Tutorial 1
Assignment 1, lib280, and you!

Mark G. Eramian; G. Scott Johnston

University of Saskatchewan

The Plan

1. Introduction to List classes in lib280.

2. Examples

   - How to use a `LinkedList280` and `LinkedIterator280`.

3. Go over Assignment 1 so that everything is clear.

   - Review the tasks to be completed on Assignment 1.

   - Run through the tutorial on importing lib280 into IntelliJ and setting up a project to use it.

Part I

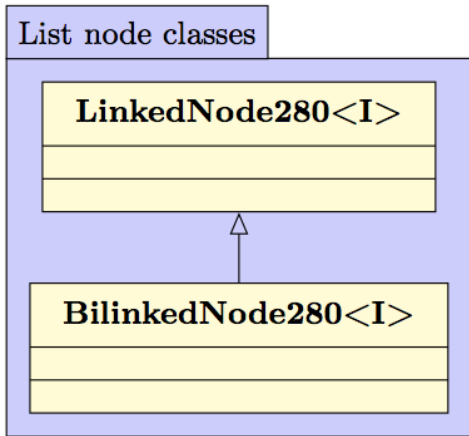## Introduction to List classes in lib280.

Singly- and Doubly- Linked List Classes in `lib280`
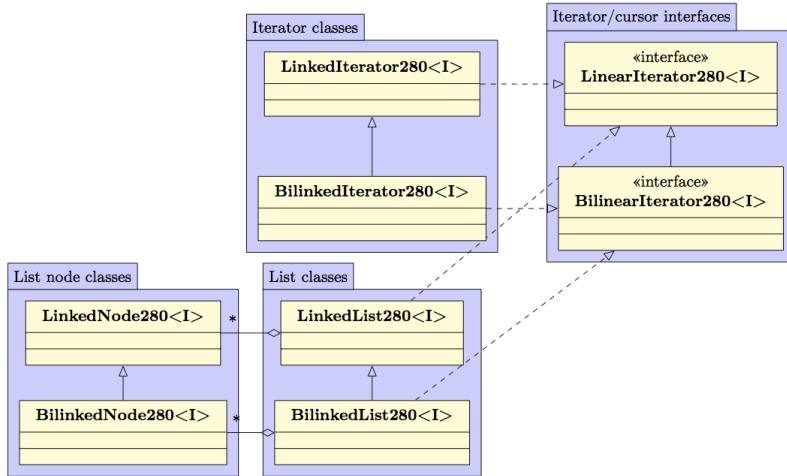
(Remember: here, purple packages are for grouping, not packages.)

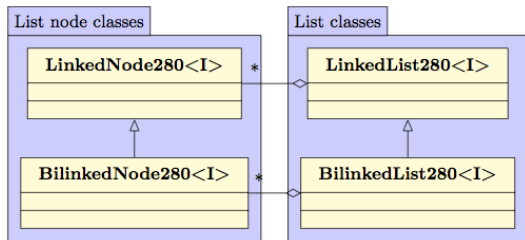# Singly- and Doubly- Linked List Classes in `lib280`

List Nodes

- Each one stores an object from generic class I.

- A `BilinkedNode280` is a little bit more than a `LinkedNode280`.

- Includes a previous (`prev`) reference in addition to the `next` node reference in `LinkedNode280`; also accessor and mutator for the `prev` reference.

Singly- and Doubly- Linked List Classes in `lib280`

Lists

- Lists are collections of nodes.

- A *BilinkedList280* is a little bit more than a LinkedList. E.g. includes `tail` reference not in `LinkedList280`.

- Some inherited methods must be overridden in *BilinkedList280* to keep track of `tail` and node `prev` pointers.
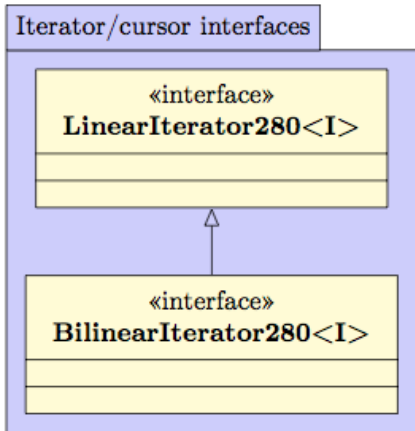
Singly- and Doubly- Linked List Classes in `lib280`

An interface is a specific pattern of behaviour.

- `boolean before()`
- `void goBefore()`
- `boolean after()`
- `void goAfter()`
- `void goFirst()`
- `void goForth()`
- `I item()`
- `boolean itemExists()`

`BlinearIterator280` adds:

- `void goLast()`
- `void goBack()`

(Image: UML diagram titled "Iterator/cursor interfaces" showing «interface» **LinearIterator280<I>** with an inheritance arrow from «interface» **BilinearIterator280<I>**.)

(Actually in the `lib280.base` package.)

Singly- and Doubly- Linked List Classes in `lib280`
Iterators
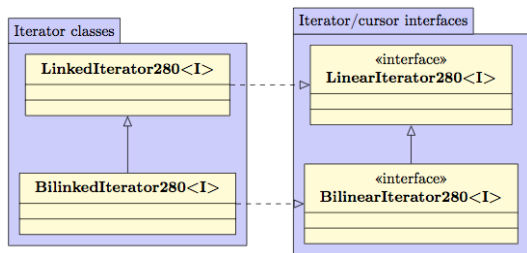
- `LinkedIterator280` and `BilinkedIterator280` implement the linear iterator interfaces.

Singly- and Doubly- Linked List Classes in `lib280`

Cursors

The list classes can have cursors by implementing the same linear iterator interfaces.

- `BilinkedList280` directly implements `BilinearIterator280`.

- `LinkedList280` implements `LinearIterator280`

Thus the linked lists in lib280 have both cursors and iterators.

Iterator/cursor interfaces

«interface»
LinearIterator280<I>

«interface»
BilinearIterator280<I>

List classes

LinkedList280<I>

BilinkedList280<I>
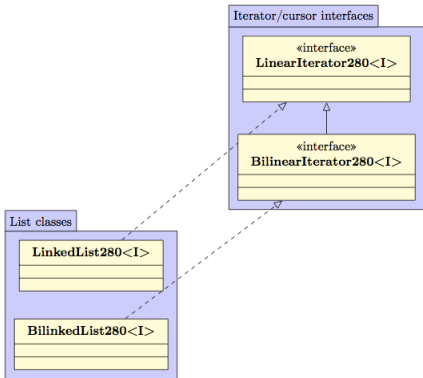
Iterators

Reminder:

Iterators provide the same functionality as a container ADT that has a cursor, but they are separate objects from the container. This allows us to record a cursor position that is different and independent from the position recorded by the container's internal cursor.

Cursors vs. Iterators
Uses of Cursors

- Insert an element y immediately before an element x already in the list:
    1. search(x) to find x
    2. insertBefore(y) to add y right after x
- Delete an item from the list:
    1. search(x) to find x in the list
    2. if itemExists() then deleteItem() to delete x

These tasks use the cursor to do something to the list based on the position of the cursor. The internal representation of position is **abstracted**. The ArrayedList280 classes uses the same cursor interface, so positions within a list can be manipulated in exactly the same way for both classes, despite the differences in the internal data structure.

Cursors vs. Iterators
Iterators

An iterator is external, for accessing data only indirectly.

- Example uses:

    - Printing out everything in a list

    - Keeping track of multiple positions in a list.

- An iterator is how you go through the elements of a structure.

Cursors vs. Iterators

- If a class exposes the interface to its cursor, then the cursor can be used like an iterator too. This is the case for the linked list classes in lib280. It is therefore important that list methods that might use the cursor in their implementation, but are not supposed to move the cursor from the user's perspective, save and restore its position.

- For example, the has() method in LinkedList280 uses the search() method to see if the given element is in the list. The search() method moves the cursor. But the has() method isn't supposed to cause the cursor to move! So notice how has() saves the cursor position using currentPosition and then, after using search(), restores the original cursor position using goPosition(). This avoids the undesirable **side effect** of has causing the cursor to move.

Cursors vs. Iterators

- But a class might NOT expose its cursor, for example, a stack. The current position in a stack is always the top element. Stacks do not include public methods to manipulate the cursor position because it should not ever be moved! But one could envision a stack with an iterator that lets you examine the contents of the stack. This would be OK because the internal cursor would not be affected.

Part II

Using the `LinkedList280` Class

Demo: `ListDemo.java`
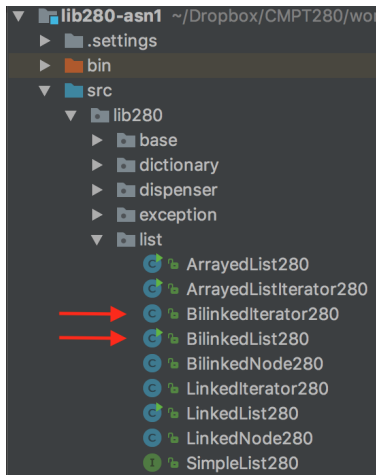
Let's take a look at `ListDemo.java`

Part III

Assignment 1

Importing `lib280-asn1` Lib280

- Demonstration of how to import lib280 into an IntelliJ project.
- Follow the parts 1 and 2 of the self-guided tutorial from the website.

Navigating `lib280-asn1` Within IntelliJ

- Important files for Asn1 are `BilinkedList280.java` and `BilinkedIterator280.java`

- Definitely ignore the other packages for now.

- Be aware of the other classes in `lib280.list` and their relationships to the bi-linked list classes.

Importing `lib280-asn1` Lib280

- Demonstration of how to create a module that references lib280 as a "module dependency".

- Follow the part 3 of the self-guided tutorial from the website.

The actual questions
Question 1

Hints and Reminders;

- The Sack.java and Grain.java classes are given. You don't have to write it yourself.

- In Ship.java, complete the isOverloaded() and sacksOfGrainType() methods.

- In CargoSimulator.java, you only have to update the main() method to display how many sacks of wheat each ship is carrying and a message for each ship that is overloaded.

The actual questions
Question 2

- Fill in the // TODOs based on the method headers.

- **ONLY** add code at the // TODOs. Do not change existing code.
  (This is both easier and required.)

The actual questions
Question 2

Tips and tricks:

- The javadoc comments are how your javadoc comments should look on future assignments.

  - They are representative in terms of both appearance and content.

- In general, good javadoc comments mean that IntelliJ will explain code to you.

  - Specifically, they'll explain the `lib280` to you.

The actual questions
Question 3

- Write regression tests for the new methods you wrote.
- Guidelines for writing tests can be found in Topic 2.