Exam Reminders

Prelim 1 is Tonight!

Early Exam: 5:30-7:00, Olin Hall 155
Main Exam: 7:30-9:00, Olin Hall 255 (Last Names A-J) and 155 (Last Names K-Z)

Bring your **Cornell ID Card** and a couple **writing utensils** (pencils, erasers, pens)
Exam is closed-book

More information and review materials linked on website / Ed

No OHs tomorrow because of exam grading.

You've learned a lot so far! Time to show it off!

Lecture 14: Iterating over Data Structures

CS 2110

October 9, 2025

Today's Learning Outcomes

59. Describe the use cases for the Iterable and Iterator interfaces.

60. Implement Iterators for a given data structure class and use iterators as a client.

61. Describe the iteratee pattern and how it differs from using an iterator.

62. Identify and define functional interfaces and implement them using lambda expressions.

frequencyOf() on Lists

Modifies our earlier frequencyOf() definition in two ways:

1. Support generic collection
   * new syntax to introduce generic type in signature of a method.
2. Takes in CS2110List instead of array.
   * interface type

```
/** Returns the # of occurrences of `key` in `list`. */
static <T> int frequencyOf(T key, CS2110List<T> list) {
    int i = 0; // next index of `list` to check
    int count = 0;
    /* Loop inv: `count` = # of occurrences of `key` in
     * the first `i` elements of `list`. */
    while (i < list.size()) {
        if (list.get(i).equals(key)) {
            count++;
        }
        i++;
    }
    return count;
}
```

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

What is the runtime complexity of `frequencyOf()` for our `CS2110List` implementations?

N = list.size()

```
while (i < list.size()) {
    if (list.get(i).equals(key)) { count++; }
    i++;
}
```

O(N) iterations

O(1) for DAL, O(N) for SLL

DAL = DynamicArrayList
SLL = SinglyLinkedList

DAL: $O(N)$	SLL: $O(N)$	**(A)**
DAL: $O(N)$	SLL: $O(N^2)$	**(B)**
DAL: $O(N^2)$	SLL: $O(N)$	**(C)**
DAL: $O(N^2)$	SLL: $O(N^2)$	**(D)**

Motivating Iterators

For data structures that don't offer a random access guarantee, looping over their elements can be expensive

SLL: On the client size, we "lose our place" after each get() call, need to rescan from head each time

We'd like to provide a client an easy/fast way to visit all elements of a collection.

Iterators help to oversee this iteration by keeping track of next element to visit.

The Iterator Interface

An Iterator models a separate object from a collection that yields (returns) each object in the collection once during its lifetime.

Two (Required) Methods:

1. boolean hasNext(): Is there an element that hasn't yet been yielded?

2. T next(): Return the next unyielded element.

T: generic element type

* Often modeled as inner classes of collection class. (offers good access and encapsulation)

Coding Demo: List Iterators

The `Iterable` Interface

The collection class (itself) implements the Iterable interface to report that it can return an Iterator over its elements.

One Method:

`Iterator<T> iterator()` : Returns a new iterator over this collection

* typically, body just returns a constructor call to its inner iterator class.

Coding Demo: Making Lists `Iterable`

Iterators from the Client Side

```
/** Returns the # of occurrences of `key` in `list`. */
static <T> int frequencyOf(T key, CS2110List<T> list)
{
  int count = 0;
  Iterator<T> it = list.iterator();
  /* Loop inv: count = # of 'key's that 'it' has yielded. */
  while (it.hasNext()) {
    T elem = it.next();
    if (elem.equals(key)) { count++; }
  }
  return count;
}
```

For both DAL and SLL, iterators' hasNext() and next() methods run in O(1) time.

⇒ frequencyOf() always has O(N) runtime

Enhanced For-Loops

Loops using Iterator<T>s almost always follow same pattern!

```
Iterator<T> it = _____.iterator();
while (it.hasNext()) {
  T elem = it.next();
  // do something with elem
}
```

=

```
for (T elem : _____) {
  // do something
     with elem
}
```

any Iterable

"for each elem in ____"

Java offers special "enhanced for-loop" syntax that automatically does this behind the scenes.

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `22333`

Suppose `list` contains `[1,-2,-3,4,-5]` at the start of this code block. What will be its `size` at the end?

```
for (Integer i : list) {
    if (i < 0) {  list.delete(i); }
}
```

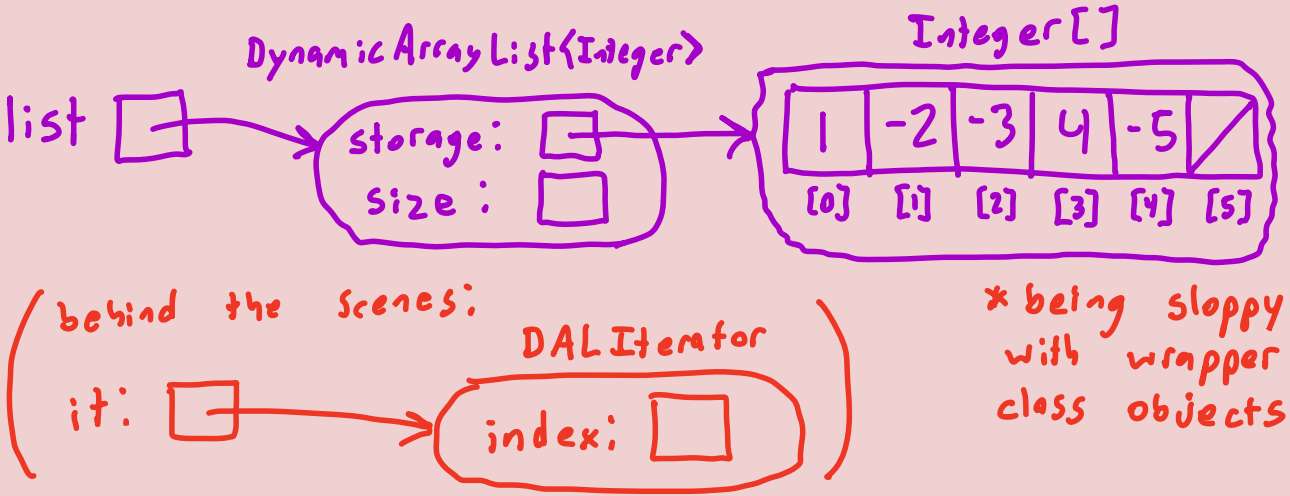
2	(A)
3	(B)
4	(C)
5	(D)

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

Suppose `list` contains `[1,-2,-3,4,-5]` at the start of this code block. What will be its `size` at the end?

```
for (Integer i : list) {
    if (i < 0) {  list.delete(i); }
}
```



2	(A)
3	(B)
4	(C)
5	(D)

Concurrent Modification

An iterator only guarantees to yield each element of a collection exactly once as long as the collection is not modified during its lifetime.

* "Concurrent" modifications can mess with iterator's state representation / invariants (and cause unexpected behaviors)

Don't modify a collection within an enhanced for-loop over its elements!

(Super common source of bugs!)

Iteration with Modification

```
/** Adds 1 to each Integer in `list`. */
public static void incrementAll(CS2110List<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        list.set(i, list.get(i) + 1);
    }
}
```

} O(N) work per iteration for SLL ⇒ $O(N^2)$ runtime

Will iterators help here?

- No, Iterators yield the elements themselves, which doesn't provide a way to modify the collection.
  (we'd need a reference to the Node object to re-assign its data field, but that would break encapsulation)

New idea: Ask collection to do modifications for us.

The Iteratee Pattern

In the iterqtee pattern, the client specifies some behaviors that a collection will perform on each of its elements "behind the scenes" during an iteration.

Ex. "Hey list, can you please add 1 to each of your elements."

How do we "pass" this behavior?

Functional Interfaces

A functional interface is any interface with exactly one* method.

- Good practice to annotate with @FunctionalInterface

Idea: Lets us create objects that represent behaviors (call the one method I've promised to define)

In the iteratee pattern, we typically parameterize the method that will perform the iteration on a functional interface type.

Coding Demo: Iteratee Pattern

Big Ideas:

1) Declare a functional interface
2) Write collection method that iterates over elements and calls functional interface method on each
3) Write a class implementing the functional interface that encapsulates the desired behavior

Lambda Expressions

A more convenient syntax for instantiating a functional interface.

(parameters) -> { method body }

(Java can use this to write class for us behind the scenes.)

```
list.transformAll((Integer x) -> { return x+1; })
```

Even simpler syntax:

Parameter types can be inferred

```
list.transformAll(x -> { return x+1; })
```

Return can be inferred

```
list.transformAll(x -> x+1)
```

Lambda Expressions: Behind the Scenes

`list.transformAll(x -> x+1)` // How does this work?

1. Java looks at `transformAll()` signature and sees `Transformation<T>` is parameter type.
2. Generic type `T` was `Integer` from list declaration.
3. Since `Transformation` is a functional interface, a class implementing it is created behind the scenes.
4. The class must contain a `transform()` method that takes in an `Integer` parameter `x`.
5. We can auto-unbox `x`, add 1 and auto-box the sum into an `Integer` to return from `transform()`.
6. Java calls constructor of new class and passes reference into `transformAll()`.

Exercise: Re-write censor()

Our previous `censor()` implementation had an $O(N^2)$ runtime.

```
/** Replaces all instances of the given `word` with "****" in these `lyrics`. */
static void censor(CS2110List<String> lyrics, String word) {
   while(lyrics.contains(word)) {
      lyrics.set(lyrics.indexOf(word), "****");
   }
}
```

Use a call to `transformAll()` with a lambda expression to achieve an $O(N)$ runtime.

```
/** Replaces all instances of the given `word` with "****" in these `lyrics`. */
static void censor(CS2110List<String> lyrics, String word) {
 transformAll( s -> s.equals(word) ? "****" : s );
}
```

Main Takeaway: The Power of Interfaces

Today, we saw two places where interfaces enabled powerful new Java language features.

1. Enhanced For-Loops

   Iterable objects can always provide Iterators with hasNext() and next() methods that let us loop over a collection

2. Lambda Expressions

   Since functional interfaces include one method, using these interface types as parameters lets clients describe actions with simpler syntax; Java infers the rest.