CSE 12 Homework 3 Getting Started --- Stack

Tracking Memory

Four sections of memory:

1. Text
2. Data
3. RTS
4. **Heap**

Tracker

Tracker is used to keep track of everything that we put onto the heap.

- Tracking
    - For every object that you created, it should contain an instance of the Tracker object
        - The Tracker object's constructor (refer to code in `LongStack.java`) takes in a size and a string
            - You are supposed to write that string, that string is for you to debug your memory usage
            - Should make it descriptive for your own benefit.
- Jettison
    - Once you no longer need
        - Program come to an end
        - The variable is going to be replaced by some new object
    - You need to jettison the object!!!!!!
        - That is for each object you should write a jettison method.
            - Which should call Tracker's jettison
            - Delegate
            - We wrote this for the `LongStack.java` for you, but you are going to write your own jettison in later assignments.
- Inspect Memory
    - You can inspect the current tracked objects at any point in your code.
    - You just call `Tracker.checkMemoryLeaks()` to see all the objects that you allocated in the heap
        - All that tracker is currently tracking will be displayed.

Driver, Stack, & Stack Engine

Overall idea: Layering.

TODO:

LongStack

AGAIN don't change anything in it.

LongStackEngine

- Java boxing (This is in Gary's notes)
  - You use this in to handle invalid/errorish situation
    - `return null` when it is bad.
      - For example, popping from an empty stack
    - `return stack[0]` when it is good
  - This is an implicit call.
- Constructor
  - Correctly allocate all the instance variable
    - You don't need to deal with tracker
  - `stack`
    - The array that is your actual stack
    - Init it base on size
  - `stackPointer`
    - Points to where the last occupied place is in your `stack`
    - Think about what is the first empty spot so that you can correctly init it.
  - `stackSize`
    - Self-explanatory
  - `stackId`
    - Although in this PA we only have one stack always.
    - But you should make use of the `static variable stackCounter` on line 6
    - You need to also increment that counter once a new stack is allocated
    - This is for your next PA.
  - Debug message
    - `ALLOCATE`
- `jettisonStack`
  - Wrote it for you, refer to this for how to use `debug messages`.
- `emptyStack`

    - You should empty the stack.
    - Think about what does it mean for the stack to be empty
    - It should be a one-liner
    - Debug message
- `getCapacity`
    - What is the capacity of a stack?
- `getOccupancy`
    - Think about what reflect the occupancy
- `isEmptyStack`
    - You can call a method that you wrote. And compare the output with a number.
- `isFullStack`
    - You can call a method that you wrote. And compare the output with another number
- `push`
    - Put it in the next empty place
        - How do you know where?
            - `stackPointer`
    - Then increment
    - Note:
        - Check for edge case
            - Make use of the catastrophic messages
- `pop`
    - Access the last object in the `stack`
        - How?
        - stackPointer - 1
        - You can be smarter but not necessary
    - Decrement
    - Note:
        - Make sure you check for the edge case
            - Make use of the catastrophic messages
- `top`
    - Similar to pop

Final Notes

- Make sure you use the debug / catastrophic messages in corresponding methods.
- Make sure you **read** the development guide.
- Make sure you match the output that is listed in the development guide.

    - You want to have a **match** for the **error messages and debug messages.**
- Make sure you come up with test cases and think about their expected behaviors, and test your code thoroughly.