# Implementing an Abstract Data Type

CSE 331: Software Design and Implementation, Autumn 2026

*Your job is to deliver code that you know works.*

# TODO

- Homework 2: due tonight, 11:59pm
- Reading 3 is out: due Friday, October 30
- Tuesday's section: abstract data types

# A faster join

- `ListQueue`'s `join` walks the list to check for the student
- `FastQueue` keeps the students twice: a list for the order, and a set

```
12  public class FastQueue implements HelpQueue {
13      // The students waiting, front first.
14      private List<String> queue;
15      // The same students, for a fast lookup.
16      private Set<String> members;
```

**Two fields hold the same students, so they have to agree**

# Still correct?

```
50  public void join(String netid) {
52      if (members.contains(netid)) {
        ...
55      queue.add(netid);
56      members.add(netid);
58  }
```

```
60  public String next() {
        ...
65      String front = queue.remove(0);
66      members.remove(front);
68      return front;
69  }
```

- The check asks the set instead of walking the list
- So `join` and `next` must change the set too, not only the list
- The set holds the same students as the list, so `join` behaves as before

**All of this rests on the set holding the same students as the list**

# A buggy next()

```
60   public String next() {
62       if (queue.isEmpty()) {
63           throw new NoSuchElementException("nobody is waiting");
64       }
65       String front = queue.remove(0);
66       members.remove(front);
68       return front;
69   }
```

**How can a client exploit this bug to cause a specification violation?**

# A buggy next(): what goes wrong?

```
60  public String next() {
        ...
65      String front = queue.remove(0);
66      members.remove(front);
68      return front;
```

```
4   FastQueue queue = new FastQueue();
5   queue.join("ada");
6   queue.next();
7   queue.join("ada");
8
9   FastQueue other = new FastQueue();
10  other.join("ada");
11  other.join("bob");
12  other.next();
13  other.size();
```

What should **join** do here?

ada is not in the queue, so the specification says: add ada

What does it do?

It throws "ada is in the queue"

Something is wrong, since the specification was violated

But which method's code is wrong?

What should **size** return here?

One student is waiting, so 1

What does it return?

2

# Which method is wrong?

```
45  public int size() {
47      return members.size();
48  }
50  public void join(String netid) {
52      if (members.contains(netid)) {
        ...
55      queue.add(netid);
56      members.add(netid);
58  }
```

```
60  public String next() {
        ...
65      String front = queue.remove(0);
66      members.remove(front);
68      return front;
69  }
```

```
 4  FastQueue queue = new FastQueue();
 5  queue.join("ada");
 6  queue.next();
 7  queue.join("ada");
 8
 9  FastQueue other = new FastQueue();
10  other.join("ada");
11  other.join("bob");
12  other.next();
13  other.size();
```

Is the bug here? It threw

Or here? It returned 2

Or here? A line is missing, but the client never saw `next` break its specification

The specification cannot say, since `members` is a private field: what was it meant to hold?

Students who left too: `join` and `size` are wrong. Only the queue: `next` is wrong

# Representation invariant

## Definition: representation invariant

A **representation invariant** (RI) of a class is a claim about its fields that is true of every object of the class whenever no method of the class is running.

The constructor makes it true. Every method may assume it when called and must make it true again before it returns. An implementation should document its representation invariant with a comment like `// RI: ...` beside the fields.

**Abstract invariant: the abstract state. RI: the concrete state, the fields**

# The RI settles it

```
65        String front = queue.remove(0);
66        members.remove(front);
68        return front;
```

- The RI: *members holds exactly the NetIDs in queue*
- Every method must make it true again before it returns
- Removing the front from `queue` means removing it from `members` too
- Without that line, `next` returns with the RI false: `next` is wrong
- `join` and `size` may assume the RI, and the client saw them misbehave
  - Because they ran from a state the RI forbids: not their bug

**Reasoning finds the mistake where it was made; a test would see it a call later**

# checkRep

- Catch a bug as soon as the RI is violated

**Definition: checkRep**

A **checkRep** method of a class checks the class's representation invariant in code: it reads the fields, computes each clause of the RI from them, and throws an exception if any clause is false.
It is private, since it is the implementer's check and not part of the specification, and it changes no field.

# FastQueue's checkRep

```
36  private void checkRep() {
37      if (!members.equals(new HashSet<String>(queue))) {
38          throw new IllegalStateException("members is not the queue");
39      }
40      if (members.size() != queue.size()) {
41          throw new IllegalStateException("someone is in queue twice");
42      }
43  }
```

- Each `if` checks one clause of the RI, and throws if it is false
- Called at the end of a constructor, and at both ends of a method

**With the buggy `next` from before, now where does it fail?**

# checkRep, the run

```
65  String front = queue.remove(0);
66  members.remove(front);
67  checkRep();
68  return front;
```

```
4  FastQueue queue = new FastQueue();
5  queue.join("ada");
6  queue.next();
7  queue.join("ada");
```

**next** throws "members is not the queue", from the check at its end

The failure is in **next**, on the call that broke the RI, not a call later

Without the checks, **join** took the blame

# Where to call checkRep

- A method that changes the object, as `join` and `next` do, is a **mutator**
- The code quality guide: call `checkRep`
  - at the end of the constructor;
  - at the beginning and end of every mutator method;
  - at the beginning of every non-mutator method.
- `checkRep` builds a set from the whole list, at every check
  - So `join` is fast only once the checks are off

# Half sheet, question 1 (3 minutes)

- **1.** On what line of `join` does the RI become false? When true again?
  - Recall the RI: *members holds exactly the NetIDs in queue*

```
50  public void join(String netid) {
51      checkRep();
52      if (members.contains(netid)) {
53          throw new IllegalArgumentException(netid + " is in the queue");
54      }
55      queue.add(netid);
56      members.add(netid);
57      checkRep();
58  }
```

# A faster next

```
60  public String next() {
        ...
65      String front = queue.remove(0);
66      members.remove(front);
68      return front;
69  }
```

- `queue.remove(0)` moves every student after the first one

**How could `next` be made fast?**

- Removing the *last* element moves nothing: so two lists
  - `join` adds at the inbox's end; `next` takes from the outbox's end
  - Outbox empty? `next` moves the inbox over in reverse

# A faster next, the fields

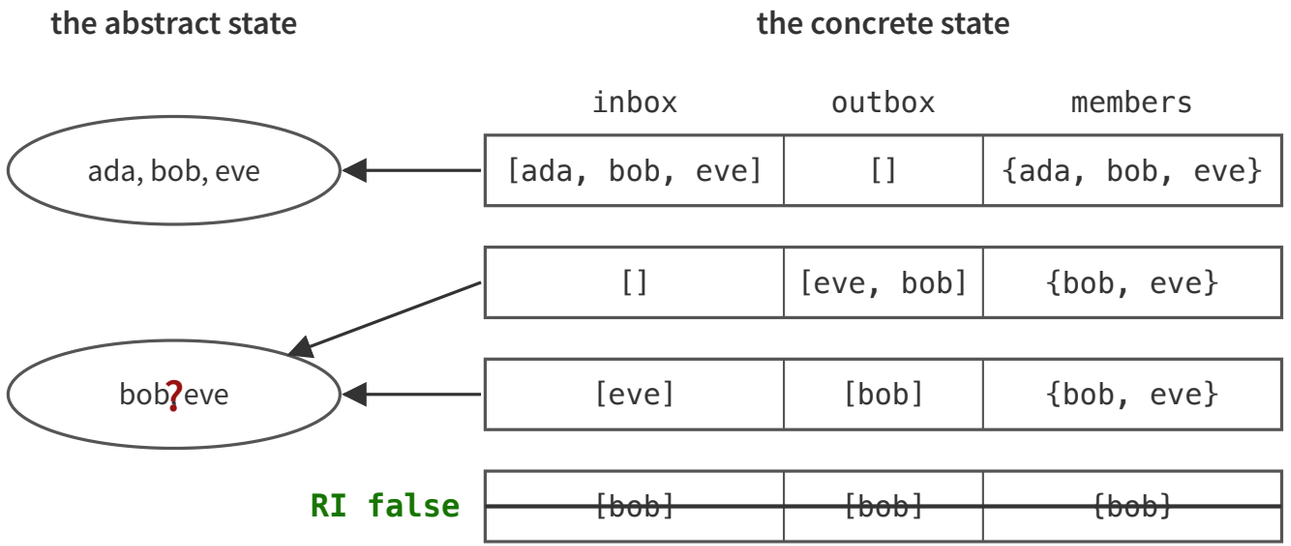
```
13  // The students who joined since the last refill, oldest first.
14  private List<String> inbox;
15  // The students ahead of them, with the front of the queue last.
16  private List<String> outbox;
17  // The same students, for a fast lookup.
18  private Set<String> members;
19  // RI: no NetID is in inbox or outbox twice, or in both, and
20  //     members holds exactly the NetIDs in inbox and outbox.
```

**The RI now names both lists**

- Wednesday's five tests and one with three students: none fails

# Which queue is this?

- `next`'s specification: *removes the student at the front*
- `inbox` = `[]`, `outbox` = `[eve, bob]`, `members` = `{bob, eve}`
- Which student is at the front? Which queue is this?



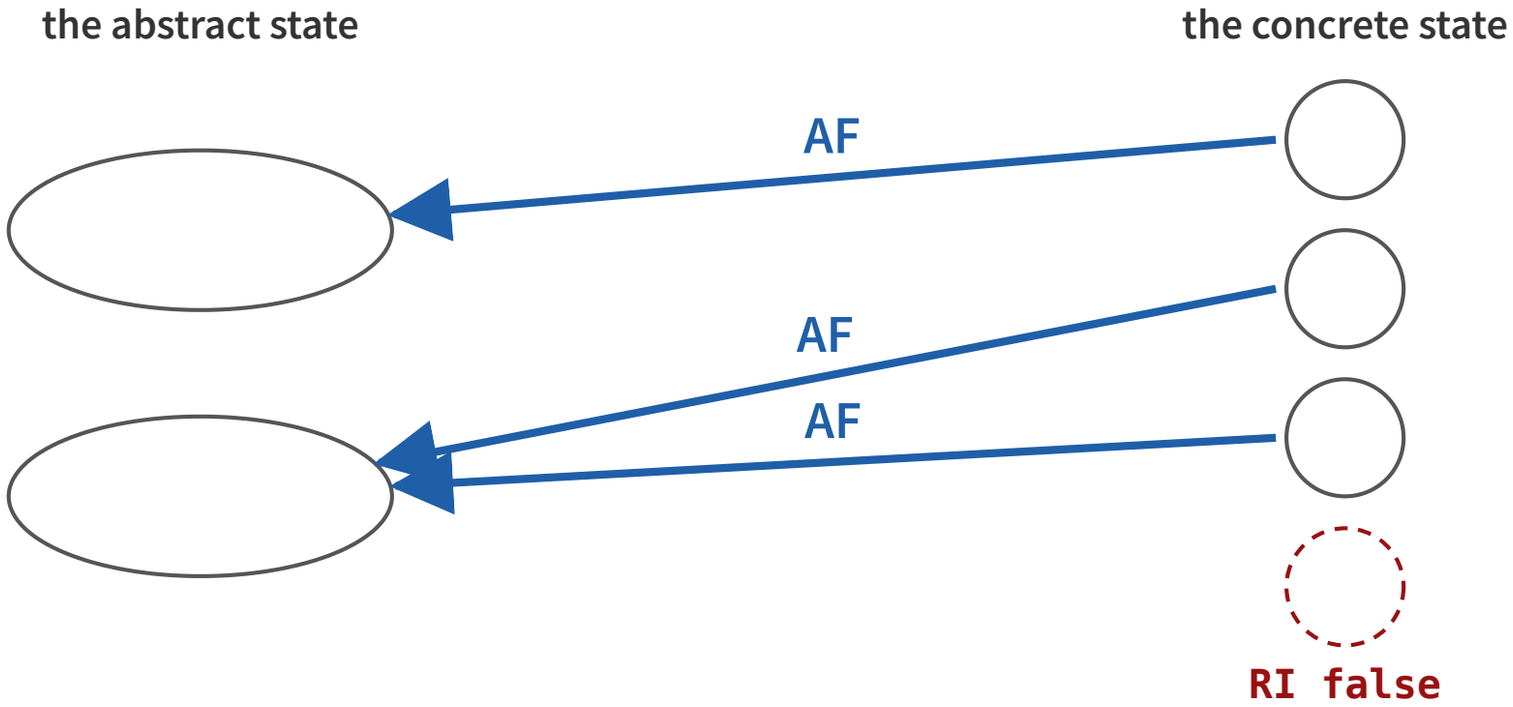
# Abstraction function

**Definition: abstraction function**

The **abstraction function** (AF) of a class maps each value of the fields that the representation invariant allows to the abstract state it stands for.
The AF is a mathematical function, not code. An implementation should document its abstraction function with a comment like `// AF(this) = ...` beside the fields.

- **RI**: which field values are allowed. **AF**: what they stand for

# Abstraction function, as a map




- Defined only where the RI is true
- Many values of the fields, one abstract state; no arrow goes back

# FasterQueue's abstraction function

```
19  // RI: no NetID is in inbox or outbox twice, or in both, and
20  //     members holds exactly the NetIDs in inbox and outbox.
21  // AF(this) = the help queue of the students in outbox, last to
22  //     first, then the students in inbox, first to last.
```

- `[]`, `[eve, bob]`, `{bob, eve}`: the queue bob, eve
- `[eve]`, `[bob]`, `{bob, eve}`: the queue bob, eve

# A buggy `waiting()`

```
86  public List<String> waiting() {
87      checkRep();
88      List<String> list = new ArrayList<String>();
89      for (int i = outbox.size() - 1; i >= 0; i--) {
89          list.addAll(outbox);
90          list.add(outbox.get(i));
91      }
92      list.addAll(inbox);
93      return list;
94  }
```

**How can a client exploit this bug to cause a specification violation?**

# What made it wrong?

```
89      for (int i = outbox.size() - 1; i >= 0; i--) {
89          list.addAll(outbox);
90              list.add(outbox.get(i));
91      }
92          list.addAll(inbox);
```

```
5   queue.join("ada");
6   queue.join("bob");
7   queue.join("eve");
8   queue.next();
9   queue.waiting();
```

What should `waiting` return here?

ada was served, so the specification says: bob, eve, front first

What does it return?

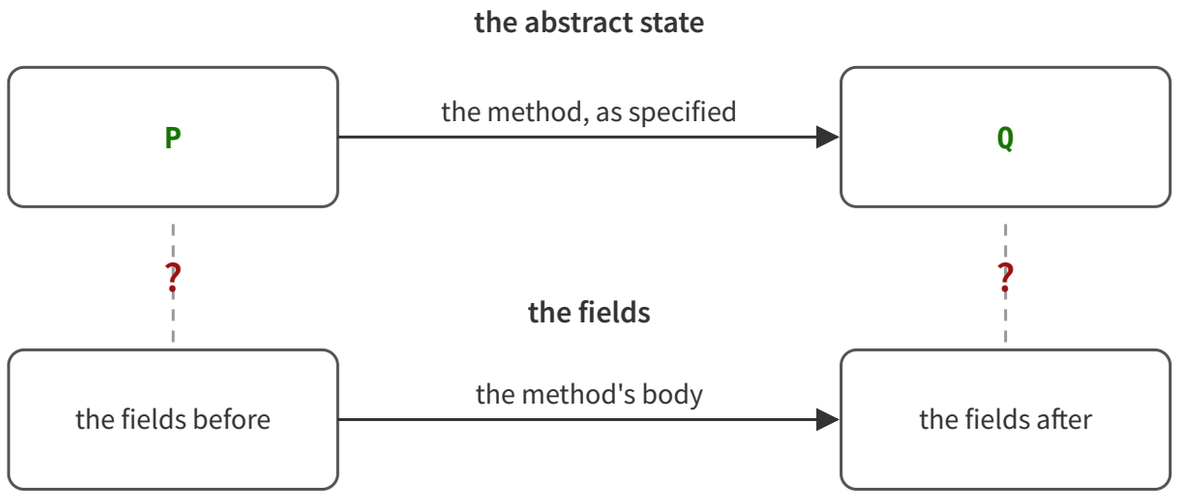
[eve, bob]

With two students the outbox holds one, so Wednesday's five tests all pass

The client's call refutes it; the AF says what it got wrong: the outbox is last to first

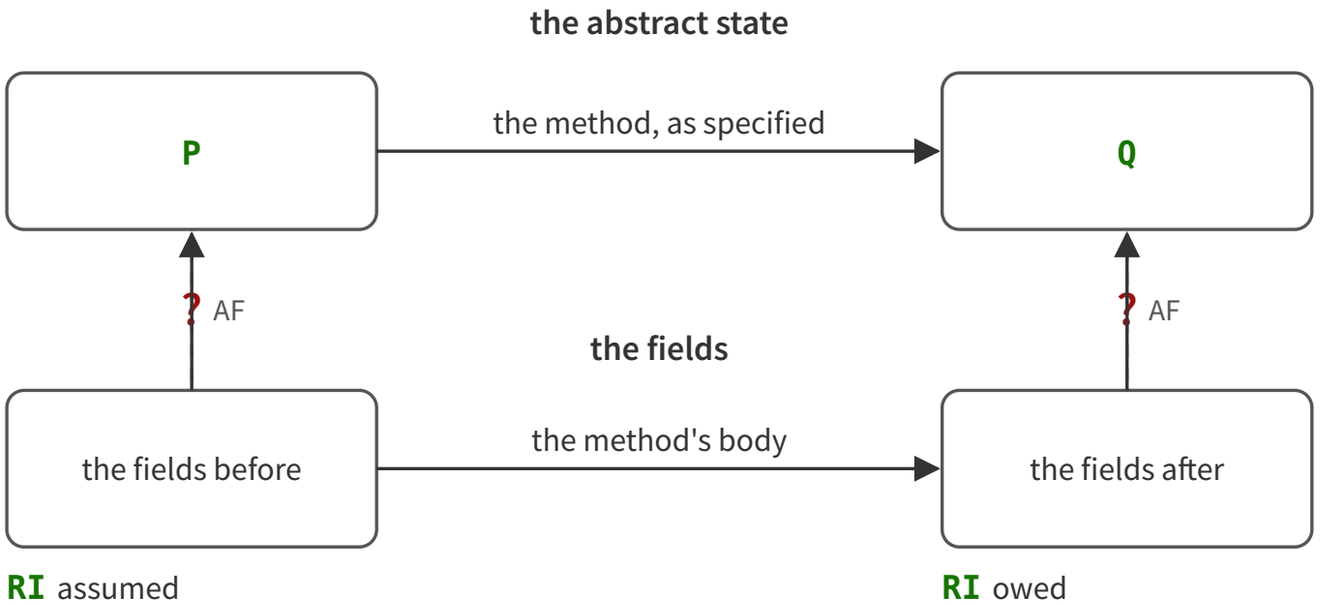
# The problem

- A specification is written about the **abstract state**
- An implementation is written about the **concrete state**, its fields
- Between the promise and the code, a gap: what do the fields stand for?



# What correct means

- The specification is about the queue; the body is about the fields
- The AF connects them
- The RI is assumed before and owed after



# What correct means, the rule

**Rule: a method of an abstract data type is correct**

A method with precondition `P` and postcondition `Q`, written about the abstract state, is correct exactly when the Hoare triple

```
{{ RI and P(AF(this)) }}
the method's body
{{ RI and Q(AF(this)) }}
```

is valid and every call that begins with `P` true terminates, where `RI` is the representation invariant and `AF` is the abstraction function.

# Is size correct?

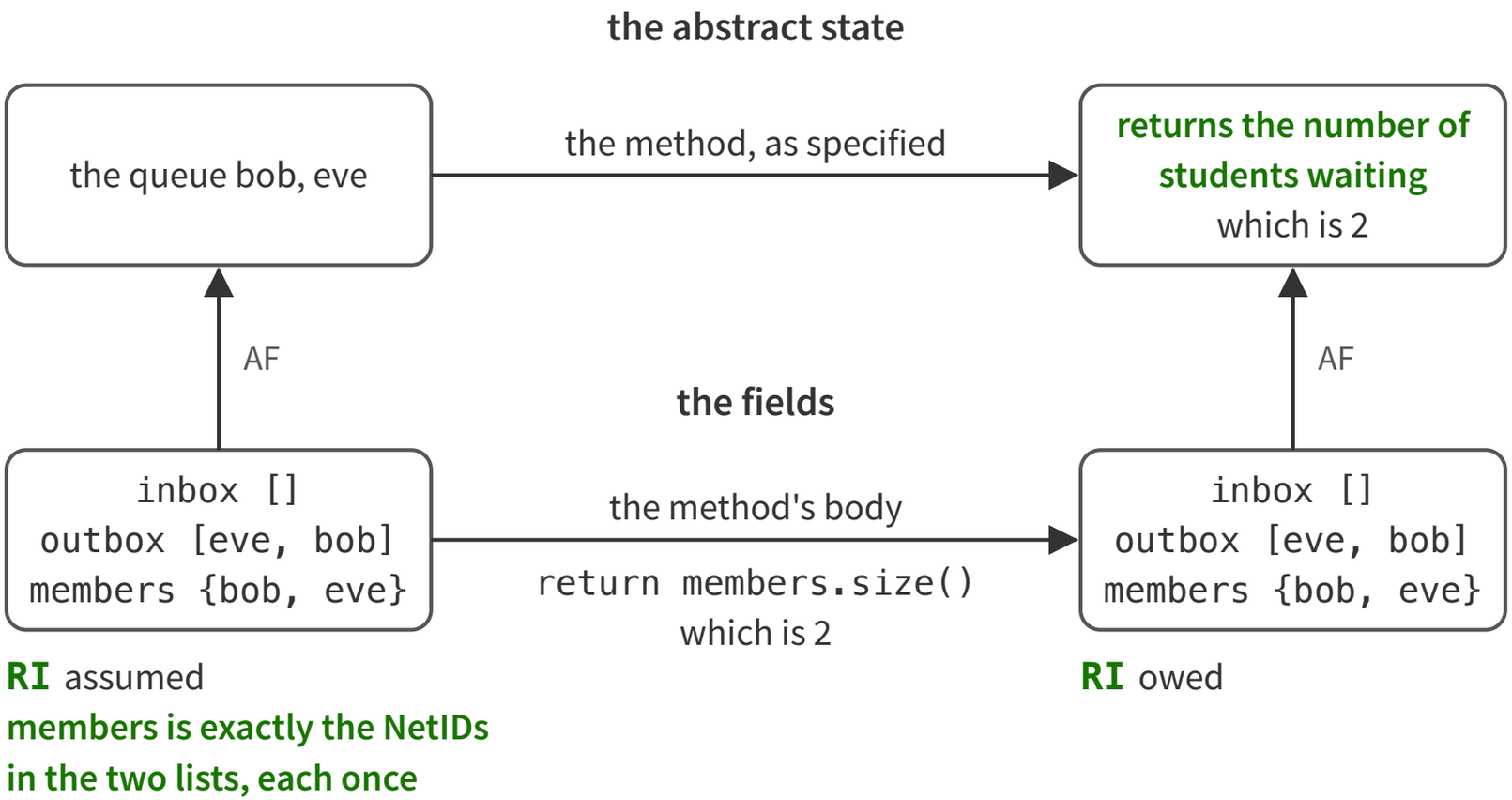
```
19  // RI: no NetID is in inbox or outbox twice, or in both, and
20  //     members holds exactly the NetIDs in inbox and outbox.
21  // AF(this) = the help queue of the students in outbox, last to
22  //     first, then the students in inbox, first to last.
```

```
52  public int size() {
        {{ RI }}
54      return members.size();
        {{ returns members.size() }}
        {{ owed by the specification: the number of students waiting in AF(this) }}
        {{ by the RI: members.size() = inbox.size() + outbox.size() }}
        {{ by the AF: that is the number of students in AF(this) }}
55  }
```

The gap: the body counts a set, the specification counts a queue

The RI and the AF close it; no field changed, so `size` is correct

# Is size correct?, drawn




- The forgetful `next`, by the rule: the argument stops at the missing line

# Half sheet, question 2 (3 minutes)

- **2.** Why call `checkRep` at the start of a method, not only at the end?
  - The last call that changed the object checked the RI at its end
- Hand it in at the door as you leave

# Copy out: a `waiting` that returns its own list

```
71  public List<String> waiting() {
72      checkRep();
73      return new ArrayList<String>(queue);
73      return queue;
74  }
```

```
49  void waitingIsACopy() {
50      HelpQueue queue = emptyQueue();
51      queue.join("ada");
52      List<String> copy = queue.waiting();
53      copy.add("eve");
```

The client's **copy** is the list in the field **queue**

The client changes the field, and no method of **`FastQueue`** ran

The check at **`waiting`**'s start throws: half sheet question 2

# Copy in: a queue that is not its own

```
26  /** @requires no NetID is in netids twice
27   *  @effects makes a queue of the students given, front first */
28  public FastQueue(List<String> netids) {
29      // A copy, so that the client cannot change the queue through
30      // the list it passed.
31      queue = new ArrayList<String>(netids);
31      queue = netids;
32      members = new HashSet<String>(netids);
33      checkRep();
34  }
```

The constructor keeps a copy of the client's list

A buggy version keeps the client's own list. Why is that a bug? Coming up

# Copy in: what goes wrong?

```
31      queue = new ArrayList<String>(netids);
31      queue = netids;
```

```
11  List<String> netids = new ArrayList<String>(List.of("ada"));
12  FastQueue queue = new FastQueue(netids);
13  netids.add("eve");
14  assertEquals(List.of("ada"), queue.waiting());
```

The queue's field **queue** is the client's list

The client changes the field, and no method of **FastQueue** ran

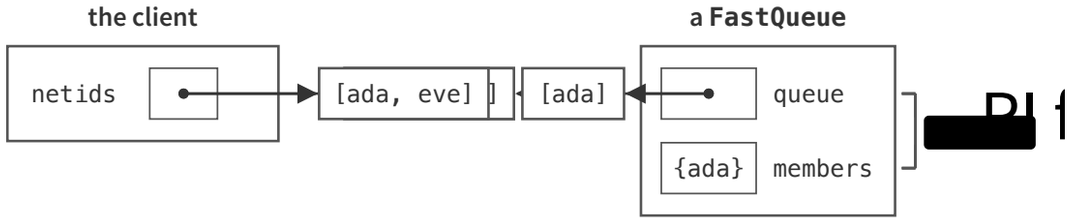
**waiting**'s check throws "members is not the queue"

# Copy in, drawn

```
31      queue = new ArrayList<String>(netids);
31      queue = netids;
```

```
31      queue = new ArrayList<String>(netids);
```

```
11      List<String> netids = new ArrayList<String>(List.of("ada"));
12      FastQueue queue = new FastQueue(netids);
13      netids.add("eve");
```




- `netids` and the field `queue` name one list: **aliasing**
- The client changes the fields, and no method of `FastQueue` ran
- The fix: **copy in**

# Representation exposure

## Definition: representation exposure

**Representation exposure** is a reference, held by code outside a class, to a changeable object among the class's fields. Through it, that code can change the fields without calling any method of the class, so the representation invariant and every specification can be broken by code the class does not control.
To prevent it, **copy in** a changeable object a client passes that becomes part of the fields, and **copy out** a changeable field that a method returns.

Copy in: line 31 of FastQueue.java. Copy out: its line 73

# Strings are safe to share

- No method of `String` can change a string
  - So a method that returns a `String` field exposes nothing
- An `int` holds a plain value: the client gets a copy of the number
- A `List`, a `Set`, an array: copy in, copy out
- **A class is correct** when every method is correct by the rule
  - and the class has no representation exposure

Java 25's documentation of `String`: "Strings are constant; their values cannot be changed after they are created"

# Coming up

- **Monday**: JavaScript in the browser, and `fetch`
- **Tuesday**: section, on abstract data types
- Homework 2: due tonight, 11:59pm
- Reading 3: due Friday, October 30
- Hand in your half sheet at the door