What’s a good list interface

We now have two different kinds of lists: lazy and eager. (We’ll talk about lazy lists on Friday.)

We want clients to not notice, not care.

Solution? A standard list interface that both can share.
public interface IList<T> {
    /**
     * return whether the list is empty or not
     */
    boolean empty();

    /**
     * Return the first element of the list. Will throw an EmptyListException
     * if the list is empty.
     */
    T head();

    /**
     * Return a list of everything but the first element of the current list. Calling
tail() on an empty list is defined to return an empty list.
     */
    IList<T> tail();

    /**
     * return a new list with this element in front of the current list
     */
    IList<T> add(T t);

    /**
     * return a list with this function applied to each of the elements of the current list
     */
    <Q> IList<Q> map(Function<T, Q> f);

    /**
     * return at most the first n elements of the list, could be fewer
     */
    IList<T> limit(int n);

    ...
}
public interface IList<T> {

    /**
     * return whether the list is empty or not
     */
    boolean empty();

    /**
     * Return the first element of the list. Will throw an EmptyListException
     * if the list is empty.
     */
    T head();

    /**
     * Return a list of everything but the first element of the current list. Calling
     * tail() on an empty list is defined to return an empty list.
     */
    IList<T> tail();

    /**
     * return a new list with this element in front of the current list
     */
    IList<T> add(T t);

    /**
     * return a list with this function applied to each of the elements of the current list
     */
    <Q> IList<Q> map(Function<T, Q> f);

    /**
     * return at most the first n elements of the list, could be fewer
     */
    IList<T> limit(int n);

    ...
}
public interface IList<T> {
    /**
     * return whether the list is empty or not
     */
    boolean empty();

    /**
     * Return the first element of the list. Will throw an EmptyListException
     * if the list is empty.
     */
    T head();

    /**
     * Return a list of everything but the first element of the current list. Calling
     * tail() on an empty list is defined to return an empty list.
     */
    IList<T> tail();

    /**
     * return a new list with this element in front of the current list
     */
    IList<T> add(T t);

    /**
     * return a list with this function applied to each of the elements of the current list
     */
    <Q> IList<Q> map(Function<T, Q> f);

    /**
     * return at most the first n elements of the list, could be fewer
     */
    IList<T> limit(int n);
    ...
}
public interface IList<T> {
    /**
     * return whether the list is empty or not
     */
    boolean empty();

    /**
     * Return the first element of the list. Will throw an EmptyListException
     * if the list is empty.
     */
    T head();

    /**
     * Return a list of everything but the first element of the current list. Calling
     * tail() on an empty list is defined to return an empty list.
     */
    IList<T> tail();

    /**
     * return a new list with this element in front of the current list
     */
    IList<T> add(T t);

    /**
     * return a list with this function applied to each of the elements of the current list
     */
    <Q> IList<Q> map(Function<T, Q> f);

    /**
     * return at most the first n elements of the list, could be fewer
     */
    IList<T> limit(int n);

    ...
}
public interface IList<T> {

    /**
     * return whether the list is empty or not
     */
    boolean empty();

    /**
     * Return the first element of the list. Will throw an EmptyListException
     * if the list is empty.
     */
    T head();

    /**
     * Return a list of everything but the first element of the current list. Calling
     * tail() on an empty list is defined to return an empty list.
     */
    IList<T> tail();

    /**
     * return a new list with this element in front of the current list
     */
    IList<T> add(T t);

    /**
     * return a list with this function applied to each of the elements of the current list
     */
    <Q> IList<Q> map(Function<T, Q> f);

    /**
     * return at most the first n elements of the list, could be fewer
     */
    IList<T> limit(int n);

    ...

Notably absent: constructors, member variables.
public class List<T> implements IList<T> {
    private final T contents;
    private final IList<T> tailList;

    public boolean empty() { return false; }

    public T head() { return contents; }

    public IList<T> tail() {
        return tailList;
    }

    ...
}

Implementation (eager list)

```java
public class List<T> implements IList<T> {
    private final T contents;
    private final IList<T> tailList;

    public boolean empty() { return false; }

    public T head() { return contents; }

    public IList<T> tail() {
        return tailList;
    }

    ...
}
```

Important: IList, not List
Cool new list feature: List.of()

Sick of typing this?
IList<String> emptyList = List.Empty.create();
IList<String> list0 = emptyList.add("Charlie").add("Bob").add("Alice");

How about this?
IList<String> list1 = List.of("Alice", "Bob", "Charlie");

You can put as many things there as you want.
(We’ll talk about how it works on Friday.)
List vs. IList

Generally, you’ll declare your variables to be IList<Something>

You’ll only ever say “List” if:
You’re writing code inside the List class and calling the constructor
You’re building an empty list with List.Empty.create()
You’re using a convenience function like List.of() or List.rangeInt()

Otherwise, you’re just dealing with IList (the interface type)
rather than List (the concrete type)
Interfaces can have code in them!

```java
public interface IList<T> {
    /**
     * Return the length of the list.
     */
    default int length() {
        // Foldl here ignores the elements in the list,
        // and just uses the folding function to increment
        // the counter each time.
        return foldl((count, elem)->count+1, 0);
    }
    ...
}
```

“Default” methods can use all the other methods in the interface
But they can’t see member variables, can’t call constructors, etc.
(New feature in Java8. Similar features in other languages: mixins, traits.)
“Diamond” dependencies

If you’ve ever heard a C++ hacker complain about multiple inheritance...

```java
interface A {
    default void foo() { System.out.println("A"); }
}
interface B extends A {
    default void foo() { System.out.println("B"); }
}
interface C extends A { ... } // no mention of foo()

class D implements B, C { ...
    foo(); // Does this print A, B, or get a compiler error?
}
```
“Diamond” dependencies

If you’ve ever heard a C++ hacker complain about multiple inheritance...

interface A {
    default void foo() { System.out.println("A"); }
}

interface B extends A {
    default void foo() { System.out.println("B"); }
}

interface C extends A { ... } // no mention of foo()

class D implements B, C { ...
    foo(); // Does this print A, B, or get a compiler error?
}

The rules:
If two interfaces provide the same method directly? Conflict.
Else, closest definition wins.

Disambiguation:
Name the one you want.
A.super.foo()
B.super.foo()
When should I put what code where?

First, write an interface with no code, only method signatures
Make sure it makes sense. Write a class that implements it.
Clients will see a concrete constructor or static helper method, but otherwise should only use the interface.

Remember zen programming rule #1: don’t repeat yourself
If you would have identical code in a second class, “promote” it to a default method in the interface.

“Default” code from interfaces vs. “overriding” code from class extension
Interface default methods written purely in terms of other interface methods.
Bonus: a class can implement multiple interfaces (vs. extend only one class).
Cool features in List / IList: nth

You can pretend that a list is like an array and fetch something.

```csharp
IList<String> names = List.of("Alice", "Bob", "Charlie", "Dan");
assertEquals("Alice", names.nth(0));
assertEquals("Bob", names.nth(1));
assertEquals("Charlie", names.nth(2));
```

This costs $O(n)$, so if you’re doing it a lot, your performance will suffer.
Cool features in List / IList: join

You can convert everything to a string (if it isn’t already) and join them with any character you want.

```csharp
IList<String> names = List.of("Alice", "Bob", "Charlie", "Dan");
assertEquals("Alice Bob Charlie Dan", names.join(" "));
```
Cool features in List / IList: sublist

You can extract a range of a list

```csharp
IList<string> names = List.of("Alice", "Bob", "Charlie", "Dan");
assertEquals("Alice Bob", names.sublist(0, 1).join(" "));
assertEquals("Charlie Dan", names.sublist(2, 3).join(" "));
```
Cool features in List / IList: rangeInt

You built this last week; it’s incredibly useful
Example: combine with map and you’ve got something like a loop counter. Here, we use that to get a list of characters using String’s charAt method.

String testString = "Hello Rice!";

IList<Character> testChars =
    List.rangeInt(0, testString.length()-1)
    .map((i)->testString.charAt(i));

assertEquals("H,e,l,l,o, ,R,i,c,e,!", testChars.join("",""));
Dealing with lists of lists

Here’s an example similar to what you’ll do in the permutation lab

Sublist returns a list. So each map returns a list. So we get a list of those lists.

```csharp
IList<Integer> numbers = List.of(1, 5, 3, 4);
IList<IList<Integer>> sublists =
    List.rangeInt(0, numbers.length()-1)
    .map((i)->numbers.sublist(0, i));

String flattenedList =
    "[" +
    sublists
    .map((sl)->sl.join(","))
    .join("] [")
    + "]";

assertEquals("[1] [1,5] [1,5,3] [1,5,3,4]", flattenedList);
```
Dealing with lists of lists

Here's an example similar to what you’ll do in the permutation lab

Sublist returns a list. So each map returns a list. So we get a list of those lists.

```csharp
IList<Integer> numbers = List.of(1, 5, 3, 4);
IList<IList<Integer>> sublists =
    List.rangeInt(0, numbers.length() - 1)
    .map((i) => numbers.sublist(0, i));

String flattenedList =
    "[" +
    sublists
    .map((sl) => sl.join(","))
    .join("] [")
+ "]";

assertEquals("[1] [1,5] [1,5,3] [1,5,3,4]", flattenedList);
```
Dealing with lists of lists

Here’s an example similar to what you’ll do in the permutation lab

Sublist returns a list. So each map returns a list. So we get a list of those lists.

```csharp
IList<Integer> numbers = List.of(1,5,3,4);
IList<IList<Integer>> sublists =
    List.rangeInt(0, numbers.length()-1)
    .map((i)->numbers.sublist(0,i));

String flattenedList =
    "[" +
    sublists
    .map((sl)->sl.join("","))
    .join("] [")
+ "]";

assertEquals("[1] [1,5] [1,5,3] [1,5,3,4]", flattenedList);
```

Each inner list becomes a string, separated by a comma. This map returns an IList<String>

Now we’re joining those strings together with brackets.
Dealing with lists of lists

Here's an example similar to what you'll do in the **permutation** lab

Sublist returns a list. So each map returns a list. So we get a list of those lists.

```csharp
IList<Integer> numbers = List.of(1, 5, 3, 4);
IList<IList<Integer>> sublists =
    List.rangeInt(0, numbers.length()-1)
    .map((i)->numbers.sublist(0, i));

String flattenedList =
    "[" +
    sublists
    .map((sl)->sl.join(","))
    .join("] [")
    + "]";

assertEquals("[1] [1,5] [1,5,3] [1,5,3,4]", flattenedList);
```

Style note: if you've got complicated pipelines like these, put each stage on its own line. Much easier to read!
Dealing with lists of lists: flatmap

What if each map returns a list and we want to concatenate them?
No problem. flatmap has us covered. Note: your lambda must return a list!

```csharp
IList<Integer> numbers = List.of(1, 5, 3, 4);
IList<Integer> list =
    List.rangeInt(0, numbers.length() - 1)
    .flatMap((i) -> numbers.sublist(0, i));

assertEquals("1, 1, 5, 1, 5, 3, 1, 5, 3, 4", list.join(",");
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```java
IList<IList<Integer>> lists = subsequences(List.of(9, 3, 20, 4));

String flattenedList = 
    "[" + 
    lists 
        .map(l -> l.join(",")) 
        .join("] [" 
    + "]";

assertEquals( 
    "[9] [9,3] [3] [9,3,20] [3,20] [20] [9,3,20,4] [3,20,4] [20,4] [4]",
    flattenedList);
```

Let's implement it!
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
IList<IList<Integer>> lists = subsequences(List.of(9, 3, 20, 4));

String flattenedList =
    "[" +
    lists
    .map((l) -> l.join("","))
    .join("] [")
    + "]";

assertEquals(
    "[9] [9,3] [3] [9,3,20] [3,20] [20] [9,3,20,4] [3,20,4] [20,4] [4]",
    flattenedList);
```

Let's implement it!
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
public static <T> IList<IList<T>> subsequences(IList<T> list) {
    if(list.empty()) return List.Empty.create();
    int length = list.length();
    IList<Integer> listEnds =
        List.rangeInt(0, length-1);
    IList<IList<T>> prefixes =
        listEnds.map((end)->list.sublist(0, end));
    IList<IList<T>> results = prefixes.flatmap((prefix) -> {
        int lastIndex = prefix.length() - 1;
        return List.rangeInt(0, lastIndex)
            .map((start)->list.sublist(start, lastIndex));
    });
    return results;
}
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
public static <T> IList<IList<T>> subsequences(IList<T> list) {
    if(list.empty()) return List.Empty.create();
    int length = list.length();
    IList<Integer> listEnds =
        List.rangeInt(0, length-1);
    IList<IList<T>> prefixes =
        listEnds.map((end)->list.sublist(0, end));
    IList<IList<T>> results = prefixes.flatmap((prefix) -> {
        int lastIndex = prefix.length()-1;
        return List.rangeInt(0, lastIndex)
            .map((start)->list.sublist(start, lastIndex));
    });
    return results;
}
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
public static <T> IList<IList<T>> subsequences(IList<T> list) {
    if (list.empty()) return List.Empty.create();
    int length = list.length();
    IList<Integer> listEnds = List.rangeInt(0, length-1);
    IList<IList<T>> prefixes = listEnds.map((end) => list.sublist(0, end));
    IList<IList<T>> results = prefixes.flatmap((prefix) => {
        int lastIndex = prefix.length()-1;
        return List.rangeInt(0, lastIndex)
            .map((start) => list.sublist(start, lastIndex));
    });
    return results;
}
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
public static <T> IList<IList<T>> subsequences(IList<T> list) {
    if(list.empty()) return List.Empty.create();
    int length = list.length();
    IList<Integer> listEnds =
        List.rangeInt(0, length-1);
    IList<IList<T>> prefixes =
        listEnds.map((end)->list.sublist(0, end));
    IList<IList<T>> results = prefixes.flatmap((prefix) -> {
        int lastIndex = prefix.length()-1;
        return List.rangeInt(0, lastIndex)
            .map((start)->list.sublist(start, lastIndex));
    });
    return results;
}
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
public static <T> IList<IList<T>> subsequences(IList<T> list) {
    if (list.isEmpty()) return List.Empty.create();
    int length = list.length();
    IList<Integer> listEnds = List.rangeInt(0, length-1);
    IList<IList<T>> prefixes = listEnds.map((end) => list.sublist(0, end));
    IList<IList<T>> results = prefixesflatMap((prefix) => {
        int lastIndex = prefix.length()-1;
        return List.rangeInt(0, lastIndex)
            .map((start) => list.sublist(start, lastIndex))
    });
    return results;
}
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
IList<IList<T>> results = prefixes.flatmap((prefix) -> {
    int lastIndex = prefix.length() - 1;
    return List.rangeInt(0, lastIndex)
        .map((start) -> list.sublist(start, lastIndex));
});
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

\[
[9] [9, 3] [9, 3, 20] [9, 3, 20, 4] 
\]

\[
9, 3, 20 
\]

\[
\text{IList}\text{<IList<T>}} \text{ results} = \text{prefixes.flatmap((prefix) -> { 
    \text{int lastIndex} = \text{prefix.length()} - 1; 
    \text{return List.rangeInt}(0, \text{lastIndex}) 
        .map((start)->list.sublist(start, lastIndex)); 
}}; 
\]
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
IList<IList<T>> results = prefixes.flatMap((prefix) -> {
    int lastIndex = prefix.length() - 1;
    return List.rangeInt(0, lastIndex)
        .map((start) -> list.subList(start, lastIndex));
});
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```
IList<IList<T>> results = prefixes.flatmap((prefix) -> {
    int lastIndex = prefix.length()-1;
    return List.rangeInt(0, lastIndex)
        .map((start)->list.sublist(start, lastIndex));
});
```
Putting it all together: subsequences

Say we want a list of all the subsequences of a list

```csharp
IList<IList<T>> results = prefixes.flatmap((prefix) -> {
    int lastIndex = prefix.length() - 1;
    return List.rangeInt(0, lastIndex)
        .map((start) -> list.sublist(start, lastIndex));
});
```

[9] [9, 3] [9, 3, 20] [9, 3, 20, 4]
9, 3, 20

[9, 3, 20] [3, 20] [20]
Putting it all together: subsequences

map would give us a list with four lists in it; flatmap gives us a list with 10 lists

<table>
<thead>
<tr>
<th>Prefix (input to the lambda)</th>
<th>Lambda’s output (list of suffixes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>[9]</td>
</tr>
<tr>
<td>9, 3</td>
<td>[9, 3] [3]</td>
</tr>
<tr>
<td>9, 3, 20</td>
<td>[9, 3, 20] [3, 20] [20]</td>
</tr>
<tr>
<td>9, 3, 20, 4</td>
<td>[9, 3, 20, 4] [3, 20, 4] [20, 4] [4]</td>
</tr>
</tbody>
</table>
Live coding

Revisit week 2 project / labs

Play with new IList / List code features