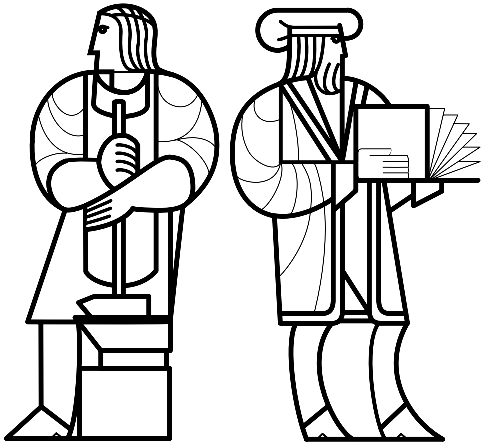
6.170 Lecture 7
Abstract Data Types



MIT EECS

Outline

1. What is an abstract data type (ADT)?
2. How to specify an ADT
   - immutable
   - mutable
3. The ADT methodology

What is an ADT?

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 way of thinking about programs and designs

Why We Need Abstract Data Types

Programming is not usually about

- Inventing and describing algorithms

It is more often about

- Organizing and manipulating data

Leads designers to start by

- Designing data structures
- Writing code to access and manipulate data

Problematical because

- Decisions about data structures made too early
- Duplication of effort in creating derived data
- Very hard to change key data structures

What Is an ADT, revisited

Abstract from organization to meaning of data

Abstract from structure to use

Avoid concern with

right_triangle = struct [base, altitude: float]

vs.

right_triangle = struct [base, hypot, angle: float]

Instead think of type as a set of operations

E.g., create, base, altitude, bottom_angle, ...

Force users to call operations to access data

Are These Classes the Same or Different?

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

Different: can't replace one with the other

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

Goal of ADT methodology

- Express the sameness
- Clients depend only on the concept "2-d point"

Good because:

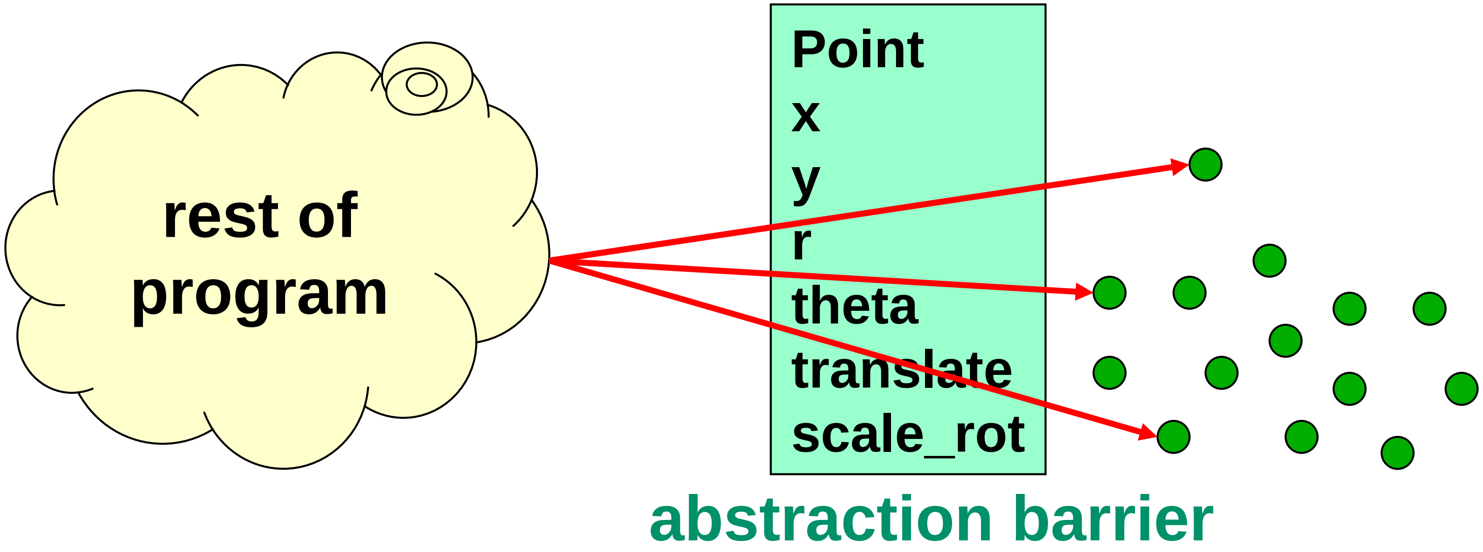
- Performance optimizations
- Fix bugs
- Delay decisions

Concept of 2-d point, as ADT

```
class Point {
  // A 2-d point exists somewhere in the plane, ...
  public float x();
  public float y();
  public float r();
  public float theta();
  // ... can be created, ...
  public Point();        // new point at (0,0)

  // ... can be moved, ...
  public void translate(float delta_x,
                        float delta_y);
  public void scale_rot(float delta_r,
                        float delta_theta);
}
```

Abstract data type = objects + operations



Implementation hidden

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

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
}
```

Primitive Data Types Are ADTs

int is an immutable ADT:

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

Poly: overview and creators

```
class Poly {
  // Overview: Polys are immutable polynomials
  // with integer coefficients. A typical Poly
  // is          $c_0 + c_1x + c_2x^2 + ...$

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

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

Notes on Overview and Creators

Overview

- Always state whether 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
- Example: Poly(int,int) creator declared to return cx^n
  - Key feature of all ADTs, state in specs is abstract

Poly: observers

```
public int degree()
  // 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)
  // 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 abstract value

Specification uses the abstraction from the overview

this

The particular Poly object being worked on

That is, 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)
  // returns: the Poly = this + q

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

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

}
```

Notes on Producers

Producers

Operations on a type that create other objects of the type

Common in immutable types, e.g., *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
  //      { $x_1$, ..., $x_n$ }.

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

IntSet: observers

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

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

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

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 obtain a nonempty IntSet**
- **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)**

Exposing the Rep

```
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 can expose the representation

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
- May make client code over-specific

Java interfaces

- Clients only see the ADT, not the implementation
- Allow multiple implementations in same program
- Cannot include creators (constructors) or fields

My suggestion

- Write and rely upon careful specifications
- Use classes or interfaces as appropriate

Preview: subtyping

A stronger specification can be substituted for a weaker

- Applies to types as well as to individual methods

Java subtypes are not necessarily true subtypes

A Java subtype is indicated via `extends` or `implements`

- Java enforces signatures (types), but not behavior

A true subtype is indicated by a stronger specification

- Also called a “behavioral subtype”
- Every fact that can be proved about supertype objects can also be proved about subtype objects

Subtyping example

```
class A {
        // returns: 0
        int zero(int i) { return 0; }
}

// Java subtype of A, but not true subtype
class B extends A {
        // returns negative of argument
        int zero(int i) { return –i; }    // overriding method
}

// True subtype of A, but not Java subtype
class C {
        // returns: 0
        int zero(int i) { return i – i; }
}
```