

## Representation Invariants and Abstraction Functions

## Announcements

- Exam 1 on Tuesday Oct. 6<sup>th</sup>
  - Closed book/phone/laptop, 2 cheat pages allowed
  - Reasoning about code, Specifications, ADTs
  - I will post Review slides and practice tests off Announcements page by Tuesday
- HW2 due today
- HW3 out today and due on Fri, before test
- Quiz 3 today, end of class

Fall 15 CSCI 2600, A Milanova

2

## Outline

- Reasoning about ADTs
  - Representation invariants (rep invariants)
  - Representation exposure
  - Checking rep invariants
- Abstraction functions

Fall 15 CSCI 2600, A Milanova

3

## Designing Data Structures

- From domain concept
  - E.g., the math concept of a polynomial, an 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

4

## Specifying an ADT

immutable	mutable
<b>class</b> TypeName	<b>class</b> TypeName
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)

5

## Example: Python Datatypes

Tuples (e.g., (1,"John"))	Lists (e.g., [1,2])
are <b>immutable</b>	are <b>mutable</b>
[j]	[j]
len(t)	len(l)
[j:k]	[j:k]
+	+
*	*
	l.append(e)
	l.reverse()

Fall 15 CSCI 2600, A Milanova

6

## Why ADTs?

- Bridges gap between **domain concept** and **implementation**
- Formalizes **domain concept**. We can reason about correctness of the implementation
- Shields client from implementation. Implementation can vary without affecting client!

Fall 15 CSCI 2600, A. Milanova

7

## Reasoning About ADTs

- ADT is a **specification**, a set of operations
  - E.g., `contains(int i)`, `add(int i)`, etc., (the `IntSet` ADT)
  - `add(Poly q)`, `mul(Poly q)`, etc., (the `Poly` ADT)
- When **specifying** ADTs, there is no mention of data representation!
- When **implementing** the ADT, we must select a specific **data representation**

Fall 15 CSCI 2600, A. Milanova

8

## Implementation of an ADT is Provided by a Class

- To implement the ADT
  - We must select the representation, the **rep**
  - Implement operations in terms of that rep
  - E.g., the **rep** of our `Poly` can be
    - a) `int[] coeffs` or
    - b) `List<Term> terms`
- Choose representation such that
  - It is possible to implement all operations
  - Most frequently used operations are efficient

Fall 15 CSCI 2600, A. Milanova (modified from Michael Ernst, UW)

9

## Connecting Implementation to Specification

- **Representation invariant**: Object  $\rightarrow$  boolean
  - Indicates whether data representation is **well-formed**. Only well-formed representations are meaningful
  - Defines the set of **valid** values
- **Abstraction function**: Object  $\rightarrow$  abstract value
  - What the data structure really **means**
    - E.g., array `[2, 3, -1]` represents  $-x^2 + 3x + 2$
  - How the data structure is to be interpreted

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

10

## IntSet ADT

```
/** Overview: An IntSet is a mutable set
 * of integers. E.g., {  $x_1, x_2, \dots, x_n$  }, {}.
 * There are no duplicates.
 */
// effects: makes a new empty IntSet
public IntSet();
// modifies: this
// effects: thispost = thispre  $\cup$  { x }
public void add(int x);
// modifies: this
// effects: thispost = thispre - { x }
public void remove(int x)
// returns: (x in this)
public boolean contains(int x)
// reruns: cardinality of this
public int size()
```

11

## One Possible Implementation

```
class IntSet {
    private List<Integer> data
        = new ArrayList<Integer>();
    public void add(int x) { data.add(x); }
    public void remove(int x) {
        data.remove(new Integer(x));
    }
    public boolean contains(int x) {
        return data.contains(x);
    }
    public int size() { return data.size(); }
}
```

The representation  
(the rep)

12

## The Representation Invariant

```
■ s = new IntSet();  
  s.add(1); s.add(1); s.remove(1);
```

- The **representation invariant** tells us what's wrong with this code:

```
class IntSet {  
    // Rep invariant:  
    // data has no nulls and no duplicates  
    private List<Integer> data; ...
```

Fall 15 CSCI 2600, A. Milanova

13

## The Representation Invariant

- States data structure **well-formedness**

- E.g., `IntSet` objects, whose data array contains duplicates, e.g., `[1,1]`, are not well-formed

- Must hold before and after every concrete operation!
- Correctness of implementation depends on it

Fall 15 CSCI 2600, A. Milanova

14

## The Representation Invariant

```
class IntSet {  
    // Rep invariant:  
    // data has no nulls and no duplicates  
    private List<Integer> data  
        = new ArrayList<Integer>();  
  
    public void add(int x) {  
        data.add(x);  
    }  
    // Rep invariant may not hold after add!  
}
```

Fall 15 CSCI 2600, A. Milanova

15

## The Representation Invariant

```
class IntSet {  
    // Rep invariant:  
    // data has no nulls and no duplicates  
    private List<Integer> data  
        = new ArrayList<Integer>();  
  
    public void add(int x) {  
        if (!contains(x))  
            data.add(x);  
    }  
    // If rep invariant holds before add, it  
    // holds after add too
```

Fall 15 CSCI 2600, A. Milanova

16

## The Representation Invariant

- Rep invariant excludes ill-formed concrete values

```
class LineSegment {  
    // Rep invariant: !(x1 = x2 && y1 = y2)  
    float x1, y1; // start point  
    float x2, y2; // end point
```

- Conceptually, a line segment is defined by two distinct points. Thus, values with same start and end point (e.g., `x1=x2=1`, `y1=y2=2`), are meaningless. Rep invariant excludes them

Fall 15 CSCI 2600, A. Milanova

17

## The Representation Invariant

- Rep invariant excludes ill-formed concrete values

```
class RightTriangle {  
    // Rep invariant: 0° < angle < 90° &&  
    // base > 0 && base = hypot * cos(toRadians(angle))  
    float base, hypot, angle;
```

// Objects that don't meet the above constraints are  
// not right triangles

Fall 15 CSCI 2600, A. Milanova

18

## Additionally...

- Rep invariant states constraints imposed by specific data structures and algorithms
  - E.g., Tree has no cycles, array must be sorted
- Rep invariant states constraints between fields that are synchronized with each other
  - E.g., **degree** and **coeffs** fields in Poly (if we choose the array data representation)
- In general, **rep invariant states correctness constraints** --- if not met, things can go into terrible mess

19

## Rep Invariant Example

```
class Account {  
    // Rep invariant:  
    // transactions != null  
    // no nulls in transactions  
    // balance >= 0  
    // balance =  $\sum_i$  transactions.get(i).amount  
  
    private int balance;  
    // history of transactions  
    private List<Transaction> transactions;  
    ...  
}
```

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

20

## Another Rep Invariant Example

```
class NameList {  
    //Rep invariant:  $0 \leq \text{index} < \text{names.length}$   
    int index;  
    String[] names;           Where is the bug?  
    ...  
    void addName(String name) {  
        index++;  
        if (index < names.length)  
            names[index] = name;  
    }  
}
```

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

21

## More Rep Invariant Examples

```
class Poly {  
    //Rep. invariant: degree = coeffs.length-1  
    // ... and more ...  
    private int degree;  
    private int[] coeffs;  
    // operations add, sub, mul, eval, etc.  
}  
  
class Poly {  
    //Rep. invariant:  
    //  $E(\text{terms.get}(i)) > E(\text{terms.get}(i+1))$  ...  
    private List<Term> terms;  
    // operations add, sub, mul, eval, etc.  
}
```

22

## Representation Exposure

- Suppose we add this operation to our **IntSet** ADT  
// returns: a List containing the elements  
`public List<Integer> getElements();`
- Suppose we decide on the following implementation  
`public List<Integer> getElements() {  
 return data;  
}`
- What can go wrong with this implementation?

Fall 15 CSCI 2600, A Milanova

23

## Representation Exposure

- Client can get control over rep and break the rep invariant! Consider  
`IntSet s = new IntSet();  
s.add(1);  
List<Integer> li = s.getElements();  
li.add(1); // Breaks IntSet's rep invariant!`
- **Representation exposure** is external access to the rep. **AVOID!!!**
- If you allow representation exposure, document why and how and feel bad about it.

## Representation Exposure

- Make a copy on the way out:

```
public List<Integer> getElements() {  
    return new ArrayList<Integer>(data);  
}
```

- Mutating a copy does not affect IntSet's rep

```
IntSet s = new IntSet();  
s.add(1);  
List<Integer> li = s.getElements();  
li.add(1); //mutates new copy, not IntSet's rep
```

Fall 15 CSCI 2600, A Milanova

25

## Representation Exposure

- Make a copy on the way in too:

```
public IntSet(ArrayList<Integer> elts) {  
    data = new ArrayList<Integer>(elts);  
    ...  
}
```

- Why?

Fall 15 CSCI 2600, A Milanova

26

## Representation Exposure

- How about this:

```
class Movie {  
    private String title;  
    ...  
    public String getTitle() {  
        return title;  
    }  
}
```

- Technically, there is representation exposure
- Representation exposure is dangerous when the rep is **mutable**

- If the rep is immutable, it's ok

27

## Immutability, again

- Suppose we add an iterator

// returns: an Iterator over the IntSet

```
public Iterator iterator();
```

- Suppose the following implementation:

```
public Iterator iterator() {  
    return new Iterator(data);  
}
```

Fall 15 CSCI 2600, A Milanova

28

## Immutability, again

```
class Iterator {  
    private List<Integer> theData;  
    private int next;  
    public Iterator(List<Integer> data) {  
        theData = data;  
        next = 0;  
    }  
    public boolean hasNext() {  
        return (next < theData.size());  
    }  
    public int next() {  
        return theData.get(next++);  
    }  
}
```

Fall 15 CSCI 2600, A Milanova

29

## Checking Rep Invariant

- **checkRep()** or **repOK()**
- Always check if rep invariant holds when debugging
- Leave checks anyway, if they are inexpensive

- Checking rep invariant of IntSet

```
private void checkRep() {  
    for (int i=0; i<data.size; i++)  
        if (data.indexOf(data.elementAt(i)) != i)  
            throw RuntimeException("duplicates!");  
}
```

Fall 15 CSCI 2600, A Milanova

30

## Practice Defensive Programming

- Assume that you will make mistakes
- Write code to catch them
  - On method entry
    - Check rep invariant (i.e., call `checkRep()`)
    - Check preconditions (requires clause)
  - On method exit
    - Check rep invariant (call `checkRep()`)
    - Check postconditions
- Checking rep invariant helps **find bugs**
- Reasoning about rep invariant helps **avoid bugs**

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

31

## Connecting Implementation to Specification

- **Representation invariant:** Object  $\rightarrow$  boolean
  - Indicates whether data representation is **well-formed**. Only well-formed representations are meaningful
  - Defines the set of **valid** values
- **Abstraction function:** Object  $\rightarrow$  abstract value
  - What the data representation really **means**
    - E.g., array [2, 3, -1] represents  $-x^2 + 3x + 2$
  - How the data structure is to be interpreted

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

32