Implementing an ADT: Representation invariants

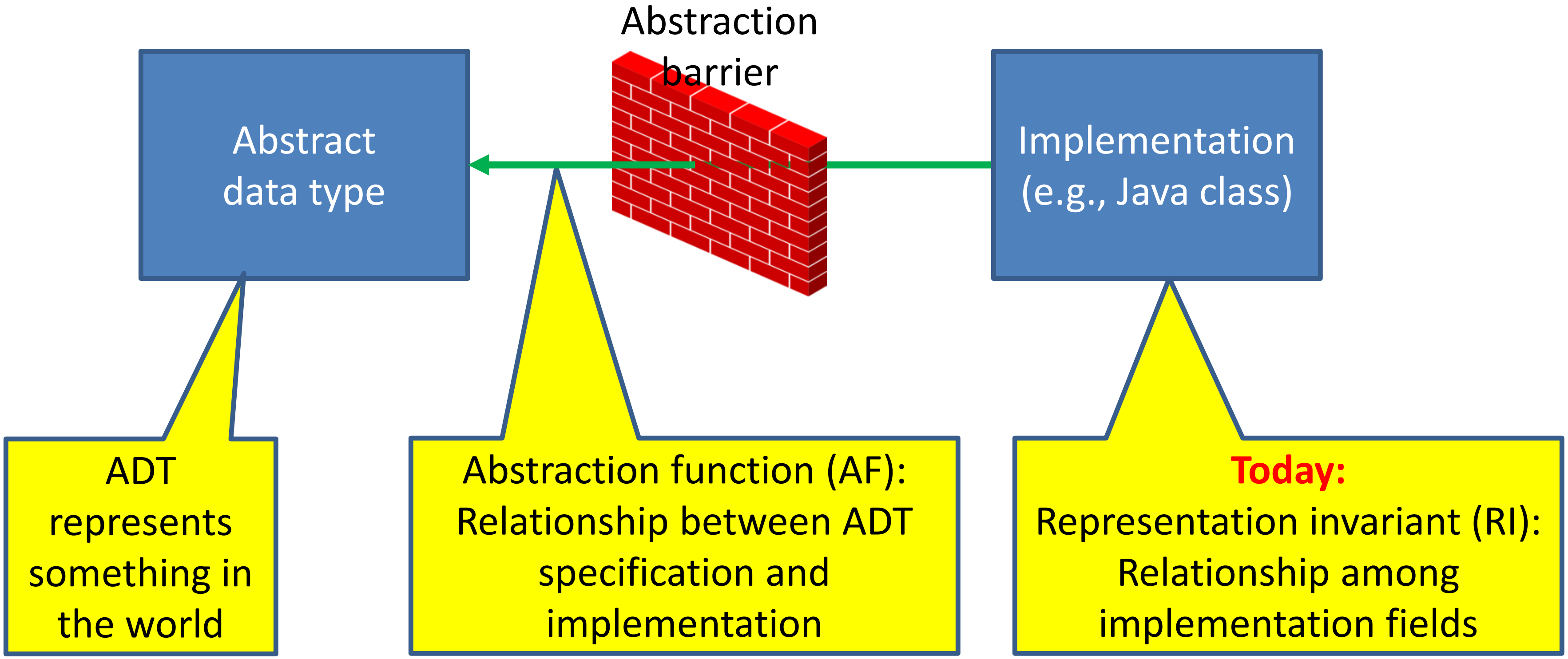
CSE 331
University of Washington

Michael Ernst

Outline of data abstraction lectures

ADT
specification

ADT
implementation



Abstraction
barrier

Abstract
data type

Implementation
(e.g., Java class)

ADT
represents
something in
the world

Abstraction function (AF):
Relationship between ADT
specification and
implementation

Today:
Representation invariant (RI):
Relationship among
implementation fields

Review: A data abstraction is defined by a specification

An ADT is a collection of procedural abstractions

- *Not* a collection of procedures

Together, these procedural abstractions provide:

- A set of values
- ***All*** the ways of directly using that set of values
  - Creating
  - Manipulating
  - Observing

Creators and producers: make new values

Mutators: change the value (affect **`equals()`** but not **`==`**)

Observers: allow the client to distinguish different values

ADTs and specifications

Specification: only in terms of the abstraction

- Never mentions the representation

How should we implement an ADT?

How can we ensure the implementation satisfies the specification?

Connecting specifications and implementations

Representation invariant: Object → boolean

Is a concrete rep (an instance) well-formed?

Abstraction function: Object → abstract value

What the instance means as an abstract value

Example: Rectangle (**`getHeight`**, **`getWidth`**, **`getArea`**)

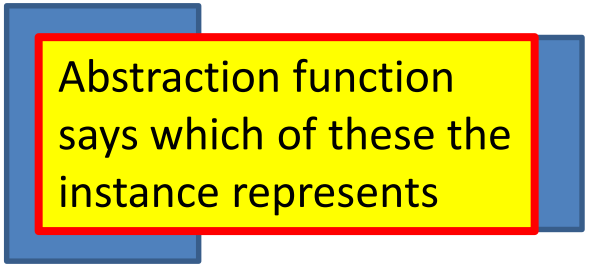
3, 4, 12

5, 5, 25

Rep invariant says which of these are valid

6, 7, 8

2, -10, -20



Abstraction function says which of these the instance represents

Review:
Implementing a data abstraction (ADT)

To implement a data abstraction:

- Select the representation of instances, the *rep*
- Implement operations in terms of that rep

Choose a representation so that

- It is possible to implement operations
- The most frequently used operations are efficient
  - You don’t know which these will be
  - Abstraction allows the rep to change later

CharSet Abstraction

// Overview: A CharSet is a finite mutable set of Characters

// effects: creates a fresh, empty CharSet
public CharSet()

// modifies: this
// effects: $\text{this}_{\text{post}} = \text{this}_{\text{pre}} \cup \{c\}$
public void insert(Character c);

// modifies: this
// effects: $\text{this}_{\text{post}} = \text{this}_{\text{pre}} - \{c\}$
public void delete(Character c);

// returns: $(c \in \text{this})$
public boolean member(Character c);

// returns: cardinality of this
public int size();

A CharSet implementation

```
class CharSet {
  private List<Character> elts
    = new ArrayList<Character>();

  public void insert(Character c) {
    elts.add(c);
  }
  public void delete(Character c) {
    elts.remove(c);
  }
  public boolean member(Character c) {
    return elts.contains(c);
  }
  public int size() {
    return elts.size();
  }
}
```

A buggy CharSet implementation.

What client code will expose the error?

```
class CharSet {
  private List<Character> elts
    = new ArrayList<Character>();

  public void insert(Character c) {
    elts.add(c);
  }
  public void delete(Character c) {
    elts.remove(c);
  }
  public boolean member(Char
    return elts.contains(c);
  }
  public int size() {
    return elts.size();
  }
}
```

```
CharSet s = new CharSet();
s.insert('a');
s.insert('a');
s.delete('a');
if (s.member('a'))
    // print “wrong”;
else
    // print “right”;
```

Where is the defect?

Where is the defect?

The answer to this question tells you what code needs to be fixed

Perhaps **`delete`** is wrong (and so is **`size`**)

- It should remove all occurrences

Perhaps **`insert`** is wrong

- It should not insert a character that is already there

How can we know?

- The representation invariant tells us

The representation invariant

- Defines data structure well-formedness
  - Which instances/reps are valid?
- Holds before and after every **`CharSet`** operation
- Operation implementations (methods) may depend on it

Write it this way:

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  …
```

Or, if you are the pedantic sort:

$\forall$ indices i of elts . elts.elementAt(i) $\neq$ null

$\forall$ indices i, j of elts .

$i \neq j \Rightarrow \neg$ elts.elementAt(i).equals(elts.elementAt(j))

Now, we can locate the error

```
// Rep invariant:
// elts has no nulls and no duplicates

public void insert(Character c) {
  elts.add(c);
}

public void delete(Character c) {
  elts.remove(c);
}
```

Another rep invariant example

```
class Account {
    private int balance;
    // history of all transactions
    private List<Transaction> transactions;
    …
}
```

Real-world constraints:

- balance ≥ 0
- balance $= \Sigma_i$ transactions.get(i).amount

Implementation-related constraints:

- transactions $\neq$ null
- no nulls in transactions

Listing the elements of a CharSet

Consider adding the following method to CharSet:

```
// returns: a List containing the members of this
public List<Character> getElts();
```

Consider this implementation:

```
// Rep invariant: elts has no nulls and no duplicates
public List<Character> getElts() { return elts; }
```

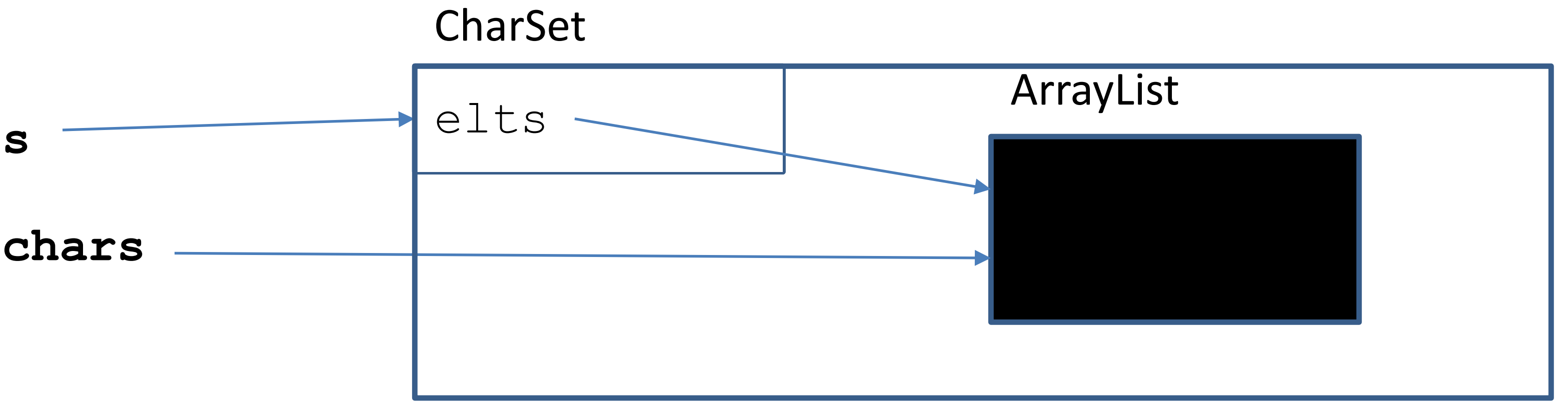
Does the implementation of getElts preserve the rep invariant?

… sort of

Representation exposure

Consider this client code (outside the CharSet implementation):

```
CharSet s = new CharSet();
s.insert('a');
List<Character> chars = s.getElts();
chars.add('a');
s.delete('a');
if (s.member('a')) …
```



Representation exposure

Consider this client code (outside the CharSet implementation):

```
CharSet s = new CharSet();
s.insert('a');
List<character> chars = s.getElts();
chars.add('a');
s.delete('a');
if (s.member('a')) …
```

Representation exposure is external access to the rep

A **big deal**, a **common bug**. Now you have a name for it.

Representation exposure is almost always EVIL

Enables violation of abstraction boundaries and the rep invariant

If you do it, document why and how

And feel guilty about it!

How to avoid/prevent rep exposure: **immutability** or **copying**

Avoid rep exposure #1: Immutability

Aliasing is no problem if the client cannot change the data

Assume **`Point`** is an *immutable* ADT:

```
class Line {
  private Point start;
  private Point end;
  public Line(Point start, Point end) {
    this.start = start;
    this.end = end;
  }
  public Point getStart() {
    return this.start;
  }
  …
```

Pros and cons of immutability

Immutability greatly simplifies reasoning

- Aliasing does not matter
- No need to make copies with identical contents
- Rep invariants cannot be broken

Can be less efficient (new objects for every modification)
Can be more efficient (no need for redundant copies)

Does require different designs.
Suppose **`Point`** is immutable but **`Line`** is mutable:

```
class Line {
  …
  void raiseLine(double deltaY) {
    this.start = new Point(start.x, start.y + deltaY);
    this.end = new Point(end.x, end.y + deltaY);
  }
```

Immutable Java classes include **`Character`**, **`Color`**, **`File`** (path), **`Font`**, **`Integer`**, **`Locale`**, **`String`**, **`URL`**, …

Are `private` and `final` enough?

Making fields **`private`**

- Is necessary to prevent rep exposure (why?)
- Is insufficient to prevent rep exposure (see CharSet example)
- The real issue is aliasing of mutable data

Making fields **`final`**

- Is neither necessary nor sufficient to achieve immutability
- A **`final`** field cannot be reassigned
  - But it can be mutated (its fields can be reassigned and/or mutated)

```
class Line {
  private final Point start;
  private final Point end;
  …
  public void translate(int deltaX, int deltaY) {
    start.x += deltaX;
    start.y += deltaY;
    end.x += deltaX;
    endy. += deltaY
  }
  …
```

Avoiding rep exposure #2: Copying

Copy data that crosses the abstraction barrier

Example (assume Point is a mutable ADT):

```
class Line {
  private Point start;
  private Point end;

  public Line(Point start, Point end) {
    this.start = new Point(start.x, start.y);
    this.end = new Point(end.x, end.y);
  }

  public Point getStart() {
    return new Point(this.start.x, this.start.y);
  }
  …
```

Copy in: parameters that become part of the implementation

Copy out: results that are part of the implementation

Shallow copying is not enough

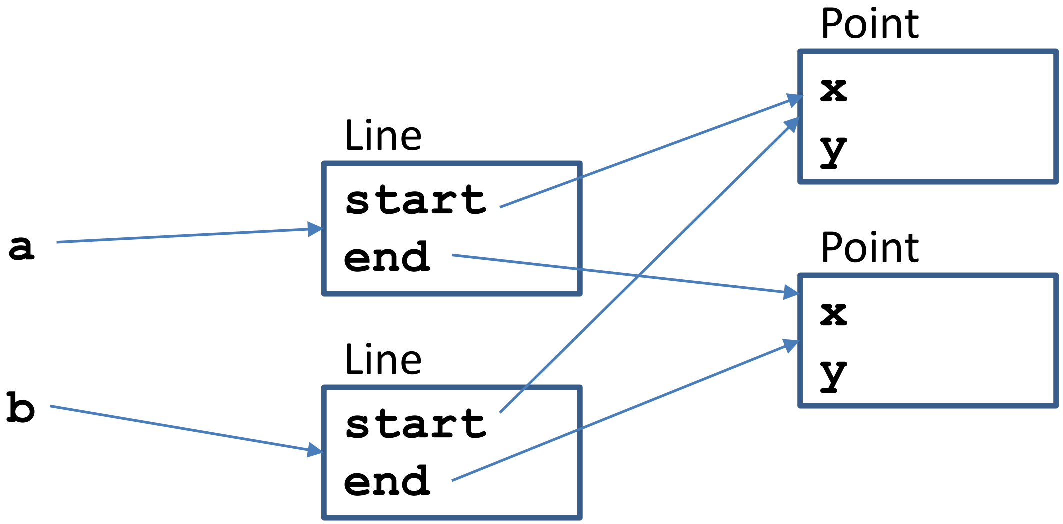
Example (assume Line and Point are mutable ADTs):

```
class Line {
  private Point start;
  private Point end;

  public Line(Line other) {
    this.start = other.start;
    this.end = other.end;
  }
}
```

<u>Client code:</u>

```
Line a = …;
Line b = new Line(a);
a.translate(3, 4);
```



Deep copying is not necessary

Must copy-in and copy-out “all the way down” to immutable parts

This combines our two ways to avoid rep exposure: **immutability** and **copying**.

Avoiding rep exposure #3: Readonly wrapper (“immutable copy”)

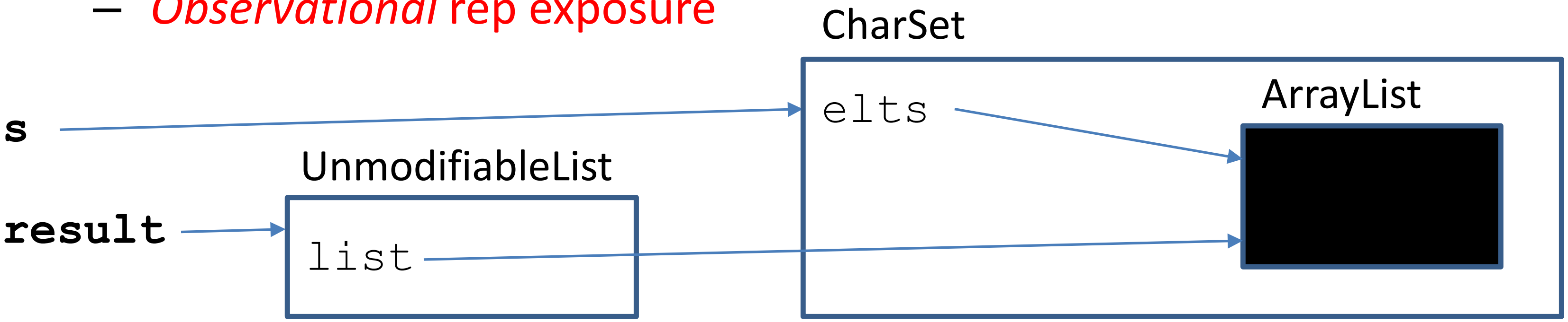
```
class CharSet {
  private List<Character> elts = …;

~~  public List<Character> getElts() { // copy~~
~~    return new ArrayList(elts);~~
~~  }~~
  public List<Character> getElts() { // readonly wrapper
    return Collections.unmodifiableList(elts);
  }
```

```
CharSet s;
result = s.getElts();
s.add(’a’);
```

`unmodifiableList()`: result can be read but not modified

- Doesn’t make a copy: its rep is aliased to its input (efficient!)
- Attempts to modify throw **`UnsupportedOperationException`**
- Still need to copy on the way in
- *Observational* rep exposure



The specification "Returns a list containing the elements"

Could mean any of these things:

1. Returns a fresh mutable list containing the elements in the set *at the time of the call.*
   - Difficult to implement efficiently
2. Returns read-only view that is *always up to date* with the current elements in the set.
   - Makes it hard to change the rep later
3. Returns a list containing the current set elements. *Behavior is unspecified* if client attempts to mutate the list or to access the list after mutating the set.
   - Weaker than #1 and #2
   - Less simple, harder to use, but sufficient for some purposes

Lesson: A seemingly simple spec may be ambiguous and subtle!

Avoiding representation exposure

Understand what representation exposure is

Design ADT implementations to prevent it

Prove that your ADT is free of representation exposure

Test for it with adversarial clients:

- Pass values to methods and then mutate them
- Mutate values returned from methods
- Check the rep invariant (in addition to client behavior)

Fix any rep exposure bugs

Checking rep invariants

Should code check that the rep invariant holds?

- Yes, if it’s inexpensive
- Yes, for debugging (even when it’s expensive)
- It’s quite hard to justify turning the checking off
- Some private methods need not check (Why?)
- Some private methods should not check (Why?)

Checking the rep invariant

Rule of thumb: check on entry *and* on exit (why?)

```
public void delete(Character c) {
  checkRep();
  elts.remove(c)
  // Is this guaranteed to get called?
  // See handouts for a less error-prone way to check at exit.
  checkRep();
}
…
/** Verify that elts contains no duplicates. */
private void checkRep() {
  for (int i = 0; i < elts.size(); i++) {
    assert elts.indexOf(elts.elementAt(i)) == i;
  }
}
```

Practice defensive programming

Assume that you will make mistakes

Write and incorporate code designed to catch them

- On entry:
  - Check rep invariant
  - Check preconditions (requires clause)
- On exit:
  - Check rep invariant
  - Check postconditions

Checking the rep invariant helps you discover errors

Reasoning about the rep invariant helps you avoid errors

- Or prove that they do not exist!

The rep invariant constrains structure, not meaning

New implementation of insert that preserves the rep invariant:

```
public void insert(Character c) {
  Character cc = new Character(encrypt(c));
  if (!elts.contains(cc))
    elts.addElement(cc);
}
public boolean member(Character c) {
  return elts.contains(c);
}
```

The program is still wrong

- Clients observe incorrect behavior
- What client code exposes the error?
- Where is the error?
- We must consider the meaning
- The *abstraction function* helps us

```
CharSet s = new CharSet();
s.insert('a');
if (s.member('a'))
    // print "right";
else
    // print "wrong";
```