Project 1: Dynamic Grid

DUE: Friday, Sept. 19th at 11:59pm

Basic Procedures

You must:

- Pass the provided style checker and JavaDoc checker from "Coding Warm-Up" on all your Java files.
- Comment your code well IN ADDITION to passing the provided JavaDoc checker on all your Java files for this project (no need to overdo it, just do it well).
- Comment your code well in JavaDoc style (no need to overdo it, just do it well)
- Have code that compiles from your code directory with the command:
  - `javac -cp .;../junit-4.11.jar *.java` (Windows)
  - `javac -cp .:../junit-4.11.jar *.java` (Linux/MacOS)
- Have code that runs with the commands:
  - `java Table`
  - `java GUI`

You may:

- Add additional methods and variables, however these methods **must be private**.

You may **NOT**:

- Make your program part of a package.
- Add additional public methods or variables
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no `ArrayList`, `LinkedList`, `HashSet`, etc.).
- Use any arrays anywhere in your program (except the data field provided in the `DynamicArray` class)
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided classes that are complete (`Combiner`, `IntegerTimer`, `IntegerAdder`, and `GUI`).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.
- Use any code that violates the CS Academic Standards code (code from the internet, code from AI, code from friends, etc.)
- Add `@SuppressWarnings` to any methods unless they are private helper methods for use with a method, we provided which already has an `@SuppressWarnings` on it.

Setup

- Create a folder for `p1`. This is your "project directory".
- Download the `p1.zip` and unzip it in your project directory. This will create a folder `yourCodeHere`. This is your "code directory".
- Move the style/Javadoc checker from "Coding Warm-Up" to P1 by taking the following files: `checkstyle.jar`, `cs310code.xml`, and `cs310comments.xml`, and put them in your project directory.

Submission Instructions

- Make a **backup copy** of your project folder and upload it to your OneDrive.
- Go to Gradescope and ***only submit the Java files from your CODE directory*** to the **Project 1** submission link.
- You can resubmit as many times as you'd like, only the last submission will be graded.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

By the end of this project, you will have defined a generic two-dimensional grid that can be used to implement various kinds of 2D tables. Normal table operations will be supported, including appending, inserting, and removing rows and columns. The table will be generic so that it can be used to represent different kinds of binary operations such as integer multiplication or string concatenation.

The table you are to implement includes the following components:

- A list of values of type **`A`** which will be the row headers (**`rowHead`**).
- A list of values of type **`B`** which will be the column headers (**`colHead`**).
- A two-dimensional list of values of type **`C`** (the **`grid`**).
- An operation that defines a binary function **`f(A,B)→C`**.
    - Note that **`A`**, **`B`**, and **`C`** can be different types/classes, so we can combine integers and floats, integers and strings, strings and strings, etc.

Once **`rowHead`**, **`colHead`**, and operation function **`f`** are defined, the **`grid`** can be calculated based on the formula below:

```
grid[i,j] = f( rowHead[i], colHead[j] )
```

You can find two examples in Figure 1 below. The example on the left represents a normal multiplication table for integers. The example on the right shows a table with string repetition – the **`rowHead`** is a list of strings while the **`colHead`** is a list of integers, and each cell at row **`i`** and column **`j`** is obtained by making **`colHead[j]`** copies of **`rowHead[i]`** concatenated together.

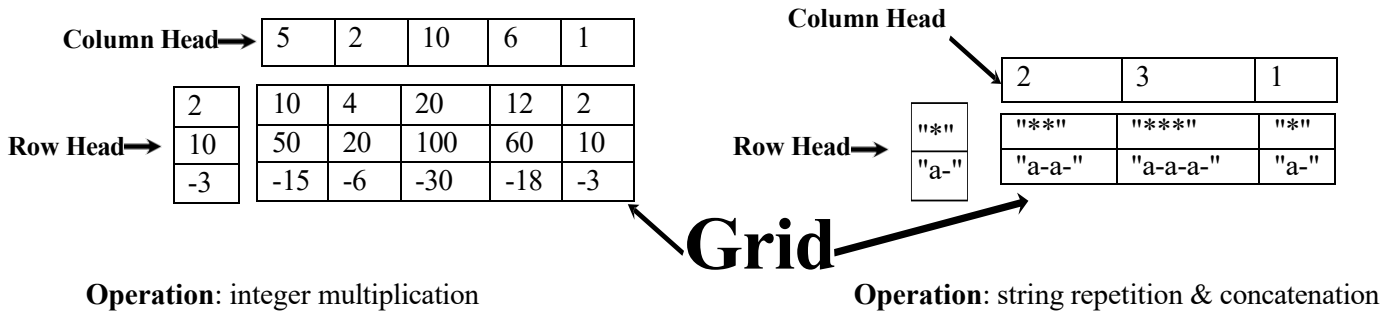


Figure 1: Table Structure and Examples

Implementation/Classes

This project will be built using a number of classes representing the component pieces of the table we described in the previous section. Here we provide a description of these classes. **Template files are provided for each class in the code directory and these contain further comments and additional details.**

DynamicArray<T> (DynamicArray.java): This class represents a generic collection of objects in an array whose capacity that can grow and shrink as needed. It supports the usual operations of adding and removing objects including **`add()`** and **`remove()`**. Check the included **template file** for all required methods and their corresponding big-O requirements.

The capacity of a **DynamicArray** must change as items are added or removed and the following rules must be followed in your implementation:

- The default initial capacity is fixed to be 2.
- When you need to add items and there is not enough space, grow the array to double its capacity.
- When you delete an item and the size falls below 1/3 of the capacity, shrink the array to half its capacity.

DynamicGrid<T> (DynamicGrid.java): This class represents a generic two-dimension grid of objects created from a DynamicArray of DynamicArrays. The capacity of both dimensions can grow and shrink, similar to the **DynamicArray** class and you should use the functionality you already built in DynamicArray as part of your solution. **DynamicGrid** will also support the operations of adding and removing including **`addRow()`**/**`addCol()`** and **`removeRow()`** / **`removeCol()`**. Again, check the **template file** for all required methods and their corresponding big-O requirements.

Combiner<A,B,C> (Combiner.java): This is a generic interface that we use to generalize any binary operation taking two operands (of type `A` and `B` respectively) which returns a type `C` result. The only required method in this interface is:

```
public C combine(A operand1, B operand2);
```

This class is provided to you and you should NOT change the file. Note: The combine operation may not be commutative.

IntegerTimer (IntegerComb.java): This class provides an example that implements the Combiner interface. It can be used to perform multiplication on two integers. This class is provided to you and you should NOT change the file.

IntegerAdder (IntegerComb.java): This class provides an example that implements the Combiner interface. It can be used to perform addition on two integers. This class is provided to you and you should NOT change the file.

StringAdder (StringAdder.java): This class can be used to perform concatenate two strings. This class is provided to you and you should NOT change the file.

StringTimer (StringTimer.java): This class implements the Combiner interface and defines the string repetition as used in Figure 1 above. You need to define the `combine()` method and add JavaDocs. Check the **template file** for details.

SubstringCounter (SubstringCounter.java): This class implements the Combiner interface. Its `combine()` method accepts two strings `s1` and `s2`, counts and returns how many times `s2` occurs as a substring in `s1`. You need to define the required `combine()` method and add JavaDoc comments. Check the **template file** for details.

Table<RowType, ColType, CellType, OpType extends Combiner<RowType, ColType, CellType>> (Table.java): This generic class has four type parameters to facilitate the definition of general two-dimension tables in the structure described above.

1. **RowType**: specifies the type of elements stored in `rowHead` of the table.
2. **ColType**: specifies the type of elements stored in `colHead` of the table.
3. **CellType**: specifies the type of elements stored in `grid` of the table.
4. **OpType**: specifies the operation applied on `row[i]` and `column[j]` to determine the value of `grid[i,j]`, i.e.

```
grid[i,j] = combine(row[i], col[j])
```

This class stores `rowHead` and `colHead` as DynamicArrays, and the 2-D grid as a DynamicGrid. You will need to implement various getter and setter methods and methods to support inserting/removing/changing a row/column of the table. In addition to those, the `setOp()` method can redefine the operation used to calculate the cell values but the *types* of `rowHead`, `colHead`, and the `grid` should remain the same (to change the *types* one would need to create a new table). Check the included **template file** for the details of required data fields and methods.

GUI (GUI.java): This class implements a graphical user interface that creates a table of color mixing. Using the main method in this file you can use buttons to add/remove/change row/column of the color table. You can also choose which two colors in the RGB color space are used to control the mix while the third one is fixed to be zero. The `GUI` class is an application of `Table` class and only works if your implementation of `Table` is complete. Inside `GUI.java`, you can also find more examples of classes that implement the generic interface `Combiner` (see `ColorComb` and its subclasses). This class is provided to you and you should NOT change the file.

Requirements

An overview of the requirements are listed below, please see the grading rubric for more details.

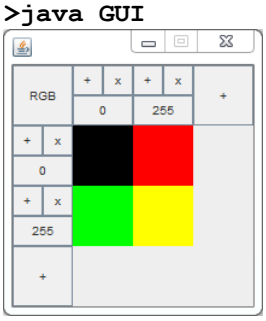
- **Implementing the classes** - You will need to implement required classes and fill the provided template files.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods.
- **Big-O** - Template files provided to you contains instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

Testing

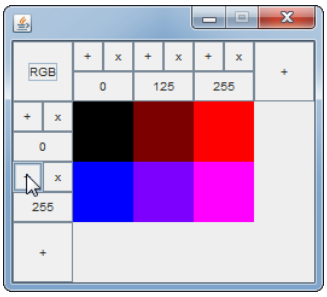
The main methods provided in the template files contain useful code to test your project as you work. You can use command like "`java DynamicArray`" to run the testing defined in `main()`. You could also edit `main()` to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

Example Run (GUI)

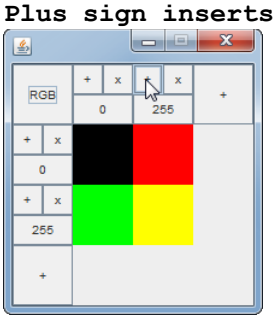
>java GUI



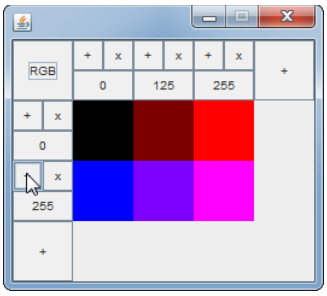
Now red column values and blue row values



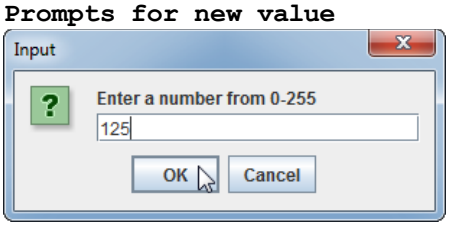
Plus sign inserts new colHead value



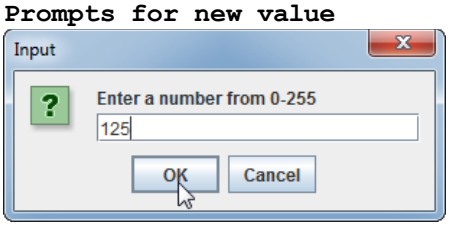
Plus button inserts new rowHead value



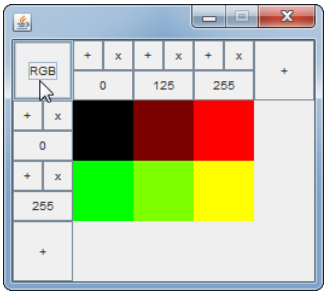
Prompts for new value



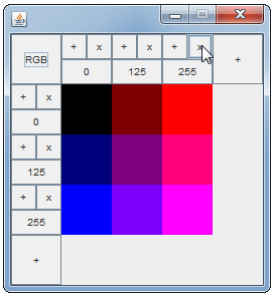
Prompts for new value



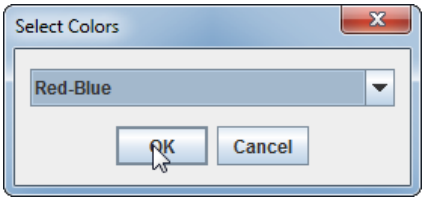
New table with inserted value



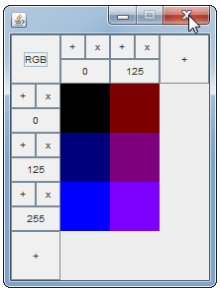
"x" button deletes column/row.



RGB button selects new color choice



New table with deleted column



Example Run (Command Line)

Note: The provided Table.java has implemented a **`toString`**() method that you can use to print out the table content to check. Inside the provide main, there are five lines of printing code you can un-comment in order to grab a print of the tables. The sample output included here corresponds to the provided **`main`**() and with the "Yay" output removed.

```
>java Table
============================
Table
Operation: class StringAdder
Size: 2 rows, 1 cols
       |        apple
--------------------
    red|    red apple
 yellow| yellow apple
============================

============================
Table
Operation: class StringAdder
Size: 3 rows, 3 cols
       |        apple|        kiwi|        banana
-----------------------------------------------
    red|    red apple|    red kiwi|    red banana
 yellow| yellow apple| yellow kiwi| yellow banana
  green|  green apple|  green kiwi|  green banana
============================

============================
Table
Operation: class StringAdder
Size: 2 rows, 3 cols
       |        apple|        kiwi|        orange
-----------------------------------------------
 yellow| yellow apple| yellow kiwi| yellow orange
  green|  green apple|  green kiwi|  green orange
============================

============================
Table
Operation: class IntegerAdder
Size: 5 rows, 5 cols
  | 50| 40| 30| 20| 10
---------------------
 1| 51| 41| 31| 21| 11
 2| 52| 42| 32| 22| 12
 3| 53| 43| 33| 23| 13
 4| 54| 44| 34| 24| 14
 5| 55| 45| 35| 25| 15
============================

============================
Table
Operation: class IntegerTimer
Size: 5 rows, 5 cols
  |  50|  40|  30|  20| 10
------------------------
 1|  50|  40|  30|  20| 10
 2| 100|  80|  60|  40| 20
 3| 150| 120|  90|  60| 30
 4| 200| 160| 120|  80| 40
 5| 250| 200| 150| 100| 50
============================
```

Table with StringAdder;
Two rows and one column added

More rows and columns added

Row deleted and column changed

Table with IntegerAdder;
Multiple rows and columns

Reset the op of the table to be IntegerTimer;
Same rowHead and colHead but cells updated