Indiana University
CSCI-C212: Introduction to Software Systems (F25) Problem Set 9
Joshua Crotts October 29, 2025

Mutation and Aliasing

Important Dates:

- Assigned: October 29, 2025
- Deadline: November 5, 2025 at 11:59 PM EST

Objectives:

- Students begin to understand the purpose of mutation and aliasing.
- Students design more complex classes to take advantage of mutation and aliasing.
- Students re-implement data structures that exist in other programming languages to understand how they work.

What To Do:

Design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

You do not need to test the accessor/getter methods.

You **do** need to test mutator methods.

You do not need to test `hashCode`.

What You Cannot Use:

You cannot use any content beyond Chapter 4.2. Namely, do not use interfaces, inheritance, or anything that trivializes the problem. Please contact a staff member if you are unsure about something you should or should not use. Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set.

Warning:

This assignment is rather difficult, and you need to start it early. If you want to go easiest-to-hardest, that's fine, but you won't earn as many points. Going from hardest-to-easiest might be better for points, but don't spend *too* long on any problem! We attach points to each problem this time to emphasize their relative difficulty, but this is our perceived difficulty; not how you feel about them!

Problem 1 (10 points):

Design the `Temperature` class to represent a temperature in some unit. The constructor should receive a temperature, as a double, in kelvin. Only store one instance variable: the value in kelvin. Then, design six methods for interoperating with the stored temperature: `double getCelsius()`, `double setCelsius(double c)`, `double getFahrenheit()`, `setFahrenheit(double f)`, `double getKelvin()`, and `setKelvin(double k)`. You may assume that the inputs to each method are well-formed, i.e., kelvin values are never negative, Celsius never goes below -273.15, and Fahrenheit never goes below -459.67. Override the `public String toString()` method to return a string with the temperature, formatted to exactly two decimal places, in kelvin, e.g., `"23.55 K"`.

Problem 2 (20 points):

In the two-player game of Mancala (play the game here!), the goal is to collect the most stones. The board has N pits plus two score pots (one for each player). Indices 0 to $(N/2 - 1)$ are player 1's pits, index $N/2$ is player 1's score pot, indices $(N/2 + 1)$ to $(N - 2)$ are player 2's pits, and index $N - 1$ is player 2's score pot.

Here's how the game works: Each pit starts with S stones. On a turn, a player selects one of their pits to play, removes all stones from the selected pit, and distributes them one-by-one from left-to-right (left-to-right in terms of indices into the array), skipping the opponent's score pot. If the last stone lands in the player's own score pot, they get another turn. If it lands in an empty pit on their side, they capture the opposite pit's stones, plus their own, and place them in their score pot. Otherwise, play passes to the other player.

To make the game interaction simpler, we will represent the Mancala board as a one-dimensional array, then present it as a two-dimensional array. Because our board is only one-dimensional, we can simply distribute the stones linearly, counting the indices upwards.

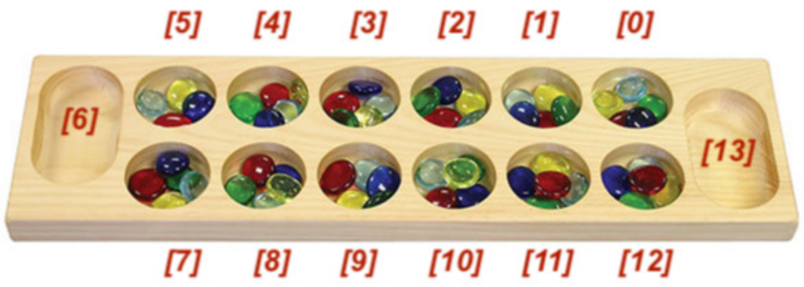

Figure 1: Mancala Board Example

In the above representation, indices 0-5 are player 1's pits, index 6 is player 1's score pot, indices 7-12 are player 2's pits, and index 13 is player two's score pot. **Remember that N, in this case, is still 12 and not 14.**

(a) First, design the `Mancala` class, which stores an `int[]` representing the board state. It should also store an integer representing whose turn it is currently.

(b) Design the `Mancala` constructor. It receive a number of pits N and a number of stones per pit S. For Mancala games, N should be at least 12 and even. The number of stones per pit S should divide evenly into N.

(c) Design the `int[] getBoard()` accessor method for returning the state of the board.

(d) Design the `int getTurn()` method, which returns 0 when it is player one's turn and 1 when it is player 2's turn.

(e) Design the `boolean playGame(int pos)` method, which implements the Mancala game via the rules as described above. It receives the index into the array where to start the game play. You should mutate the underlying one-dimensional array and modify whose turn it is after it completes. The method returns `true` if the given position was valid and `false` otherwise. A position is valid if: (1) the position lies within the current player's pits and (2) the position is not a score pot.

Problem 3 (25 points):

(a) Design the generic `Matrix<T>` class, which stores a two-dimensional array of elements of type `T`. Its constructor should receive a two-dimensional array of type `T` such that the number of rows and columns are at least 1. Copy the elements from the passed array into an instance variable array. Do *not* simply assign the provided array to the instance variable!

Be sure to include three methods for accessing the rows, columns, and the underlying array. Call them `int getRows()`, `int getCols()`. Accessing the underlying array is not as straightforward because of Java's type erasure feature with generics. Therefore, we must label the respective header as `Object[][] getMatrix()`.

(b) Design the `boolean set(int i, int j, T val)` method, which sets the value at row i and column j to `val`. If r or c is out of bounds, do nothing and return `false`, otherwise return `true`.

(c) Design the `boolean add(Matrix<T> m, BiFunction<T, T, T> adder)` method, which adds the corresponding entries of the passed matrix `m` to `this` matrix using the provided `adder` function. If the dimensions of the two matrices do not match, return `false` and do not modify the current matrix. Otherwise, modify `this` matrix appropriately and return `true`.

Hint: remember that a `BiFunction` is nothing more than a two-argument function that returns some value! Remember that the `reduce` method uses a `BiFunction`. Don't let that be the thing that confuses you!

(d) Design the `boolean multiply(Matrix<T> m, BiFunction<T, T, T> multiplier, BiFunction<T, T, T> adder, T zero)` method, which multiplies `this` matrix by another matrix `m`. Two matrices $A = P \times Q$ and $B = R \times S$ can be multiplied if and only if $Q = R$. The resulting matrix has dimensions $P \times S$. If the matrices cannot be multiplied, return `false` and do not modify `this` matrix. The product of two matrices A and B is the matrix C where:

$$C_{i,j} = \sum_{k=1}^{N} \texttt{multiplier.apply}(A_{i,k}, B_{k,j})$$

Where the sum is accumulated via the `adder` object, starting from the zero element `zero`.

Hint 1: to follow this formula, you will need 3 for loops.

Hint 2: if the notion of how to write this method is confusing you, think about how it would be invoked. For example, assuming `m` and `m2` are `Matrix<Integer>` instances, then we can invoke `multiply` as: `m.multiply(m2, (a, b) -> a * b, (a, b) -> a + b, 0)`.

(e) Design the `void transpose()` method, which transposes the matrix. That is, the rows become the columns and the columns become the rows. You may need to alter the dimensions of the matrix.

(f) Design the `void rotate()` method, rotates the matrix 90 degrees clockwise. To rotate a matrix, compute the transposition and then reverse the rows. You may need to alter the dimensions of the matrix.

(g) Override the `public String toString()` method to return a stringified version of the matrix. As an example, a `Matrix<Integer>` such as `"[[1, 2, 3], [4, 5, 6]]"` represents the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Problem 4 (15 points):

Repeated string concatenation is a common performance issue in Java. As we know, Java `String` objects are immutable, which means that concatenation creates a new `String` objects. This is fine for small strings, but for larger strings (or concatenation operations performed in a loop), this can be a performance bottleneck. Each concatenation requires copying the entire string. Java provides the `StringBuilder` class to alleviate the issue. In this exercise, you will design a `MiniStringBuilder` class that mimics the behavior of `StringBuilder`. You cannot use `StringBuilder` or the older `StringBuffer` classes in your implementation.

(a) Design the `MiniStringBuilder` class, which stores a `char[]` as an instance variable. The class should also store a variable to keep track of the number of "logical characters" that are in-use by the buffer.

(b) Design two constructors for the `MiniStringBuilder` class: one that receives no arguments and initializes the default capacity of the underlying `char[]` array to 20, and another that receives a `String` *s* and initializes the `char[]` array to the characters of *s*.

(c) Override the `public boolean equals(Object o)` method to return whether two `MiniSt-ringBuilder` objects represent the same string.

(d) Override the `public int hashCode()` method to return the hash code of the `MiniStringB-uilder` object. The hash code is defined as the hash code of the underlying array of characters. Use `Arrays.hashCode` rather than `Objects.hash`.

(e) Override the `public String toString()` method, which returns the `char[]` array as a `String` object. The resulting string should contain only the logical characters in the buffer, and not the entire array. Output the characters without any additional characters, such as brackets or commas.

(f) Design the `void append(String s)` method, which appends the given string *s* onto the end of the current string stored in the buffer. The given string should not simply be appended onto the end of the buffer, but rather added to the end of the previous string in the buffer. If the buffer runs out of space, reallocate the array to be twice its current size, similar to how we reallocate the array in the `MiniArrayList` example class.

(g) Design the `void clear()` method, which resets the `char[]` array to the default size of 20 and clears the character buffer.

Problem 5 (30 points):

A *persistent data structure* is one that saves intermittent data structures after applying operations that would otherwise alter the contents of the data structure. Take, for instance, a standard FIFO queue. When we invoke its 'enqueue' method, we modify the underlying data structure to now contain the new element. If this were a persistent queue, then enqueueing a new element would, instead, return a new queue that contains all elements and the newly-enqueued value, thereby leaving the original queue unchanged.

(a) First, design the generic, private, and static class `Node` inside a generic `PQueue` class skeleton. It should store, as instance variables, a pointer to its next element as well as its associated value.

(b) Then, design the `PQueue` class, which represents a persistent queue data structure. As instance variables, store "first" and "last" pointers as `Node` objects, as well as an integer to represent the number of existing elements. In the constructor, instantiate the pointers to `null` and the number of elements to zero.

(c) Design the `private PQueue<T> copy()` method that returns a new queue with the same elements as the current queue. You should divide this method into a case analysis: one where `this` queue is empty and another where it is not. In the former case, return a new queue with no elements. In the latter case, iterate over the elements of the queue, enqueuing each element into a new queue. You will need instantiate a new `Node` (reference) for each element.

(d) Override the `public boolean equals(Object o)` method that returns whether the elements of this queue are equal to the provided queue's elements. You will need to traverse over the queues in a fashion similar to how you do it in `copy`. Again, break this up into a case analysis: (1) if the provided object is not a `PQueue<?>`, return `false`. (2) Otherwise, if they do not have the same number of elements, return `false`. Otherwise, compare each element sequentially.

(e) Design the `PQueue<T> enqueue(T t)` method that enqueues a value onto the end of a new queue containing all the old elements, in addition to the new value. You should use the `copy` method to your advantage.

(f) Design the `PQueue<T> dequeue()` method that removes the first element of the queue, returning a new queue without this first value. You should use the `copy` method to your advantage.

(g) Design the `T peek()` method that returns the first element of the queue.

(h) Design the `static <T> PQueue<T> of(T... vals)` method that creates a queue with the values passed as `vals`. Note that this must be a variadic method. **Warning:** do not create a series of `PQueue` objects by enqueueing each element into a distinct queue; this is incredibly inefficient. Instead, allocate each `Node` one-by-one, thereby never calling `enqueue`.

(i) Design the `int size()` method that returns the number of elements in the queue. You should not traverse the queue to compute this value.