Outline

This lecture:
1. What is an Abstract Data Type (ADT)?
2. How to specify an ADT?
3. Design methodology for ADTs

Very related next lectures:
• Representation invariants
• Abstraction functions
Two distinct, complementary ideas for reasoning about ADTs
Procedural and data abstractions

_Procedural_ abstraction:
- Abstract from details of _procedures_ (e.g., methods)
- Specification is the abstraction
  - Abstraction is the specification
- Satisfy the specification with an implementation

_Data_ abstraction:
- Abstract from details of _data representation_
- Also a specification mechanism
  - A way of thinking about programs and design
- Standard terminology: _Abstract Data Type_, or _ADT_
Why we need Data Abstractions (ADTs)

Organizing and manipulating data is pervasive
  – Inventing and describing algorithms is less common

Start your design by designing data structures
  – How will relevant data be organized
  – What operations will be permitted on the data by clients
  – Cf. CSE 332

Potential problems with choosing a data abstraction:
  – Decisions about data structures often made too early
  – Duplication of effort in creating derived data
  – Very hard to change key data structures (modularity!)
An ADT is a set of operations

- ADT abstracts from the organization to meaning of data
- ADT abstracts from structure to use
- Representation should not matter to the client
  - So hide it from the client

Instead, think of a type as a set of operations
  create, getBase, getAltitude, getBottomAngle, ...
Force clients to use operations to access data
Are these classes the same?

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

**Different:** cannot replace one with the other in a program

**Same:** both classes implement the concept “2-d point”

Goal of ADT methodology is to express the sameness:
- Clients depend only on the concept “2-d point”
Benefits of ADTs

If clients “respect” or “are forced to respect” data abstractions…
  – For example, “it’s a 2-D point with these operations…”

• Can delay decisions on how ADT is implemented
• Can fix bugs by changing how ADT is implemented
• Can change algorithms
  – For performance
  – In general or in specialized situations
• ...

We talk about an “abstraction barrier”
  – A good thing to have and not cross (also known as violate)
class Point {
    // A 2-d point exists 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)
    public Point centroid(Set<Point> points);

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

Observers
Creators/Producers
Mutators
Abstract data type = objects + operations

- Implementation is hidden
- The only operations on objects of the type are those provided by the abstraction
Specifying a data abstraction

• A collection of procedural abstractions
  – Not a collection of procedures

• An abstract state
  – Not the (concrete) representation in terms of fields, objects, …
  – “Does not exist” but used to specify the operations
  – Concrete state, not part of the specification, implements the abstract state
    • More in upcoming lecture

• Each operation described in terms of “creating”, “observing”, “producing”, or “mutating”
  – No operations other than those in the specification
## Specifying an ADT

<table>
<thead>
<tr>
<th>Immutable</th>
<th>Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. overview</td>
<td>1. overview</td>
</tr>
<tr>
<td>2. abstract state</td>
<td>2. abstract state</td>
</tr>
<tr>
<td>3. creators</td>
<td>3. creators</td>
</tr>
<tr>
<td>4. observers</td>
<td>4. observers</td>
</tr>
<tr>
<td>5. producers</td>
<td>5. producers (rare)</td>
</tr>
<tr>
<td>6. mutators</td>
<td>6. mutators</td>
</tr>
</tbody>
</table>

- **Creators**: return new ADT values (e.g., Java constructors)
- **Producers**: ADT operations that return new values
- **Mutators**: Modify a value of an ADT
- **Observers**: Return information about an ADT
Implementing an ADT

To implement a data abstraction (e.g., with a Java class):
  – See next two lectures
  – This lecture is just about specifying an ADT
  – *Nothing* about the concrete representation appears in the specification
Poly, an immutable datatype: overview

/**
 * A Poly is an immutable polynomial with integer coefficients. A typical Poly is
 * \[ c_0 + c_1 x + c_2 x^2 + \ldots \]
 **/

class Poly {

Overview:

– State whether mutable or immutable
– Define an abstract model for use in operation specifications
  • Difficult and vital!
  • Appeal to math if appropriate
  • Give an example (reuse it in operation definitions)
– State in specifications is *abstract*, not concrete
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: in 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
// throws: NegExponent if d < 0
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
– Target of the invocation
– Also known as the receiver

```java
Poly x = new Poly(4, 3);
int c = x.coeff(3);
System.out.println(c);  // prints 4
```
// returns: this + q (as a Poly)
public Poly add(Poly q)

// returns: the Poly equal to 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 substring(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 and only if x ∈ this
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: \( \text{this}_{\text{post}} = \text{this}_{\text{pre}} \cup \{x\} \)
public void add(int x)

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

• Operations that modify an element of the type

• Rarely modify anything (available to clients) other than this
  – List this in modifies clause (if appropriate)

• Typically have no return value
  – “Do one thing and do it well”
  – (Sometimes return “old” value that was replaced)

• Mutable ADTs may have producers too, but that is less common