CSE 331
Software Design & Implementation

Hal Perkins

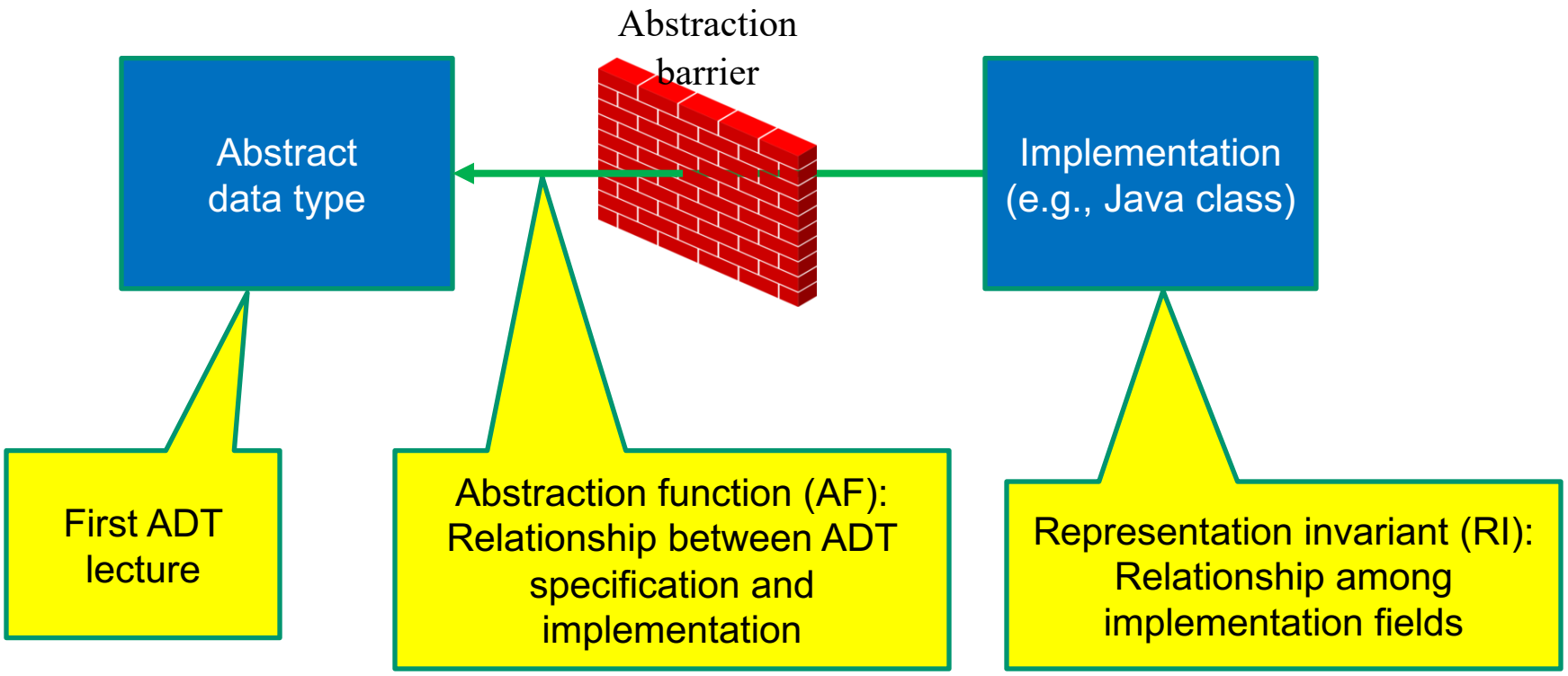
Winter 2020

Abstraction Functions

Data abstraction outline

ADT specification

ADT implementation



Connecting implementations to specs

Representation Invariant: maps Object → boolean

- An assertion about the object state
- Indicates if an instance is *well-formed*
- Defines the set of valid concrete values
- Only values in the valid set make sense as implementations of an abstract value
- **For implementors/debuggers/maintainers of the abstraction: no object should *ever* violate the rep invariant**
  - Such an object has no useful meaning

Abstraction Function: maps Object → abstract value

- What the data structure *means* as an abstract value
- How the data structure is to be interpreted
- Only defined on objects meeting the rep invariant
- **For implementors/debuggers/maintainers of the abstraction:** Each procedure should meet its spec (abstract values) by "doing the right thing" with the concrete representation

Rep inv. constrains structure, not meaning

An 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(Cha
  return elts.contains(c)
}
```

```
CharSet s = new CharSet();
s.insert('a');
if (s.member('a'))
    …
```

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

Abstraction function: rep→abstract value

The abstraction function maps the concrete representation to the abstract value it represents

AF: Object → abstract value

AF(CharSet this) = { c | c is contained in this.elts }

"set of Characters contained in this.elts"

Not executable because abstract values are "just" conceptual

The abstraction function lets us reason about what [concrete] methods do in terms of the clients' [abstract] view

Abstraction function and `insert`

Goal is to satisfy the specification of insert:

```
// modifies: this
// effects: this_post = this_pre U {c}
public void insert (Character c) {…}
```

The AF tells us what the rep means, which lets us place the blame

AF(CharSet this) = { c | c is contained in this.elts }

Consider a call to (buggy) **`insert`**:

On *entry*, abstract meaning of rep is AF($this_{pre}$) = $elts_{pre}$

On *exit*, meaning is AF($this_{post}$) = AF($this_{pre}$) U {encrypt('a')}

which is not what we want….

What if we used this abstraction function instead?

AF(this) = { c | encrypt(c) is contained in this.elts }

= { decrypt(c) | c is contained in this.elts }

The abstraction function is a function

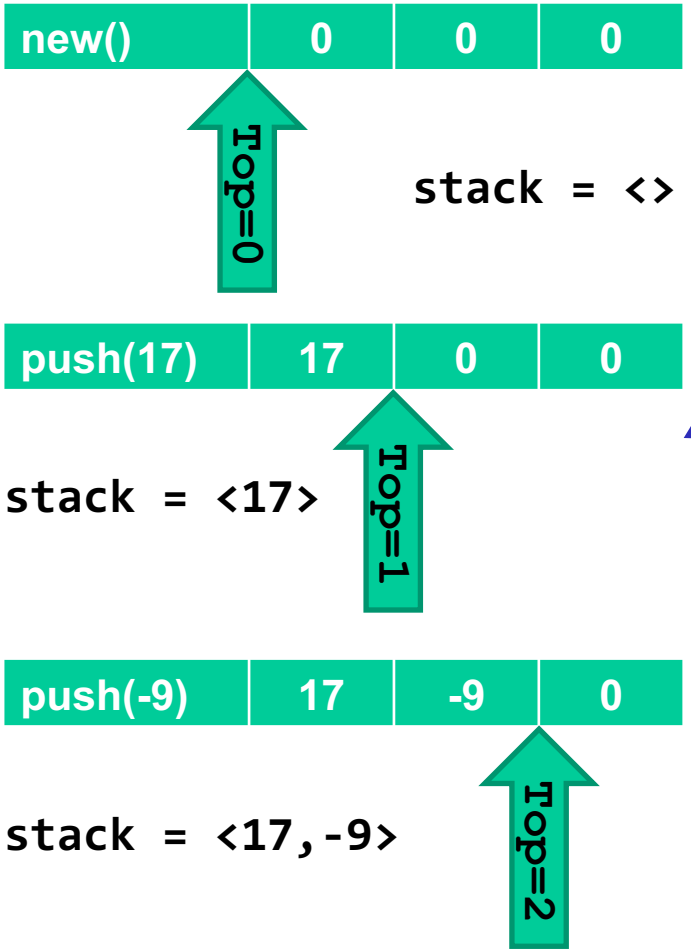
Why do we map concrete to abstract and not vice versa?

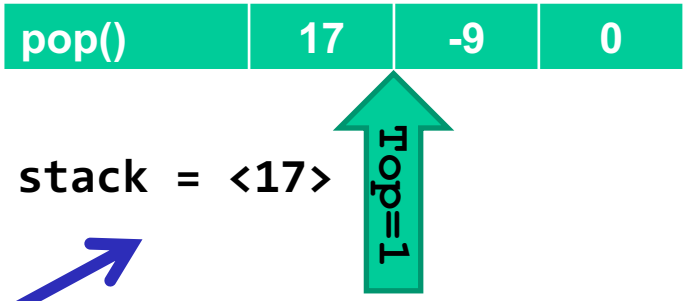
- It's not a function in the other direction
  - Example: lists `[a,b]` and `[b,a]` might each represent the set `{a,b}`

- It's not as useful in the other direction
  - Purpose is to reason about whether our methods are manipulating concrete representations correctly in terms of the abstract specifications

Stack AF example

Abstract stack with array and "top" index implementation





Abstract states are the same

`stack = <17> = <17>`

Concrete states are different

`<[17,0,0], top=1>`

≠

`<[17,-9,0], top=1>`

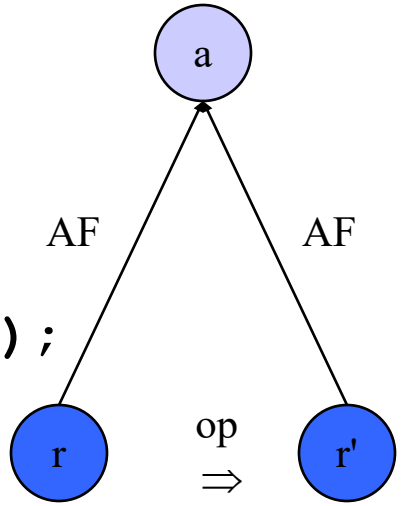
AF is a function

Inverse of AF is not a function

Benevolent side effects

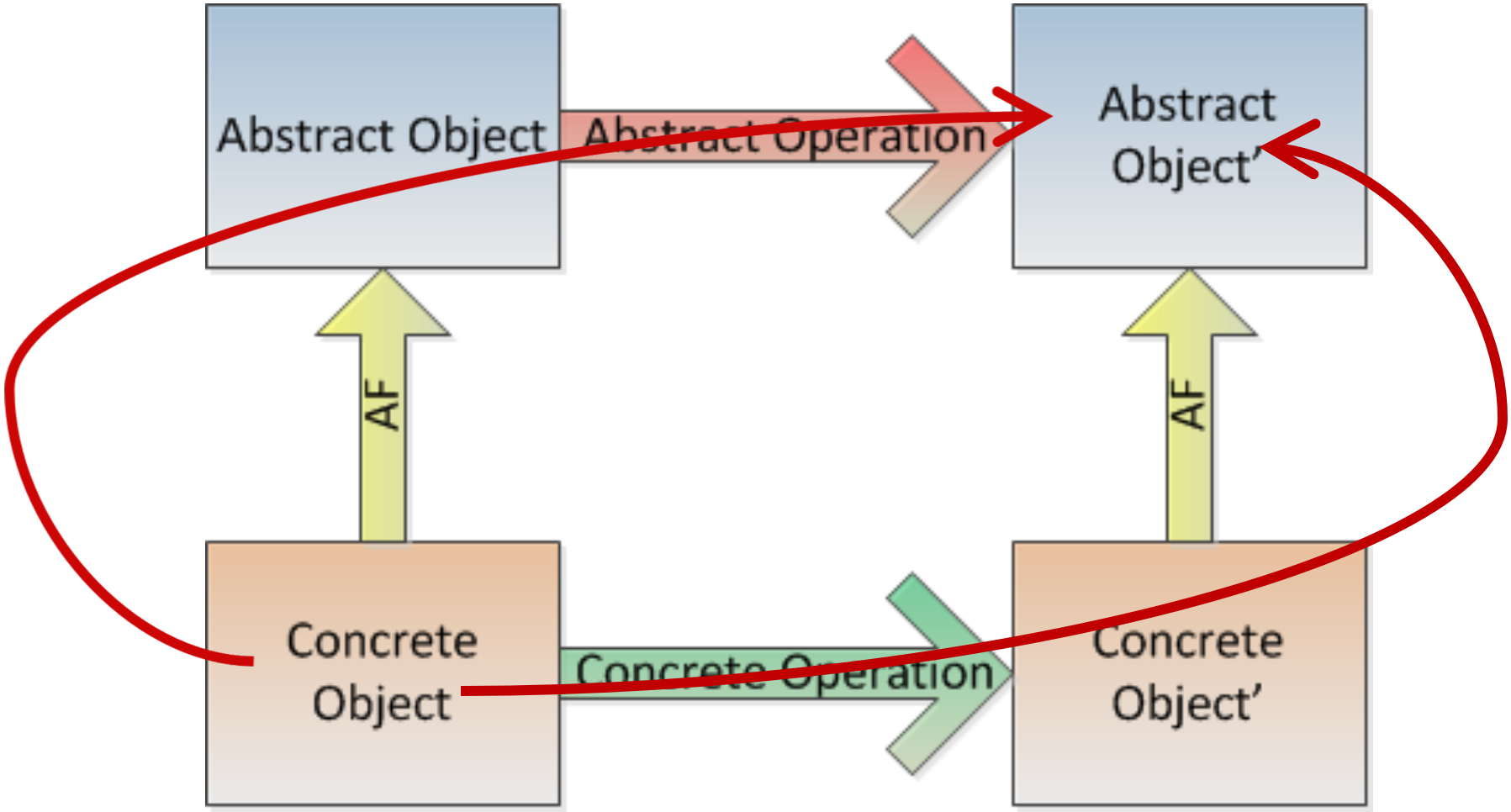
Different implementation of **`member`**:

```
boolean member(Character c1) {
  int i = elts.indexOf(c1);
  if (i == -1)
    return false;
  // move-to-front optimization
  Character c2 = elts.elementAt(0);
  elts.set(0, c1);
  elts.set(i, c2);
  return true;
}
```



- Move-to-front speeds up repeated membership tests
- Mutates rep, but does not change *abstract* value
  - *AF maps both reps to the same abstract value*
    - Precise reasoning/explanation for "clients can't tell"

For any correct operation…



Writing an abstraction function

Domain: all representations that satisfy the rep invariant

Range: can be tricky to denote

For mathematical entities like sets: easy

For more complex abstractions: give names to "fields" – parts of the abstract value

- AF defines the value of each "specification field"
  - (Course notes have examples of complex AFs with many spec. fields, but go for simple, correct, understandable and use fields only when they contribute to clarity & precision. Often don't need lots of complex fields.)

Overview section of the specification should provide a notation for writing abstract values

- A printed representation (`toString`) is valuable for debugging

Summary: connecting data abstractions (ADTs) to implementations

Rep invariant

- Which concrete values represent abstract values

Abstraction function

- For each concrete value, which abstract value it represents

Together, they modularize the implementation

- Neither one is part of the ADT's specification
- Both are needed to reason that an implementation satisfies the specification

When you program,

- Always write a rep invariant (standard industry best practice)
- Write an AF when you need it (you need it for all 331 code)
  - Write at least an informal one for all non-trivial classes