



Lecture 14: Iterating over Data Structures

CS 2110

October 8, 2026

Today's Learning Outcomes

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

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

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

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

Motivating Iterators

Looping over data structures that don't offer a random access guarantee can be expensive.

The Iterator Interface

An `Iterator<T>` models a separate object from a collection that yields (returns) each element of the collection once during its lifetime.

Two (Required) Methods:



Coding Demo: List Iterators



The Iterable Interface

A collection class (itself) implements the `Iterable<T>` interface to report that it can return an `Iterator` over its elements.

One (Required) Method:



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) {
}
```

Enhanced For-Loops

Loops using `Iterator<T>`s almost always following the same pattern:

Java offers a special "enhanced for loop" syntax that automatically takes care of this behind the scenes.

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

Concurrent Modification

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

Updates During Iteration

What is the runtime of this method?

```
/** 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);  
    }  
}
```

Will iterators help here?

The Iteratee Pattern

In the **iteratee pattern**, the client specifies a behavior for a collection to perform on each of its elements "behind the scenes" during an iteration.

Functional Interfaces

= an interface with exactly one* method.

It's a good programming practice to annotate these with `@FunctionalInterface`.



Coding Demo: Iteratee Pattern



Big Ideas:

1. Declare a functional interface type (or use one provided by Java)
2. Write a 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.

Lambda Expressions: Behind the Scenes

How does the call `list.transformAll(x -> x + 1)` actually work?

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 we can use to 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.