CS 61B Fall 2025

Inheritance, Comparators, Generic Functions

Exam-Level Discussion 03: September 15, 2025

1 Default

Suppose we have a `MyStack` interface that we want to implement. We want to add two default methods to the interface: `insesrtAtBottom` and `flip`. Fill in these methods in the code below.

```
public interface MyStack<E> {
    void push(E element); // adds an element to the top of the stack
    E pop();              // removes and returns the top element of the stack
    boolean isEmpty();    // returns true if the stack is empty
    int size();           // returns the number of elements in the stack

    // inserts the item at the bottom of the stack using push, pop, isEmpty, and size
    private void insertAtBottom(E item) {









    }

    // flips the stack upside down (hint: use insertAtBottom)
    default void flip() {









    }
}
```

2 HOFery

Suppose we have the following higher-order function interface:

```
public interface UnaryFunction {
  public Integer apply(Integer x);
}
```

A **Spy** is a **UnaryFunction** that wraps around another **UnaryFunction**, **f**. For any input **x**, it returns the result **f** would return, but it remembers the arguments on which it has been called and can print them out on command.

For example, if **SquareFunction** represents a function that squares integers:

```
UnaryFunction sq = new SquareFunction();
System.out.println(sq.apply(4)); // prints ‘‘16’’, not including quotes
Spy spy = new Spy(sq);
System.out.println(spy.apply(2)); // prints "4"
System.out.println(spy.apply(3)); // prints "9"
spy.printArgumentHistory(); // prints "2 3 "
```

Complete the Spy class below.

```
public class Spy _____ {

  _____

  _____


  public Spy(_____) {

    _____

    _____

  }
  @Override
  public Integer apply(_____) {

    _____

    _____

  }
  public void printArgumentHistory() {

    _____

    _____

    _____

  }
}
```

3 The Downside of Default

Consider the **`ListOfInts`** interface below. **`addLast`**, **`get`**, and **`size`** behave exactly as your Deque interface from mini-project 1.

`set(int i, int value)` sets the ith integer in the list equal to value.

`plusEquals` adds each int in x to the corresponding int in the current list.

Example: if we have a list L = [2, 3, 4] and we call L.plusEquals([5, 6, 7]), then after the call is complete, L will contain the values [7, 9, 11].

If the lists are not of the same length, plusEquals should have no effect.

(a) Fill in the **`plusEquals`** method below.

```
public interface ListOfInts {
  public void addLast(int i);
  public int get(int i);
  public int size();
  public void set(int i, int value);

  default public void plusEquals(ListOfInts x) { // assume x is non-null

    if (_____________________________){ return; }

      for (int i = 0; _____________________________) {

        this.set(i, _____________________________);
      }
    }
  }
}
```

The **DLListOfInts** class stores a doubly linked list of integers, similar to your **LinkedListDeque** class. Assume that the list has a single sentinel node that points at itself when the list as empty, just as in lecture and in the recommended approach for mini-project 1. Fill in the **plusEquals** method so that it behaves as in the previous part.

You cannot call `super.plusEquals`

(b)
```
public class DLListOfInts implements ListOfInts {
  public class IntNode {
    public int item;
    public IntNode next, prev;
  }

  private IntNode sentinel;
  private int size;

  public void plusEquals(DLListInt x) {

    if (_____) {
      return;
    }

    IntNode xPtr = x.sentinel.next;

    for (IntNode p = sentinel.next; _____) {

      _____;

      _____;
    }
  }
}
```