GMU Spring 2023 – CS 211 – Project 5

2% extra credit for an optional task

Due Date: Saturday, May 6th, 11:59pm

Changelog (all updates are highlighted in green in the document)

- In **`randomIntegers`** what's random is the placement, not the quantity, of the obstacles
- The use of `java.util.Random` is permitted **and recommended** (read the changelog in method `randomIntegers`)

Overview

The purpose of this assignment is to practice iterators, inner classes, recursion and, optionally, graphical user interfaces.

There are mainly two approaches to implement an iterator in Java. One is by implementing the **`Iterator`** interface as an anonymous inner class, and the other one is by writing a separate class that implements the **`Iterator`** interface. You're going to practice both approaches by writing two different iterators. Additionally, you will implement a recursive method to solve an algorithmic task. Last, you can optionally implement a GUI to make your application more user-friendly and get 2% extra credit (i.e. this project can count for **9%**, instead of the default 7%, of your overall course grade).

Instructions

- Honor Code: This assignment is **individual work** of each student, you're not allowed to collaborate in any form. Copying code from other sources (peers, websites, etc.) is a serious violation of the Honor Code. Your submission will be examined for similarities with other sources.
- Validation: Make sure all method parameters are properly validated. If a parameter has an invalid value, the method must raise an **`IllegalArgumentException`** (you can use any message you want in this case).
- Documentation: Comments in JavaDoc-style are **required**. You must comment each class, each method and each instance/class variable you declare. You must also comment any piece of code that is not obvious what it does.
- Visibility: All fields should be made **private** and all methods provided in the specification are public. You may add any fields or methods you want as long as they are **private**.
- Packages: You may not import any package (or use the fully-qualified name) except the following ones: **`Scanner`**, **`File`**, **`ArrayList`**, **`Iterator`**, **`NoSuchElementException`**, `Random`, `FileNotFoundException`

- Restrictions: You may **not** use lambda expressions or any language construct that hasn't been covered in class.

Submission

Submission instructions are as follows:

1. Upload all the source files (`*.java`) to Gradescope using the following link. Do **not** zip the files!

   https://www.gradescope.com/courses/498183/assignments/2844330

2. Download the files you just uploaded to Gradescope and compile/run them to verify that the upload was complete and correct.
3. Make a backup of your files on OneDrive using your GMU account

   If you skip steps 2 and 3 and your submission is missing the proper files, there won't be a way to verify your work and you will get zero points.

Grading

- Grading will be primarily automated with the use of unit tests.
- Manual grading will be used only for checking comments, hard-coding, violations, and the GUI
- **If your code doesn't compile with the compliance checker provided in Gradescope, it is highly likely that it won't compile with the autograder either, in which case you will get zero points.**

Provided files

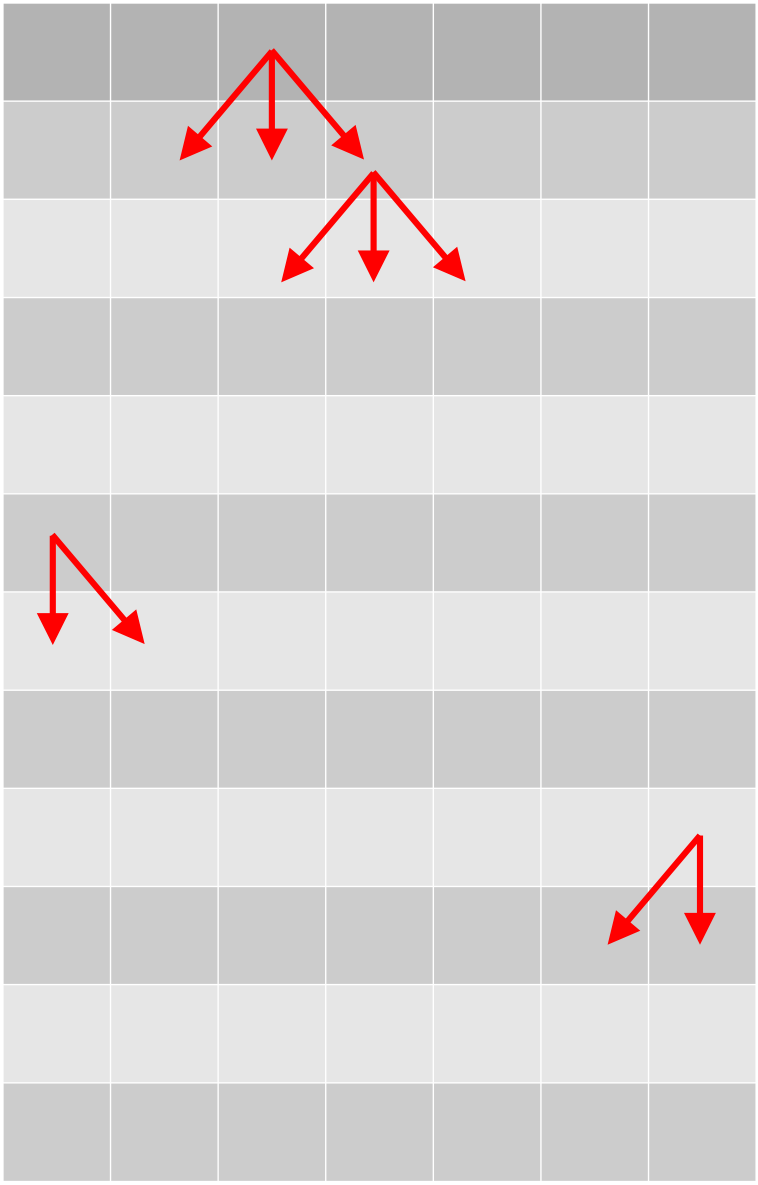
There are two files included in this project:

- Sample code for running and testing your implementation
- An example of a text file that can be used for the method **`loadDataFromFile`**

Description

Imagine that you have a rectangular football field (the dimensions can vary), and you want to find the best route to dash from one endline to the other endline at the opposite side of the field. We will simulate this field with a 2D matrix. In this matrix, some cells allow free movement and we call them "passages" and some cells are occupied by players of the opponent team and we can't cross them, so we call them "obstacles". Some passages are easier than others, so we will simulate this with a number of points that each passage is assigned with – the higher the number the easier it is to cross the passage. The goal is to find the easiest route from one endline to the other one. One endline is the first row of the matrix and the opposing endline is the last row of the matrix. You can start the dash from any position of the first endline (i.e. from any column of the first row) and you can finish in any position of the other

endline (i.e. in any column of the last row). During your dash, you can only move forward (you can't go backwards), and you can only move one cell at a time either straight ahead or diagonally. So, from each cell of the matrix, there are either 2 or 3 movement options only (see the figure below).



As you move from the first row to the last one, you collect the points of each passage you go through. Your goal is to follow a route that will give you the highest number of points.

Tasks

- You must implement the following classes based on the specification provided below.
- It is recommended that you work on them in the order provided.

Block

An abstract class that represents a single cell of the field. The **`Passage`** and the **`Obstacle`** classes will derive from it.

Methods

`public abstract int getValue()`

A placeholder method to be overridden by **`Passage`** and **`Obstacle`** classes.

Obstacle

It's a concrete class that represents a single cell of the field that is specifically an *obstacle*. It inherits from **`Block`**.

Methods

`public Obstacle(String label)`

A simple constructor

`public String toString()`

It returns the **`label`** of the object

`public int getValue()`

Since it's impossible to go through an obstacle, it is assigned **`0`** points.

Passage

It's a concrete class that represents a single cell of the field that is specifically a *passage*. It inherits from **`Block`**.

Methods

`public Passage(int value)`

A simple constructor

`public String toString()`

It returns the **`label`** of the object which is simply its value

`public int getValue()`

It returns the number of points assigned to the object when constructed.

FlexibleIterable

A generic interface that extends the **`Iterable`** interface. The Iterable is a functional interface, i.e. it has one method only, and we want to overload this method. So, to do this, we need to create a new interface that extends the Iterable in order to provide this additional method.

Methods

`public Iterator<T> iterator(String iterableObjectName)`

Creates an iterator that iterates only on objects whose datatype name is **`iterableObjectName`** and skips everything else in the iterable object. For example, if the value of the `iterableObjectName` is "Obstacle", this iterator will iterate only over the Obstacle cells of the field. Similarly, if the value of the `iterableObjectName` is "Passage", the iterator will iterate only over the Passage cells of the field.

If you have an object named **`obj`**, you can use the following code to get the name of its datatype:

`obj.getClass().getName()`

Field

A generic class that represents a field. It implements the **`FlexibleIterable`** interface. Use the following signature:

`public class Field<T> implements FlexibleIterable<T>`

Fields

All fields are private. The following is the minimum; feel free to add more fields if you want.

`private T[][] matrix`

A 2D array that can hold any generic type of objects

Methods

`Field(int height, int width)`

The constructor allocates memory for the **`matrix`** . All cells have the value of **`null`** after the constructor completes.

`public T getElement(int row, int col)`

Getter method for the **`matrix`** elements

`public void setElement(int row, int col, T el)`

Setter method for the **`matrix`** elements

`public int getHeight()`

Returns the height of the rectangular field

`public int getWidth()`

Returns the width of the rectangular field

`public String toString()`

The string representations of all the matrix elements are merged into a single string. Use a single-space separator between the elements and a newline character at the end of each row. No added leading or trailing spaces.

`public Iterator<T> iterator()`

The default iterator. It must be implemented as an anonymous inner class. In this inner class, you must fully implement the **`hasNext`** and **`next`** methods. As for the **`remove`** method, put nothing but the following line of code in its body:

```
throw new UnsupportedOperationException();
```

Check the `Iterator` interface for more details on the methods you must implement.

`public Iterator<T> iterator(String iterableObjectName)`

This is the overloaded iterator method as explained in the FlexibleIterable interface. The only thing that this method does is to create and return a new **`FieldIterator`** object. So, the body of the method should have one line of code only. Obviously, this iterator is not implemented as an anonymous inner class but as a separate class.

FieldIterator

A generic class that provides an iterator for the **`Field`**. It implements the **`Iterator`** interface. Use the following signature:

```
public class FieldIterator<T> implements Iterator<T>
```

The purpose of this iterator is to provide an alternative to the default iteration. Based on the value of the argument `iterableObjectName` this iterator will not iterate over the entire `Field` but it will restrict itself to Blocks of specific datatype, e.g. `Passage` or `Obstacle`.

Fields

All fields are private. Feel free to add any fields you want. However, **you may not use an array or a List to create a temporary storage for the contents of the `Field` class**; you must call directly the public methods of the **`Field`** class anytime you want access to its contents.

Methods

`constructor`

The signature is your call. This is because we don't instantiate a `FieldIterator` by calling the constructor directly, instead we call the `iterator(String iterableObjectName)` method of the Field object (see the sample code for an example). Thus, it's your call how to design the

constructor and how to invoke it from within the above **`iterator`** method.

`next`

The bulk of the work of this class will be in this method. In addition to everything else you do in this method, you must also take care of the case that someone tries to invoke this method when the iteration is over and there aren't anymore items remaining. In this case, the method should raise an `NoSuchElementException`

`hasNext`

Returns `true` or `false` depending on whether or not there exist more elements on the Field to iterate on.

`remove`

Do not provide an implementation. Just put the following code in the body:

```
throw new UnsupportedOperationException();
```

FieldGenerator

A utility class that provides two methods for generating a **`Field`**.

Methods

`public static Field<Block> loadDataFromFile(String filename)`

It generates a Field based on the data that is stored in a text file (see the provided sample). Passages are represented with an integer that corresponds to the points assigned to each one of them, while Obstacles are represented with a **`-`** character.

`public static Field<Block> randomIntegers(int h, int w, int l, int m, int n)`

It generates a Field where passages and obstacles have random placements.

h is the height of the Field

w is the width of the Field

l is the lowest random number of points that a `Passage` can have

m is the largest random number of points that a `Passage` can have

n is the number of `Obstacle` objects in the Field.

If parameter ***n*** is large w.r.t. the parameter ***w***, there is no guarantee that the generated Field can be crossed. We will not test the methods `bestStartingPoint` and `bestRoute` with such fields that a solution is not warranted.

It's recommended to use the **`java.util.Random`** class instead of the `Math.random` method because it gives you more options in terms of constructing random numbers. Also, very important, by using the overloaded constructor of the **`Random`** class, you control the **`seed`** of the sequence of pseudorandom numbers generated, and this is very useful when you do debugging because you don't want to get a different sequence of random numbers every time you test your code. Of course, you want to remove the seed when code goes to production.

Game

This class contains the **`main`** method that you can use to run and test your application (see the sample code for more details), and two methods for solving the task of finding the best route from one endline to the other.

Methods

`public static TwoValues bestStartingPoint(Field<Block> board)`

You will need recursion to solve this task.

It finds the best starting point, i.e. what column of the first row of the Field we must use in order to collect the highest number of points while crossing the Field. It also calculates the sum of these points.

Since Java methods don't support the return of more than one value, we package these two calculated quantities into an inner class named **`TwoValues`** (see the sample code).

`public static ArrayList<Block> bestRoute(Field<Block> board, int col)`

Do not attempt this method before you complete the previous one. This one requires recursion too.

Given a starting point (i.e. a column index in row 0) it returns the list of Blocks that form the best route from one endline to the other one.

GUI

This is an optional task. If you decide to implement a Graphical User Interface class, you must name it **`GUI`** and you must still implement the **`Game`** class. This means that one should be able to execute your program either as a text-based application by running the **`Game`** class or as a graphical application by running the **`GUI`** class.

This class should provide at a minimum the following functionalities:

1. Select a file to open and load data
2. Select what kind of iterations to run. Be reminded that there are three types (entire Field, Obstacles only, Passages only).
3. Display the result of the iteration
4. Run the recursion and display the result graphically

You're free to add more functionalities if you want of course.

There are no restrictions on what fields, methods, and packages you use in this class but your code must adhere to the OOP principles.