

## Specifications, conclusion. Abstract Data Types (ADTs)

Based on notes by Michael Ernst,  
University of Washington

## Announcements

- Quiz 1 and 2 graded
  - Grades in LMS
- Labs 1 and 2 graded
  - Feedback in Homework server
- Currently grading HW0 and HW1
  - Feedback in Homework server
  - Grades in Homework server and the LMS

2

## Announcements

- HW2 due on Friday, Sep. 25<sup>th</sup>
  - Commit to SVN then
  - **Submit through Homework Server!**
- HW3 out on Friday

Fall 15 CSCI 2600, A Milanova

3

## So Far

- Specifications
  - Benefits of specifications
  - Specification conventions
    - Javadocs, PoS specifications, JML/Dafny
  - Specification style
  - Specification strength
  - Comparing specifications

4

## Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- Specifying an ADT
  - Immutable
  - Mutable
- The ADT design methodology
- Next: reasoning about ADT implementations

5

## Specification Strength

- “A is stronger than B” (or  $A \Rightarrow B$ ) means
  - For every implementation *I*
    - “*I* satisfies A” implies “*I* satisfies B”
    - The opposite is not necessarily true!
    - Larger world of *I*s satisfies the weaker spec
  - For every client *C*
    - “*C* works with B” implies “*C* works with A”
    - The opposite is not necessarily true!
    - Larger world of clients works with stronger spec
- A stronger spec is **harder to implement**
- A stronger spec is **easier to use**

6

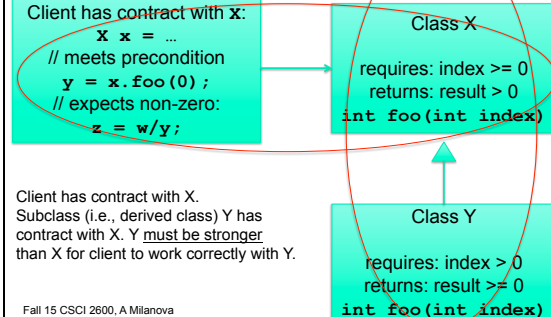
## Why Care About Specification Strength?

- Because of **substitutability**!
- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - I.e., an implementation that satisfies a **stronger** specification, can be used in a client that expects a **weaker** specification

Fall 15 CSCI 2600, A Milanova

7

## Substitutability



Fall 15 CSCI 2600, A Milanova

## Substitutability

- Substitutability ensures correct hierarchies
- Client code: `x x; ... x.foo(index);`
  - Client is "polymorphic": written against `x`, but is expected to work with any subclass of `x`
  - A subclass of `x`, say `y`, may have its own implementation of `foo`, `y.foo(int)`. Client must work correctly with `y.foo(int)` too!
- If the spec of `y.foo(int)` is stronger than the spec of `x.foo(int)` then we can safely substitute `y.foo(int)` for `x.foo(int)`!

9

## Comparison by Logical Formulas

- The following is a sufficient condition:  
If  $P_B \Rightarrow P_A$  and  $Q_A \Rightarrow Q_B$  then A is stronger than B

$$P_B \Rightarrow P_A \text{ and } Q_A \Rightarrow Q_B$$

A is stronger than B

- Too strong a requirement!

Fall 15 CSCI 2600, A Milanova

10

## Simple Example

- Spec A: **requires:** `0 <= arg`  
**returns:** `1 <= result <= 10`
- Spec B: **requires:** `-1 <= arg`  
**returns:** `2 <= result <= 5`
- Spec C: **requires:** `true` // the weakest condition  
**returns:** `2 <= result <= 5`
- Spec D: **requires:** `1 <= arg <= 10`  
**returns:** `1 <= result <= 20`

Fall 15 CSCI 2600, A Milanova

11

## Example: `int find(int[] a, int val)`

- Specification B:  
**requires:** `a` is non-null and `val` occurs in `a` [ $P_B$ ]  
**returns:** `i` such that `a[i] = val` [ $Q_B$ ]
- Specification A:  
**requires:** `a` is non-null [ $P_A$ ]  
**returns:** `i` such that `a[i] = val` if `val` occurs in `a` and `-1` if `val` is not in `a` [ $Q_A$ ]

Clearly,  $P_B \Rightarrow P_A$ .  
But  $Q_A$ , which states  
"`val` occurs in `a` => returns `i` such that `a[i]=val` AND  
`val` does not occur in `a` => returns `-1`"  
does not imply  $Q_B$ !

12

## Comparing by Logical Formulas

- **(I satisfies specification A)** is a logical formula:  $P_A \Rightarrow Q_A$   
( $P_A$  is the precondition of A,  $Q_A$  is the postcondition of A)
- Spec A is **stronger** than spec B if and only if  
for each implementation I, **(I satisfies A)  $\Rightarrow$  (I satisfies B)**  
which is equivalent to  **$A \Rightarrow B$**
- A is stronger than B iff  $(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B)$

Recall from FoCS and/or Intro to Logic:  $p \Rightarrow q \equiv !p \vee q$

Fall 15 CSCI 2600, A Milanova

13

## Comparing by Logical Formulas

$(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) =$   
 $!(P_A \Rightarrow Q_A) \vee (P_B \Rightarrow Q_B) =$  [ due to law  $p \Rightarrow q = !p \vee q$  ]  
 $!(P_A \vee Q_A) \vee (!P_B \vee Q_B) =$  [ due to  $p \Rightarrow q = !p \vee q$  ]  
 $(P_A \wedge !Q_A) \vee (!P_B \vee Q_B) =$  [ due to  $!(p \vee q) = !p \wedge !q$  ]  
 $(!P_B \vee Q_B) \vee (P_A \wedge !Q_A) =$  [ due to commutativity of  $\vee$  ]  
 $(!P_B \vee Q_B \vee P_A) \wedge (!P_B \vee Q_B \vee !Q_A)$  [ distributivity ]  
 **$[ P_B \Rightarrow (Q_B \vee P_A) ] \wedge [ (P_B \wedge Q_A) \Rightarrow Q_B ]$**   
 A is stronger than B if and only if  
 $P_B \Rightarrow Q_B$  is true trivially or  $P_B$  implies  $P_A$  AND  
 $Q_A$  together with  $P_B$  imply  $Q_B$  (i.e., for the inputs permitted  
 by  $P_B$ ,  $Q_B$  holds)

Fall 15 CSCI 2600, A Milanova

14

## Example: `int find(int[] a, int val)`

- Specification B:  
requires: a is non-null and val occurs in a [ $P_B$ ]  
returns: i such that  $a[i] = \text{val}$  [ $Q_B$ ]
- Specification A:  
requires: a is non-null [ $P_A$ ]  
returns: i such that  $a[i] = \text{val}$  if val occurs in a and  
-1 if val does not occur in a [ $Q_A$ ]

$P_B \Rightarrow P_A$  ( $P_B$  includes  $P_A$  and one more condition)

Now, let's show  $P_B \wedge Q_A \Rightarrow Q_B$ .

$P_B$  implies "val occurs in a".  $Q_A$  states

"val occurs in a  $\Rightarrow$  returns i s.t.  $a[i] = \text{val}$ ".

$P_B \wedge Q_A \Rightarrow$  "returns i s.t.  $a[i] = \text{val}$ ", precisely  $Q_B$ !

15

## Example: `int find(int[] a, int val)`

- Specification B:  
requires: a is non-null and val occurs in a [ $P_B$ ]  
returns: i such that  $a[i] = \text{val}$  [ $Q_B$ ]
- Specification A:  
requires: a is non-null [ $P_A$ ]  
returns: i such that  $a[i] = \text{val}$  if val occurs in a and  
-1 if val does not occur in a [ $Q_A$ ]

Intuition:  $Q_A$ , by itself, does not imply  $Q_B$  because A may return -1. But  $P_B$  and  $Q_A$  imply  $Q_B$ . Thus, it's still ok to substitute A for B. The client, written against B, meets the precondition  $P_B$ .  $P_B$  ensures the "right branch" of  $Q_A$  which subsumes  $Q_B$ .

16

## Converting PoS Specs into Logical Formulas

- PoS specification has 5 clauses  
Precondition:  
requires: ...  
Postcondition:  
modifies: ..., effects: ..., returns: ..., throws: ...
- To reason about PoS specs, we must first convert specs into  $P \Rightarrow Q$  logical formulas
  - Step 1: absorbs returns and throws into effects
  - Step 2: converts into logical formula

Fall 15 CSCI 2600, A Milanova

17

## Converting PoS Specs into Logical Formulas

- PoS specification  
requires: R  
modifies: M  
effects: E  
is equivalent to this logical formula  
 $R \Rightarrow (E \wedge (\text{nothing but M is modified}))$
- throws and returns are absorbed into effects E

Fall 15 CSCI 2600, A Milanova (slide modified from slides by Michael Ernst)

18

## Convert Spec to Formula, step 1: absorb throws and returns into effects

### ■ PoS specification convention

**requires:** (unchanged)  
**modifies:** (unchanged)  
**effects:** } absorbed into effects  
**returns:** }  
**throws:** }

Fall 15 CSCI 2600, A Milanova (slide modified from slides by Michael Ernst)

19

## Convert Spec to Formula, step 1: absorb throws and returns into effects

### ■ set from java.util.ArrayList<T>

**T set(int index, T element)**

**requires:** true  
**modifies:** this[index]  
**effects:** this<sub>post</sub>[index] = element  
**throws:** IndexOutOfBoundsException if index < 0 || index ≥ size  
**returns:** this<sub>pre</sub>[index]

Absorb **effects**, **returns** and **throws** into new effects:

if index < 0 || index ≥ this.size then  
   throw IndexOutOfBoundsException  
 else this<sub>post</sub>[index] = element and returns this<sub>pre</sub>[index]

Fall 15 CSCI 2600, A Milanova (slide modified from slides by Michael Ernst)

20

## Convert Spec to Formula, step 2: Convert into Formula

### ■ set from java.util.ArrayList<T>

**T set(int index, T element)**

**requires:** true  
**modifies:** this[index]  
**effects:** if index < 0 || index ≥ this.size then  
   throws IndexOutOfBoundsException  
   else  
     this<sub>post</sub>[index] = element and returns this<sub>pre</sub>[index]

Denote effects expression by E. Resulting formula is:

true => (E ∧ (foreach i ≠ index, this<sub>post</sub>[i] = this<sub>pre</sub>[i]))

Fall 15 CSCI 2600, A Milanova (slide modified from slides by Michael Ernst)

21

## Exercise

### ■ Convert PoS spec into logical formula

**public static int binarySearch(int[] a, int key)**

**requires:** a is sorted in ascending order  
**modifies:** none  
**effects:** none  
**returns:** i such that a[i] = key if such an i exists; -1 otherwise

Fall 15 CSCI 2600, A Milanova

22

## Exercise

### ■ Convert spec into logical formula

**static void listAdd2(List<Integer> lst1, List<Integer> lst2)**

**requires:** lst1, lst2 are non-null.  
           lst1 and lst2 are same size.  
**modifies:** lst1  
**effects:** i-th element of lst1 is replaced with the sum of  
           i-th elements of lst1 and lst2  
**returns:** none

Fall 15 CSCI 2600, A Milanova

23

## Exercise

### ■ Convert spec into logical formula

**private static void swap(int[] a, int i, int j)**

**requires:** a non-null, 0 ≤ i, j < a.length  
**modifies:** a[i] and a[j]  
**effects:** a<sub>post</sub>[i] = a<sub>pre</sub>[j] and a<sub>post</sub>[j] = a<sub>pre</sub>[i]  
**returns:** none

```
static void swap(int[] a, int i, int j) {
    int tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```

Fall 15 CSCI 2600, A Milanova

24

## Specifications, Review

- Benefits
- Conventions
  - Javadocs, PoS specifications, JML
- Style
- Strength
- Comparing specifications
  - By hand, by logical formulas
- Converting PoS specs into logical formulas

Fall 15 CSCI 2600, A. Milanova

25

## Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- Specifying an ADT
  - Immutable
  - Mutable
- The ADT design methodology
- Next: reasoning about ADT implementations

26

## Abstraction

- Abstraction: hiding unnecessary low-level details
- Control abstraction (procedural abstraction)
  - A procedure (method) abstracts the details of an algorithm
  - One part of abstraction: signature, provides name, parameter types, return type. Not enough!
  - Another part: specification, provides detail about behavior and effects
  - E.g., `int binarySearch(int[] a, int key)`
  - Reasoning about code connects implementation to specification

Fall 15 CSCI 2600, A. Milanova

27

## Abstraction

- Data abstraction
  - Types: abstract away from the details of data representation
  - E.g., type `integer` is an abstraction
  - E.g., C type `struct Person` is an abstraction
- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is operations + object
  - A specification mechanism
  - A way of thinking about programs and design

Fall 15 CSCI 2600, A. Milanova (slide based on slide by Michael Ernst)

28

## Abstract Data Types are Important

- ADTs are about organizing and manipulating data
- Organizing and manipulating data is pervasive. Inventing and describing algorithms is not
- Start your design by **designing red data structures**. Write code to access and manipulate data
- Chose data structures carefully!

Fall 15 CSCI 2600, A. Milanova (based on slide by Michael Ernst)

29

## ADT is a way of thinking about programs and design

- From **domain concept**
  - E.g., the math concept of the polynomial, the integer set, the concept of a library item, etc.
- through **ADT**
  - Describes domain concept in terms **specification fields** and **abstract operations**
- to **implementation**
  - Implements ADT with representation fields and concrete operations

Fall 15 CSCI 2600, A. Milanova

30

## Example: Polynomial with Int Coefficients, Domain Concept & ADT

ADT:

Overview description:

A Poly is an **immutable** polynomial with int coefficients. A Poly is:

•  $c_0 + c_1x + c_2x^2 + \dots$

Specification fields (Abstract fields).

Set of abstract operations:

add, mul, eval, etc. with PoS style specs referencing **abstract** specification fields

Fall 15 CSCI 2600, A Milanova

31

## Example: Polynomial with Int Coefficients, Implementation

```
class Poly {
    // rep. invariant: d = coeffs.length-1
    private int d; // degree of the polynomial
    private int[] coeffs; // coefficients
    // concrete operations add, sub, mul,
    // eval, in terms of rep. fields coeffs, d.
}
```

```
class Poly {
    // rep. invariant: ...
    private List<Term> terms; // terms of poly
    // operations add, sub, mul, eval, etc. in
    // terms of rep. field terms.
}
```

Fall 15 CSCI 2600, A Milanova

32

## Another Example: A Meeting, Domain Concept & ADT

ADT:

Overview description:

An appointment for a meeting.

date : Date // the time

room : integer // room number

with : Set<Person> // appt with

Specification fields.

Set of abstract operations:

e.g., addAttendee, etc. with PoS style specs referencing **abstract** specification fields

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)

33

## Why ADTs?

- Bridges gap between **domain concept** and **implementation**
- Formalizes domain concept, provides basis for reasoning about correctness of the implementation
- Shields client from implementation. Implementation can vary without affecting client!

Fall 15 CSCI 2600, A Milanova

34

## An ADT Is a Set of Operations

- Operations operate on data
- ADT abstracts from **organization** to **meaning** of data
- ADT abstracts from **structure** to **use**
- Data representation (implementation) doesn't matter!

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

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

- Instead, think of a type as a **set of operations**: create, getBase, getAltitude, getBottomAngle, etc.
- Force clients to call operations to access data

Fall 15 CSCI 2600, A Milanova (modified from a slide by Michael Ernst)

35

## Are These Types Same or Different?

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

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

- They are **different**!
- They are the **same**! Both implement the concept of the 2-d point. Goal of ADT methodology is to **express sameness**
  - Clients depend only on the set of operations: x(), y(), r(), theta(), etc.
  - Data representation can be changed: to change algorithms, to delay decisions, to fix bugs

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

36

## Are These Types Same or Different?

```
class Poly {
    private int d;
    private int[] coeffs;
}
```

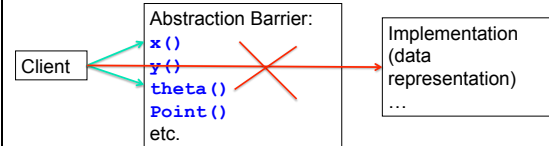
```
class Poly {
    private List<Term> terms;
}
```

- Clients depend only on the set of operations: `add(Poly)`, `mul(Poly)`, etc.

Fall 15 CSCI 2600, A. Milanova

37

## Abstraction Barrier



Clients access the ADT through its operations. They never access the data representation.

Fall 15 CSCI 2600, A. Milanova

38

## 2-d Point as an ADT

```
class Point {
    // A 2-d point in the plane
    public float x();
    public float y();
    public float r();
    public float theta();
    // ... can be created
    public Point(); (0,0)
    public Point(float x, float y);
    public Point centroid(Set<Point> points);
}
```

Observers

Creators/  
Producers

Fall 15 CSCI 2600, A. Milanova (slide due to Michael Ernst)

39

## 2-d Point as an ADT

```
// class Point continued
...

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

Mutators

Fall 15 CSCI 2600, A. Milanova (slide due to Michael Ernst)

40

## Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- **Specifying an ADT**
  - immutable
  - mutable
- The ADT design methodology
- Next: reasoning about ADT implementations

41

## Specifying an ADT

immutable	mutable
<b>class TypeName</b>	<b>class TypeName</b>
1. overview	1. overview
2. specification fields	2. specification fields
3. creators	3. creators
4. observers	4. observers
5. producers	5. producers (rare!)
<del>6. mutators</del>	6. mutators

Fall 15 CSCI 2600, A. Milanova (slide by Michael Ernst)

42

## Poly, an immutable datatype: overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients. A Poly is:
 *  $c_0 + c_1x + c_2x^2 + \dots$ 
 */
class Poly {
```

Abstract state (specification fields).  
More on this later.

**Overview:** Always state whether **mutable** or **immutable**. Define **abstract model** for use in specification of operations. In ADTs state is **abstract**, not concrete (i.e., this are NOT actual, implementation fields of Poly, just what we call specification fields.)

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

43

## Poly, an immutable datatype: creators

```
// modifies: none
// effects: makes a new Poly = 0
public Poly()

// modifies: none
// effect: makes a new Poly =  $cx^n$ 
// throws: NegExponentException if  $n < 0$ 
public Poly(int c, int n)
```

**Creators:** This is example of overloading, two Poly constructors with different signatures. New object is part of **effects** not preexisting state. Hence, **modifies** is none.

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

44

## Poly, an immutable datatype: observers

```
// returns: degree of this polynomial
public int degree()

// returns: the coefficient of the term of
// this polynomial, whose power is d
public int coeff(int d)
```

**Observers:** Used to obtain information about this polynomial. Return values of **other types**. Never modify the abstract state!

**this:** the current Poly object. Also called the **receiver**.  
Poly x = new Poly(...)  
c = x.coeff(3);

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

45

## Poly, an immutable datatype: producers

```
// modifies: none
// returns: a new Poly with value this + q
public Poly add(Poly q)

// modifies: none
// returns: a new Poly with value this*q
public Poly mul(Poly q)
```

**Producers:** Operations on a type that create other objects of the same type. Common in immutable types. **No side effects**, i.e., cannot change **\_abstract\_** values of any existing object

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

46

## IntSet, a mutable datatype: overview, creators and observers

```
/** Overview: An IntSet is a mutable,
 * unbounded set of integers. E.g.,
 * {  $x_1, x_2, \dots, x_n$  }
 */
class IntSet {

    // effects: makes a new empty IntSet
    public IntSet()

    // returns: true if x in this IntSet,
    // else false
    public boolean contains(int x)
```

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

47

## IntSet, a mutable datatype: mutators

```
// modifies: this
// effects:  $this_{post} = this_{pre} \cup \{ x \}$ 
public void add(int x)

// modifies: this
// effects:  $this_{post} = this_{pre} - \{ x \}$ 
public void remove(int x)
```

**Mutators:** operations that modify receiver **this**. Rarely modify anything other than this. Must list **this** in **modifies** clause. Typically, mutators have no return value.

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

48



## Exercise: String, an immutable datatype

- Overview?
- Creators?  
`String()`, `String(char[] value)`,  
`String(String original)`, ...
- Observers?  
`charAt`, `compareTo`, `contains`, `endsWith`, ...
- Producers?  
`concat`, `format`, `substring`, ...
- Mutators?
  - None!

49

## Exercise: The Stack datatype

```
public Stack()  
  
public boolean empty()  
public E peek()  
public int search(Object o)  
public E push(E item)  
public E pop()
```

50

## ADTs and Java Language Features

- Both classes and interfaces are appropriate
- Java classes
  - Operations in the ADT are **public**
  - Other operations are private
  - Clients can only access ADT operations
- Java interfaces
  - Clients only see the ADT operations
  - Cannot include creators or fields

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

51

## Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- Specifying an ADT
  - immutable
  - mutable
- The ADT design methodology
- **Next: reasoning about ADT implementations**

52