Abstract Data Types

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Homework 2: due Friday, October 23, 11:59pm
- Reading 3: out Thursday, October 22

The office hours queue

- **The queue**: the students waiting for help, in the order they joined
- **Its rules**: each student waits at most once, and joins at the back
- Each student is named by their NetID

ListQueue, a help queue on a list

```
 7  private List<String> queue = new ArrayList<String>();
 7  public List<String> queue = new ArrayList<String>();
 8
 9  public int size() {
10      return queue.size();
11  }
12
13  public void join(String netid) {
14      if (queue.contains(netid)) {
15          throw new IllegalArgumentException(netid + " is in the queue");
16      }
17      queue.add(netid);
18  }
```

join turns away a student who is already waiting

What if line 7 said public?

A client that takes a shortcut

```
 4  ListQueue queue = new ListQueue();
 5  queue.join("ada");
 6  queue.join("bob");
 7  // Skip join and change the list itself.
 8  queue.queue.add("ada");
 9  queue.queue.add(0, "eve");
10  System.out.println(queue.waiting());
11  System.out.println("helping " + queue.next());
```

queue.queue: the field named queue inside ListQueue

With the field public, what does it print?

A client that takes a shortcut, the run

```
$ javac Shortcut.java
$ java Shortcut
[eve, ada, bob, ada]
helping eve
```

ada is waiting twice, and eve cut in line: both rules broken

With line 7 of ListQueue.java made `public`

Who broke the rules?

- Not `ListQueue`: its `join` checks
- Lines 8 and 9 of `Shortcut`, which skip `join`
- With the field public, any line of any client can do the same
  - When the queue goes wrong, which line did it?

Make the field private

```
$ javac Shortcut.java
Shortcut.java:8: error: queue has private access in ListQueue
⋮
Shortcut.java:9: error: queue has private access in ListQueue
⋮
2 errors
```

Code outside `ListQueue` now gets only its methods

A client that keeps to the methods

```
4   ListQueue queue = new ListQueue();
5   // Students join at the back.
6   queue.join("ada");
7   queue.join("bob");
8   // The TA shows who is waiting, then helps the first.
9   System.out.println(queue.waiting());
10  System.out.println("helping " + queue.next());
```

What does it print?

A client that keeps to the methods, the run

```
$ javac OfficeHours.java
$ java OfficeHours
[ada, bob]
helping ada
```

Only `join` and `next` changed the queue, and both rules held

`ListQueue`'s operations, written down

```
3  /**
4   * The queue for office hours: the students waiting for help, in
5   * the order they joined. A student is named by their NetID, and a
6   * student is in the queue at most once.
7   */
8  public interface HelpQueue {
9      /** @return the number of students waiting */
10     int size();
11
12     /**
13      * @modifies this
14      * @effects adds the student at the back of the queue
15      * @throws IllegalArgumentException if the student is already in
16      *     the queue, and leaves the queue unchanged
17      */
18     void join(String netid);
```

`ListQueue`'s operations, continued

```
20      /**
21       * @modifies this
22       * @effects removes the student at the front of the queue
23       * @return the NetID of the student removed
24       * @throws NoSuchElementException if the queue is empty
25       */
26      String next();
27
28      /** @return the students waiting, front first, in a new list */
29      List<String> waiting();
30  }
```

No fields: every specification is about the students waiting

`ListQueue`'s line 6 says `implements HelpQueue`

Abstract data type

Definition: abstract data type (ADT)

An **abstract data type** is a type defined by specs for its operations.

The **abstract state** is how clients should think about the object.

`HelpQueue`'s abstract state: the students waiting, in the order they joined

Representation

Definition: representation

The **representation** of an ADT is how it is implemented.

This includes the fields of the class and the code of its methods.

The specification never mentions the representation.

`ListQueue`'s representation: one list of the NetIDs waiting, front first

Who keeps this rule?

```
4   * The queue for office hours: the students waiting for help, in
5   * the order they joined. A student is named by their NetID, and a
6   * student is in the queue at most once.
```

A student is in the queue at most once

- Only `join` and `next` can change the queue
  - `join` turns away a student who is already waiting
  - `next` only removes, and `waiting` hands out a new list

Abstract invariant

Definition: abstract invariant

An **abstract invariant** of an ADT is a claim about the abstract state that is true whenever no method of the type is running.

- **Ours**: a student is in the queue at most once

Half sheet, question 1 (3 minutes)

- **1.** `ListQueue`'s `join` walks the list for a student already waiting
  - How could we make this faster?

A help queue on a chain of nodes

```
 6  public class LinkedQueue implements HelpQueue {
 7      /** One student waiting, and the node of the student after them. */
 8      private static class Node {
 9          String netid;
10          Node next;  // null for the last node in the chain
11
12          Node(String netid) {
13              this.netid = netid;
14          }
15      }
16
17      // The first and last nodes of the chain, both null when nobody
18      // is waiting.
19      private Node front;
20      private Node back;
```

Node is a class declared inside LinkedQueue, and private to it

OfficeHours on a LinkedQueue

```
4   ListQueue queue = new ListQueue();
4   HelpQueue queue = new LinkedQueue();
```

```
$ javac OfficeHours.java
$ java OfficeHours
[ada, bob]
helping ada
```

What does it print now?

The same two lines as on a `ListQueue`

Same or different?

- `OfficeHours` takes either class by changing one line
- **Different**: a list inside one, a chain of nodes inside the other
  - A method body from either would not compile in the other
- **The same**: both are a `HelpQueue`
  - The same four methods, with the same specifications
  - Code that relies only on those works with either

`List` is an ADT too

- `ArrayList` keeps an array, `LinkedList` a chain of nodes
- Both are a `List`: the same methods, the same specifications
- Code that declares a `List` takes either by changing one line

`List`'s abstract state: its elements, in order

Barbara Liskov

- **1968**: among the first women in the US with a CS PhD
- **1974**: *Programming with Abstract Data Types*, with Stephen Zilles
  - Clients get the operations, never the data
- **At MIT**: CLU, a language built around data abstraction
- **2008**: the Turing Award, its citation naming data abstraction first

Sources: ACM's A.M. Turing Award page for Barbara Liskov; Liskov and Zilles, SIGPLAN Notices 9(4), 1974

Testing the queue, not a class

```
 9  public abstract class HelpQueueTest {
10      // A subclass says which implementation to test.
11      abstract HelpQueue emptyQueue();
12
13      @Test
14      void studentsAreServedInTheOrderTheyJoined() {
15          HelpQueue queue = emptyQueue();
```

Each test gets its queue from **`emptyQueue`**

`abstract`: no body here, so a subclass must give it one

Specification-based tests: they pass on every correct implementation

One subclass per implementation

```
1  public class ListQueueSpecTest extends HelpQueueTest {
2      HelpQueue emptyQueue() {
3          return new ListQueue();
4      }
5  }
```

```
1  public class LinkedQueueSpecTest extends HelpQueueTest {
2      HelpQueue emptyQueue() {
3          return new LinkedQueue();
4      }
5  }
```

The line that picks the implementation

Five tests each: how many of the ten pass?

Both subclasses, the run

```
$ java -jar junit.jar execute --disable-banner \
    --class-path . --select-class ListQueueSpecTest \
    --select-class LinkedQueueSpecTest
┊
├─ ListQueueSpecTest ✔
┊
└─ LinkedQueueSpecTest ✔
┊
[         10 tests successful      ]
[          0 tests failed          ]
```

None of the five can tell the two classes apart

A test of ListQueue

```
10  @Test
11  void joinTwiceSaysWho() {
12      HelpQueue queue = new ListQueue();
13      queue.join("ada");
14      IllegalArgumentException e = assertThrows(
15          IllegalArgumentException.class, () -> queue.join("ada"));
16      assertEquals("ada is in the queue", e.getMessage());
17  }
```

`() -> queue.join("ada")`: the call, handed to `assertThrows` to make

It checks `join`'s message, word for word

Change `new ListQueue()` to `new LinkedQueue()`: will the test pass?

A test of ListQueue, on a LinkedQueue

```
$ java -jar junit.jar execute --disable-banner \
    --class-path . --select-class ListQueueImplTest
⋮
│     ├─ joinTwiceSaysWho() ✘
        expected: <ada is in the queue>
        but was: <ada is already waiting>
⋮
```

`LinkedQueue` is right: **`HelpQueue`** never promised those words

Implementation-based: the test relied on more than the specification

In ListQueueImplTest.java, `new ListQueue()` changed to `new LinkedQueue()`

Take `join`'s check out

```
13  public void join(String netid) {
14      if (queue.contains(netid)) {
15          throw new IllegalArgumentException(netid + " is in the queue");
16      }
17      queue.add(netid);
18  }
```

Which of the five tests fail?

Take `join`'s check out, the run

```
$ java -jar junit.jar execute --disable-banner \
    --class-path . --select-class ListQueueSpecTest
⋮
│      ├─ nobodyIsInTheQueueTwice() ✘
         Expected java.lang.IllegalArgumentException to be thrown,
         but nothing was thrown.
⋮
[         4 tests successful      ]
[         1 tests failed          ]
```

The other four never join a student who is already waiting

Half sheet, question 2 (3 minutes)

- **2.** Suppose `ListQueue`'s `waiting` returned its field `queue`
  - How could a client put ada in the queue twice?
- Hand it in at the door as you leave

Suppose `waiting` said less

```
28   /** @return the students waiting, front first, in a new list */
```

- Suppose it said only: *returns the students waiting*

What could a client take that to mean?

- A new list: the students waiting at the moment of the call
- The queue's own list, which changes as the queue changes
- A list the client may read, but must not change or keep

`HelpQueue` promises the first: *in a new list*

The queue's own list

```
27  public List<String> waiting() {
28      // A copy, so that a client cannot change the queue through it.
29      return new ArrayList<String>(queue);
29      return queue;
30  }
```

Question 2: how did you get ada in twice?

`waitingIsACopy`, on the queue's own list

```
48  @Test
49  void waitingIsACopy() {
50      HelpQueue queue = emptyQueue();
51      queue.join("ada");
52      List<String> copy = queue.waiting();
53      copy.add("eve");
54      assertEquals(List.of("ada"), queue.waiting());
55  }
```

One of the five tests changes the list `waiting` gave it

The queue's own list, the run

```
$ java -jar junit.jar execute --disable-banner \
    --class-path . --select-class ListQueueSpecTest
⋮
│      ├─ nextOnAnEmptyQueueThrows() ✔
│      ├─ waitingIsACopy() ✘ expected: <[ada]> but was: <[ada, eve]>
│      ├─ nobodyIsInTheQueueTwice() ✔
│      ├─ aStudentWhoLeftMayComeBack() ✔
│      └─ studentsAreServedInTheOrderTheyJoined() ✔
⋮
```

eve got in without `join`

The public field's problems, back again

Why `waiting` makes a copy

```
27   public List<String> waiting() {
28       // A copy, so that a client cannot change the queue through it.
29       return new ArrayList<String>(queue);
30   }
```

- Making the copy walks the whole queue on each call
- In return, only `join` and `next` can change the queue

Coming up

- **Friday**: implementing an ADT
- Homework 2: due Friday, October 23, 11:59pm
- Reading 3: out Thursday, October 22
- Hand in your half sheet at the door