In-class exercises (lecture 10)

Exercise 1: Comparable

Take a look at the Book class in the lecture 10 code > datatypes package.

How would you implement `compareTo`?

- When should a Book be **less than** another Book?
- When should a Book be **equal to** another Book?
- When should a Book be **greater than** another Book?

If you have time, try implementing `compareTo`.

- The Book class will need to implement interface `Comparable<Book>` - add this to the class definition.
- You will need a `compareTo` method that takes a Book as its only parameter and returns an int. The returned int should be negative if the current Book (this) is less than the given Book, 0 if the two Books are equal, and positive if the current Book is greater than the given Book.

Exercise 2: Remove redundancy caused by the iterator

The `ListIterator`'s `next` method is causing a lot of redundant operations because it uses the `IListADT`'s get method, which is also iterating through each node. You can see this when you use a for loop to iterate through a `NestedList`. Your task is to try to remove the redundancy by making the underlying inner nodes iterable, instead of relying on the external `ListIterator` class.

Your outer class (`NestedList`) will still need an `iterator` method, but instead of returning an instance of the `ListIterator` class, it will return an instance of a new static inner class that implements `Iterator`.

Here are the steps:

1. Add `extends Iterable<T>` to the `ILinkedListIterable` interface definition.
2. Both implementing node classes will need to implement `Iterable`'s required `iterator` method. Add placeholders for now—just use the methods auto-generated by IntelliJ.
3. Create a new static inner class called `LinkedListIterator<T>`, which should implement `Iterator<T>`. Implement the required `hasNext` and `next` methods. **This is the focus of the exercise**. You will need to figure out how to implement these efficiently.
4. Fill in the `iterator` method in each node class. They should return a new `LinkedListIterator`

starting from the current node.

5. In the outer class `iterator` method, return a new instance of `LinkedListIterator`.

Exercise 3 (optional extension): BST with nested node

The goal of this extension exercise is to give you more practice with nested classes and to familiarize you with a Java implementation of a binary search tree. Sample solutions will be provided after class but I encourage you to try this on your own before looking at them.

The provided BinaryTree class uses an external class, BinaryNode, to store data and create the tree structure. For better information hiding and abstraction, we could move that node inside the tree class.

Create a new class called NestedBinaryTree that implements the provided IBinaryTree. Within the NestedBinaryTree class, create a private static inner class to represent the node. This class will need the same methods that are in BinaryNode and the implementation will be the same (change the datatypes where necessary, of course).

In the outer class, implement the IBinaryTree methods. You can use the provided BinaryTree class as reference—the new implementation will be very similar!