Data abstraction: Abstract Data Types (ADTs)

CSE 331
University of Washington

Michael Ernst

Outline

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

This lecture: ADT *specifications*

Next lectures: ADT *implementations*

Representation invariants (RIs):

Relationship among implementation fields

Abstraction functions (AFs)

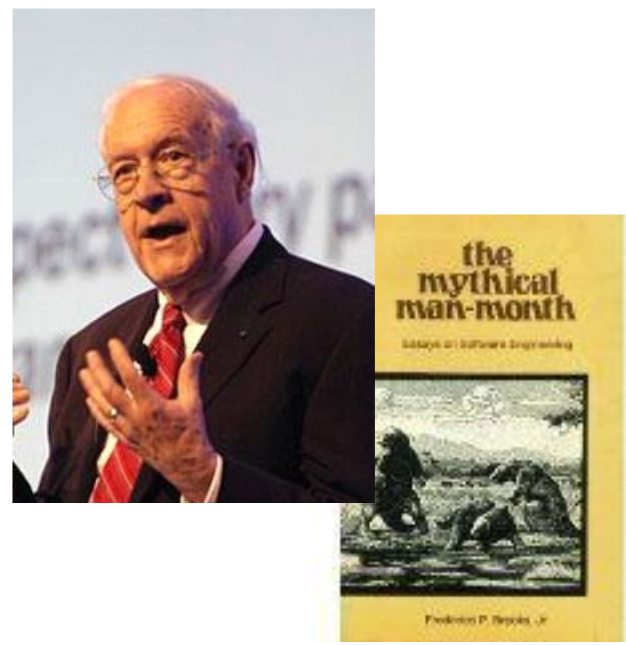
Relationship between ADT specification and implementation

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- Linus Torvalds

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

-- Fred Brooks



Procedural and data abstraction

Recall procedural abstraction:

- Abstracts from details of procedure *implementations*
- A specification mechanism
- Satisfy the specification with an implementation

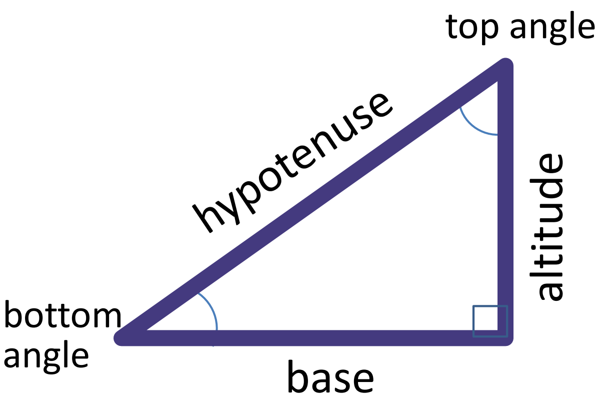
Data abstraction:

- Abstracts from the details of *data representation*
- A specification mechanism
  + a way of thinking about programs and designs
- Standard terminology: Abstract Data Type, or ADT

Why we need data abstraction

- Organizing and manipulating data is pervasive
  - Inventing and describing algorithms is rare
- Start your design by designing data structures
  - What operations are permitted by clients
  - Secondary:
    - How data is organized/represented/stored
    - What algorithms manipulate the data
- It is challenging to design a data structure:
  - Decisions about data structures are 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



A right triangle

ADT indicates the meaning of data and how it is used

ADT abstracts away the organization/structure of data

A type is a set of operations

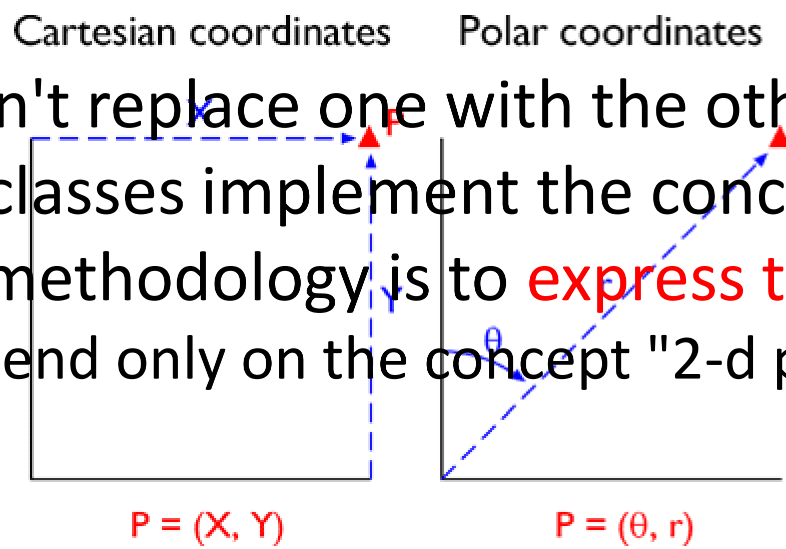
`create`, **`getBase`**, **`getAltitude`**, **`getBottomAngle`**, …

Operations are the only way clients can access data

Are these classes the same or different?

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

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



Different: can't replace one with the other

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"

Concept of 2-d point, as an ADT

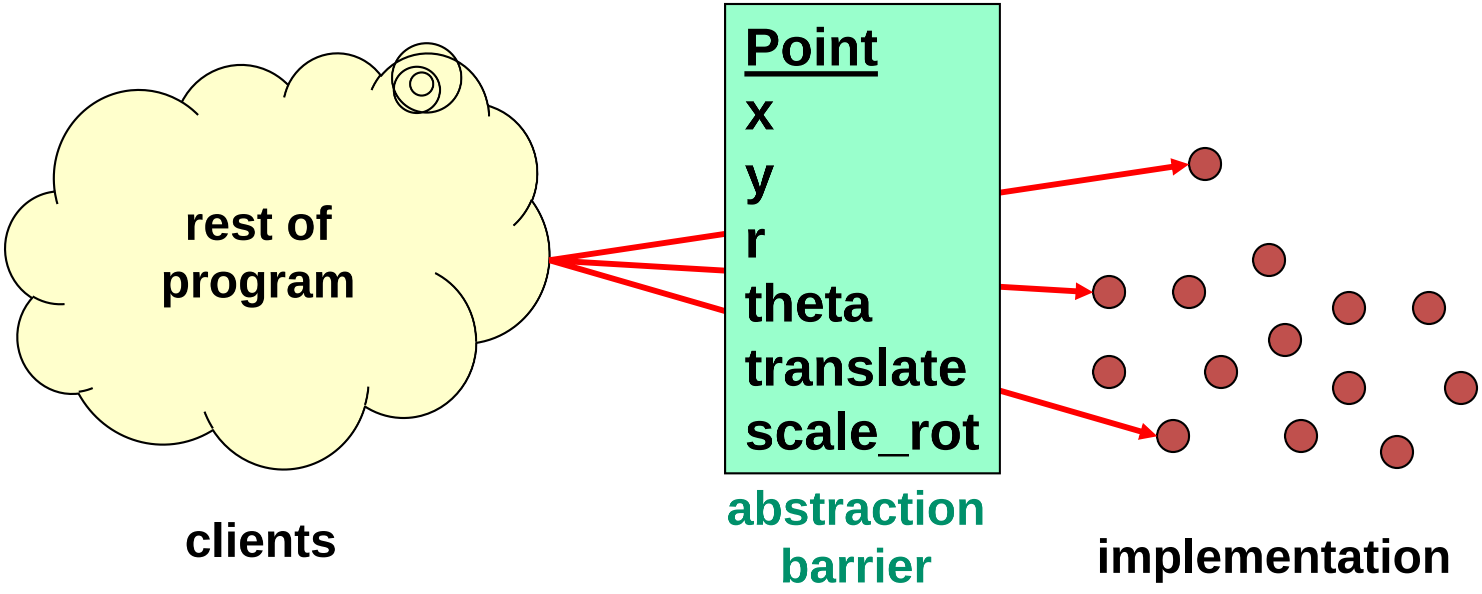
```
class Point {
  // A 2-d point exists somewhere in the plane, ...
  public float x();
  public float y();
  public float r();
  public float theta();

  // ... can be created, ...
  public Point();          // new point at (0,0)
  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 (x, y, r, theta)

Creators/ Producers (Point, centroid)

Mutators (translate, scaleAndRotate)

Abstract data type = objects + operations



The implementation is hidden

The only operations on objects of the type are those provided by the abstraction

Specifying a data abstraction

- An *abstract state*
  - *Not* the (concrete) representation in terms of fields, objects, …
    - Parts of the abstract and concrete state might coincide
  - Used to specify the operations
- A collection of *operations* (procedural abstractions)
  - *Not* a collection of procedure implementations
  - Specified in terms of abstract state
  - No other way to interact with the data abstraction
  - 4 types of operations: creators, observers, producers, mutators
- Says nothing about the concrete representation

How to specify an ADT

immutable	mutable
`class` ***TypeName*** `{`	`class` ***TypeName*** `{`
`1. overview`	`1. overview`
Documentation	
`2. abstract fields`	`2. abstract fields`
Abstract fields (a.k.a. specification fields): next lecture	
`3. creators`	`3. creators`
Return new ADT value (e.g., Java constructor)	
`4. observers`	`4. observers`
Return information about the abstract value	
`5. producers`	`5. producers` (rare)
Return new ADT value, from an existing value	
~~`6. mutators`~~	`6. mutators`
Modify an ADT's abstract value	
`}`	`}`

A primitive data types is an ADT

`int` is an immutable ADT:

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

Another definition of `int`:

(Known as "Peano arithmetic")

- creators: `0`
- producers: `successor`, `predecessor`
- observer: `Integer.toString(int)`

Why would we want to do that?

Poly, an immutable datatype: overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients.  A typical Poly is
 *          c0 + c1x + c2x^2 + ...
 **/
class Poly {
```

Abstract state (specification fields)

Overview:

- Always state whether mutable or immutable
- Define abstract model for use in specs of operations
  - Difficult and vital!
  - Appeal to math if appropriate
  - Give an example (reuse it in operation definitions)

In all ADTs, state in specs is *abstract*, not concrete

- Refers to specification fields, not implementation fields

Poly: creators

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

// effects: makes a new Poly = cx^n
// throws: NegExponent when 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

(Slides use terse comments for brevity; 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 equal to this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()
```

Producers

- Operations on a type that create other objects of the type
- Common in immutable types, e.g., `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
//      { x1, ..., xn }.
class IntSet {

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

IntSet: observers

```
// returns: true iff x ∈ this
public boolean contains(int x)
```

Or:
returns x ∈ this
Or
returns true if x ∈ this
else returns false

```
// 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_post = this_pre ∪ {x}
public void add(int x)         // insert an element

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

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
- Mutable ADTs may have producers too (uncommon)

Representation exposure

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

Does that change **`line`**?

Lesson: storing a mutable object in an immutable collection may expose the representation

- A client can determine information about the rep
- A client can directly change the rep

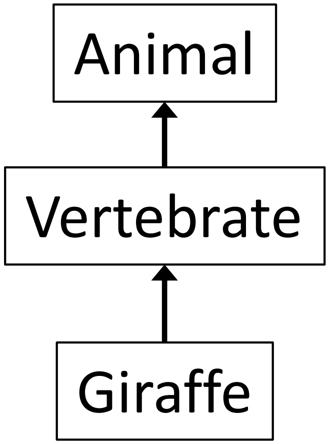
ADTs and Java language features

- Java classes – how to use them
  - 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

Subtyping and substitutability

A stronger specification can be substituted for a weaker one.

Applies to types as well as to individual methods




```
class Vertebrate extends Animal {
    // number of bones in neck; result > 0
    int neckBones() { … }
}
```

Client code:

```
    Giraffe g = new Giraffe();
    Animal a = g;
    g.neckBones();   // OK
    a.neckBones();   // compile-time error!
```

Which can be used as a subtype?

```
class Vertebrate extends Animal {
  // returns > 0
  abstract int neckBones();
}


class Squid extends Vertebrate {
  @Override
  int neckBones() { return 0; }
}


class Human {
  int neckBones() { return 7; }
}
```

A possible use:

```
  // returns average length of vertebrae in neck
  int vertebraLength(Vertebrate v) {
    return v.neckLength()/v.neckBones();
  }
```

Java subtypes vs. 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

Don't write a Java subtype that is not a true subtype

- Causes unexpected, confusing, incorrect behavior