CSE 331
Software Design and Implementation

Lecture 5
Abstract Data Types

Leah Perlmutter / Summer 2018

Platonic Forms

- Quote

Announcements

Announcements

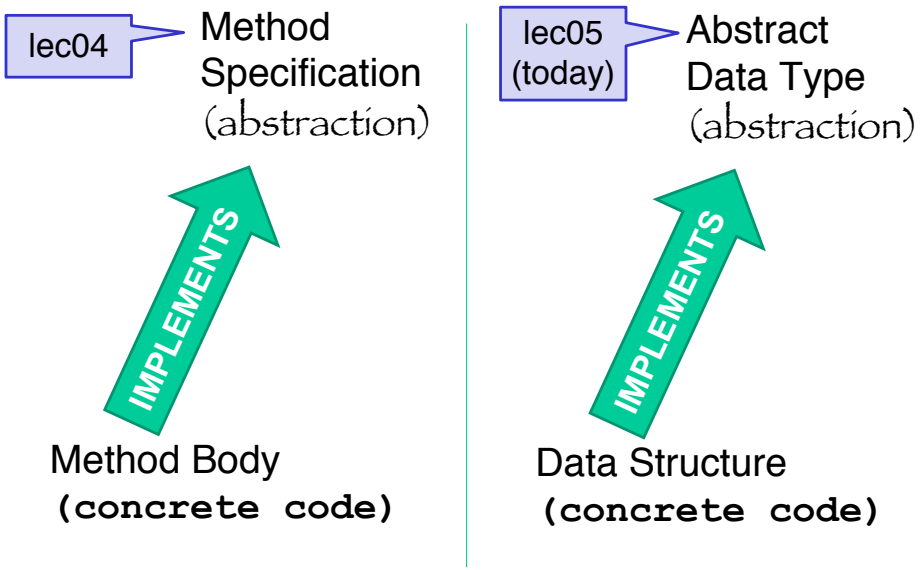
- Section tomorrow!
  - Loop reasoning
    - useful for HW2
    - historically one of the most challenging concepts in 331
  - Development environment setup
    - please install Eclipse and bring your laptop
    - Eclipse installation instructions on the Course Website

Announcements

- Please attend the section that you are formally enrolled in
  - Makes it possible to earn your section participation grade
  - Makes your TAs' lives much easier!
- HW0 feedback published on gradescope
- Reading assignment 2 posted, Quiz 2 coming soon!
  - Due tomorrow: Thursday 6/28 at 10 pm
- HW2 is out! Due Monday 7/2 at 10 pm
  - Topic is loop reasoning – harder than HW1 so start early

What is an ADT?

Procedural and data abstractions



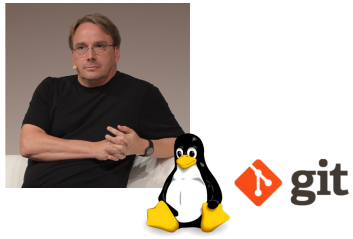
Procedural and data abstractions

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

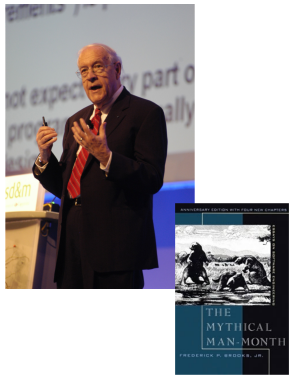
Good programmers worry about data structures and their relationships.

-- 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



The need for data abstractions (ADTs)

Organizing and manipulating data is pervasive
- See also: CSE 332 – Data Structures & Parallelism

Start your design by designing data abstractions
- What is the meaning of the data?
- What operations will be permitted on the data by clients?

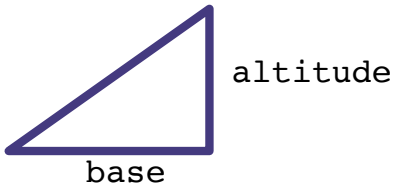
Later, you can choose a data structure
- This means writing the implementation
- Decisions about data structures often made too early
- Very hard to change key data structures (modularity!)

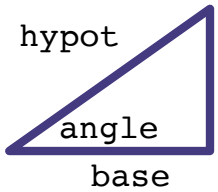
An ADT is a set of operations

- ADT abstracts from the *organization* to *meaning* of data
- ADT abstracts from *structure* to *use*
- Here are two bad examples of how to implement a triangle class
  - Why are they bad?

```
class BadRightTriangle1 {
  float base, altitude;
}
```

```
class BadRightTriangle2 {
  float base, hypot, angle;
}
```





An ADT is a set of operations

- ADT abstracts from the *organization* to *meaning* of data
- ADT abstracts from *structure* to *use*
- Here are two bad examples of how to implement a triangle class
  - Why are they bad?

```
class BadRightTriangle1 {
  float base, altitude;
}
```

```
class BadRightTriangle2 {
  float base, hypot, angle;
}
```

Instead, we should think of a type as a set of operations

`create, getBase, getAltitude, getBottomAngle, …`

Force clients to use operations to access data

An ADT is a set of operations

```
class BadRightTriangle1 {
  float base, altitude;
}
```

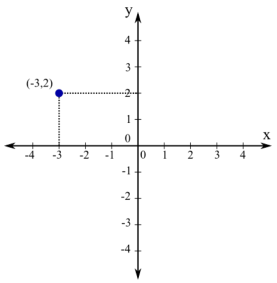
```
class BadRightTriangle2 {
  float base, hypot, angle;
}
```

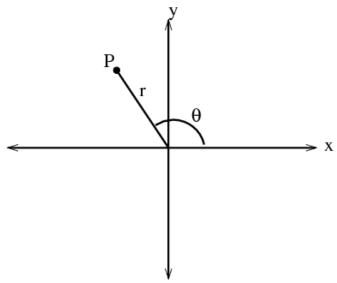
```
class RightTriangle {
  // fields don't matter to client!
  // Not part of ADT
  private float ...;

  // Operations are the important stuff.
  // Same ops, regardless of which fields we use
  public RightTriangle create();
  public float getBase();
  public float getAltitude();
  public float getBottomAngle();
  ...
}
```

Are these classes the same?

```
class BadPoint1 {
  public float x;
  public float y;
}
```

```
class BadPoint2 {
  public float r;
  public float theta;
}
```





Are these classes the same?

```
class BadPoint1 {
  public float x;
  public float y;
}
```

```
class BadPoint2 {
  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"

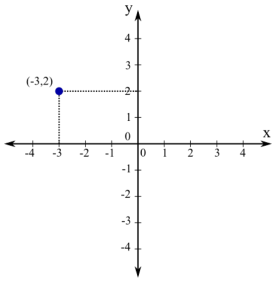
Goal of ADT methodology is to express the sameness:

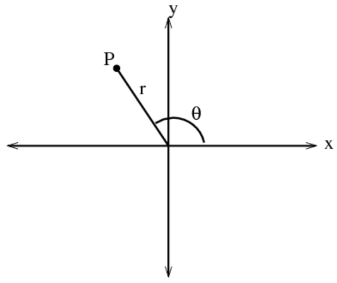
- Analogy with Platonic Forms
- Clients depend only on the concept "2-d point"

Are these classes the same?

```
class Point1 {
  private float x;
  private float y;
  // public ops..
}
```

```
class Point2 {
  private float r;
  private float theta;
  // public ops..
}
```





Concept of 2-d point, as an ADT

Informal notation warning

```
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

Benefits of ADTs

Suppose clients respect our data abstractions…

- For example, "it's a 2-D point with these operations…"

Then, as the implementer, we can do these good things:

- 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

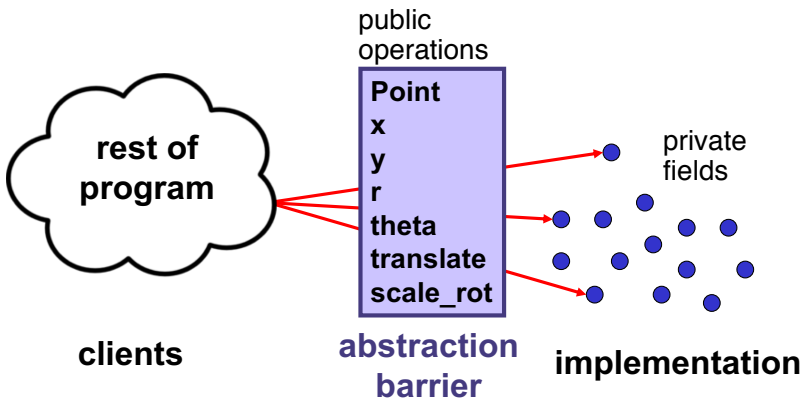
Debuggable

Flexible

We talk about an "*abstraction barrier*"

- A good thing to have and not *cross*

Abstract data type = objects + operations



- Implementation is hidden
- Only operations on objects of the type are provided by abstraction

Specifying an ADT

Specifying a data abstraction

An abstract state
- Not the (concrete) representation in terms of fields, objects, …
- "Does not exist" but used to specify the operations
- Excludes concrete state that implements the abstract state (more in upcoming lecture)

Abstract vs. Concrete State Example

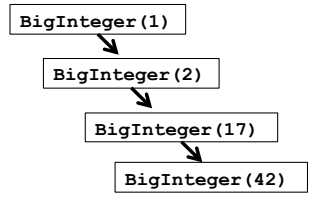
Abstract State of an int list:
Ordered sequence of integer values

1, 2, 17, 42

One Abstract State to rule them all!

generalize across

Possible Concrete State of an int list:
Linked list of BigInteger



Possible Concrete State of an int list:
Array of primitive ints

```
[1, 2, 17, 42]
```

Many more possible Concrete States...!

Specifying a data abstraction

An abstract state

e.g. the fact that an int list is a sequence of integer values

- Not the (concrete) representation in terms of fields, objects, …
- "Does not exist" but used to specify the operations
- Excludes concrete state that implements the abstract state (more in upcoming lecture)

A collection of procedural abstractions

e.g. a well specified set of list operations on an int list

- aka operations; aka method specs
- Excludes code
- Each operation described in terms of "creating", "observing", "producing", or "mutating"
  - No operations other than those in the specification

Specifying an ADT

Immutable

1. overview
2. abstract state
3. creators
4. observers
5. producers
6. ~~mutators~~

Mutable

1. overview
2. abstract state
3. creators
4. observers
5. producers (rare)
6. mutators

- 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 spec

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)

1. Overview:
   - English description, states whether mutable or immutable
2. Define Abstract State for use in operation specifications
   - Difficult and vital!
   - Appeal to math if appropriate
   - Give an example (reuse it in operation definitions)
   - Excludes concrete state

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)
```

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

Informal Notation Warning: 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

4. 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*

```
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()
```

Notes on producers

5. 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 { x1, ..., xn }.
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:  this_post = this_pre ∪ {x}
public void add(int x)

// modifies: this
// effects:  this_post = this_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

Mutable/Immutable ADTs (revisited)

Immutable

1. overview
2. abstract state
3. creators
4. observers
5. producers
6. ~~mutators~~

Mutable

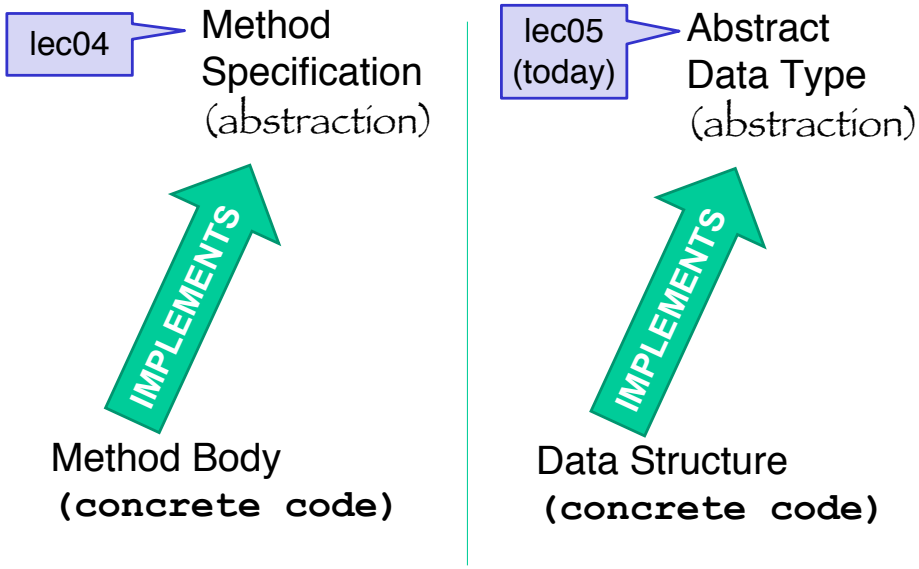
1. overview
2. abstract state
3. creators
4. observers
5. producers (rare)
6. mutators

- 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

Why immutable?

- If you are curious, read Effective Java!
  - Minimize Mutability (EJ2: 39; EJ3: 50)

Procedural and data abstractions



Coming up…

Very related next lectures:

- Representation invariants
- Abstraction functions

Distinct, complementary ideas for ADT reasoning

Closing

Closing

- Section tomorrow!
  - install eclipse and bring laptop
- Quiz 2 due Thursday
- HW2 due Monday