of procedures
a specification mechanism

Data abstraction (Abstract Data Type, or ADT):

abstracts from the details of data representation
a specification mechanism
+ a design methodology

Are these two classes the same or different?

```
class Point {
  public real x;
  public real y;
}
```

```
class Point {
  public real r;
  public real theta;
}
```

Different: can't replace one with the other

Same: both classes implement the concept "2-d point"

Goal of ADT methodology:

express the same-ness
clients depend only on the concept "2-d point"

Good because:

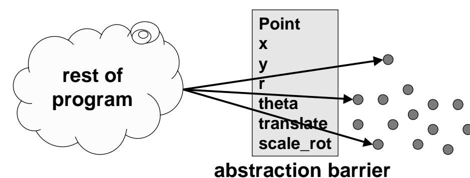
performance optimizations
fix bugs
delay decisions

Concept of 2-d point, as ADT

```
class Point {
  // A 2-d point exists somewhere in the plane, ...
  public real x();
  public real y();
  public real r();
  public real theta();

  // ... can be moved, ...
  public void translate(real delta_x,
                        real delta_y);
  public void scale_rot(real delta_r,
                        real delta_theta);
  // and can be created.
  public Point();        // new point at (0,0)
}
```


Abstract data type = objects + operations



Implementation hidden

No operations on objects of the type except those provided by the abstraction

Part 2: How to specify an ADT

immutable

```
class typename {
  1. overview
  2. creators
  3. observers
  4. producers
}
```

mutable

```
class typename {
  1. overview
  2. creators
  3. observers
  4. mutators
}
```


Poly: overview and creators

```
class Poly {
  // Overview: Polys are immutable polynomials
  // with integer coefficients. A typical Poly
  // is        c0 + c1x + c2x^2 + ...

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

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


Notes on overview and creators

Overview:
- always say if mutable or immutable
- define abstract model for use in specs of ops
  - difficult and vital!
  - appeal to math if appropriate
  - give example (reuse in operation definitions)

Creators:
- new object, not part of prestate: in effects, not modifies
- overloading: distinguish procs of same name by arglist
- Poly(int,int) creator declared to return cx^n
  - key feature of all ADTs: state in op specs is abstract

Poly: observers

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

  public int coeff(int d)
    // effects: returns the coefficient of
    //   the term of this whose exponent is d
```


Notes on observers

Observers:
- used to obtain information about objects of the type
- return values of other types
- never modify the object (since Poly immut, wouldn't anyway)
- reminder: spec uses the abstraction from the overview

this:
- the particular Poly object being worked on
- i.e., the target of the invocation

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


Poly: producers

```
  public Poly add(Poly q)
    // effects: returns the Poly = this + q

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

  public Poly minus()
    // effects: returns the Poly = -this

  } // end Poly
```

Notes on producers

Producers

- operations on a type that create other objects of the type
- common in immutable types, eg. java.lang.String:
  - String substring(int offs, int len)

IntSet: overview and creators

```
class IntSet {
  // Overview: IntSets are mutable, unbounded
  // sets of integers. A typical IntSet is
  //     { x1, ..., xn }.

  public IntSet()
  // effects: makes a new IntSet = {}
```


IntSet: observers

```
public boolean isIn(int x)
  // effects: returns true if x ∈ this
  //          else returns false

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

public int choose() throws EmptyException
  // effects: returns some element of this,
  //    unless size()==0
  //              => throws EmptyException
```


IntSet: mutators

```
public void insert(int x)
  // modifies: this
  // effects:  this_post = this ∪ {x}

public void remove(int x)
  // modifies: this
  // effects:  this_post = this - {x}

} // end IntSet
```


Notes on mutators

This is how we get all nonempty IntSets

Mutators:

- operations that modify an element of the type
- almost never modify anything other than this
- mutable ADTs may have producers too, but less common

Must list this in modifies clause (if appropriate)

Part 3: Things to know

Limitations of the ADT methodology

ADTs and Java language features

Primitive data types are ADTs

Limitations of the ADT methodology (1)

```
class Line {
  // An immutable line in the plane
  public Line(Point p1, Point p2)

  void display()
  // effects: draw the line on the
  //   screen
}
```

Is `display` an observer, mutator or producer?

None of the above!
It is a behavioral method

© John Chapin/John Guttag

Limitations of the ADT methodology (2)

```
Point p1 = new Point();
Point p2 = new Point();
Line line = new Line(p1,p2);
p1.translate(5, 10);  // move point p1
```

Is `Line` mutable or immutable?

Implementation dependent!
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 exposes the representation.

© John Chapin/John Guttag

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	good
clients can/must read implementation	bad

Java interfaces

clients only see the ADT, not the implement.	very good
allow multiple implementations in same program	good
cannot include creators	bad

© John Chapin/John Guttag

Primitive data types are ADTs

int is an immutable ADT:

creators:	`1`, `2`, ...
producers:	`+ - * /` ...
observer:	`Integer.toString(int)`

representation: who knows?

© John Chapin/John Guttag