## Project 01: Java Introduction

This project will introduce you to programming in Java by asking you to develop an abstract data type for graphs, and then adapt it for a slightly different interface.

A [graph (Links to an external site.)](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)) consists of a set of *nodes* (or *vertices*) *N* and a set of edges *E ⊆ N × N*. For this project, graphs are *directed*, meaning there could be an edge from *a* to *b* without there being an edge from *b* to *a*.

We will define graphs as an abstract data type that implements the following interface:

public interface Graph {

boolean addNode(String n);

boolean addEdge(String n1, String n2);

boolean hasNode(String n);

boolean hasEdge(String n1, String n2);

boolean removeNode(String n);

boolean removeEdge(String n1, String n2);

List<String> nodes();

List<String> succ(String n);

List<String> pred(String n);

Graph union(Graph g);

Graph subGraph(Set<String> nodes);

boolean connected(String n1, String n2);

}

Here, nodes are labeled by strings, and if two strings are equals then they always refer to the same node. The methods do the following:

* boolean addNode(String n) adds the node n to the graph. It returns true if the node was not previously in the graph (i.e., it was added by the call), and false if the node was already present.
* boolean addEdge(String n1, String n2) adds an edge from the node n1 to the node n2. It returns true if the edge was not previously in the graph, and false otherwise. This method should throw NoSuchElementException if n1 or n2 were not previously added as nodes.
* boolean hasNode(String n) returns true if the node n was added to the graph previously (and not removed), and false if not.
* boolean hasEdge(String n1, String n2) returns true if the edge from n1 to n2 was added to the graph previously (and not removed), and false if not.
* boolean removeNode(String n) removes node n from the graph and all edges to or from n. It returns true if n was previously in the graph and false otherwise.
* boolean removeEdge(String n1, String n2) removes the edge from n1 to n2 from the graph, returning true if the edge was previously in the graph and false otherwise. This method should throw NoSuchElementException if n1 or n2 were not previously added as nodes.
* List<String> nodes() return a list containing all the nodes in the current graph, in some unspecified order.
* List <String> succ(String n) returns a list of all nodes n2 such that there is an edge from n to n2 in the graph, i.e., it returns the *successors* of n. This method type uses the [List (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html) interface. This is in a *generic* type, meaning you can have lists of strings, list of integers, lists of lists of strings, etc. We'll go into detail about how this works later, but for now, all you need to know is that List<String> is a list of strings, and in the documentation, anywhere you see the *type parameter* E you can mentally substitute String. This method should throw NoSuchElementException if n was not previously added as a node.
* List <String> pred(String n) returns a list (a List<String>) of all nodes n2 such that there is an edge from n2 to n in the graph, i.e., it returns the *predecessors* of n. This method should throw NoSuchElementException if n was not previously added as a node.
* Graph union(Graph g) returns a new graph that includes all the nodes and edges of the current graph and all the nodes and edges of g. Nodes identified by the same string in both graphs are coalesced to be the same node in the returned graph. *Note:* You may **not** assume that g is implemented by an ListGraph, i.e., code that casts g to an ListGraph will receive no credit. This requirement means this method will be extremely annoying to write, which sometimes happens when interfaces and APIs are not ideal.
* Graph subGraph(Set<String> nodes) returns a new graph containing the nodes of the current graph that are included in nodes and the current graph's edges among them. For example, if the current graph has nodes [A,B,C,D] and edges A→B, A→C, C→D and the nodes argument to subGraph is [A, B, C, E], then the resulting graph would contain edges A→B and A→C, and nodes A, B, and C.
* boolean connected(String n1, String n2) returns true if and only if there is a *path* from n1 to n2, meaning there is a sequence of edges from n1 to some na to some nb etc to n2. If n1.equals(n2), this method should return true, i.e., a path of length 0 counts. To implement this method, you'll probably want to use either [breadth-first search (Links to an external site.)](https://en.wikipedia.org/wiki/Breadth-first_search) or [depth-first search (Links to an external site.)](https://en.wikipedia.org/wiki/Depth-first_search) (either will work). For this method to work correctly in the presence of cycles in the graph (i.e., the case when one node is connected to itself), you might want to use a [HashSet<String> (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html" \t "_blank). This method should throw NoSuchElementException if n1 or n2 were not previously added as nodes.

Here is the [code skeleton](https://canvas.tufts.edu/courses/37277/files/4123644?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4123644/download?download_frd=1)for the project.

Part 1: Graphs with Adjacency Lists

A key algorithmic design choice in implementing graphs is how to represent edges in the graph. For the first part of the project, you will implement the Graph interface using *adjacency lists*. More specifically, write an implementation of Graph in the file ListGraph.java in which the nodes and edges of the graph are represented using the following field:

private HashMap<String, LinkedList<String>> nodes;

Here, the graph is represented as a mapping from nodes to lists of their successors in the graph. The map itself is a hash table (a [HashMap (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html)), and the lists are [LinkedList (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/LinkedList.html)s, which are just like linked lists in C.

For example, here is some code that creates a few nodes and edges in the graph and does some tests to see what the graph contains.

nodes.put("a", new LinkedList<String>()); // add node "a" to the graph

nodes.put("b", new LinkedList<String>()); // add node "b"

nodes.put("c", new LinkedList<String>()); // add node "c"

nodes.containsKey("a"); // returns true, "a" is a graph node

nodes.containsKey("d"); // returns false, "d" is not a graph node

nodes.get("a").add("b"); // add edge from "a" to "b"

nodes.get("c").add("a"); // add edge from "c" to "a"

nodes.containsKey("c") &&

nodes.get("c").contains("a") &&

nodes.containsKey("a") &&

nodes.get("a").contains("b") // returns true, there's a path from "c" to "b"

*Hint:* If you want to iterate through a LinkedList in Java, you can use a while loop, or you can use the following syntactic sugar; we'll explain why this works later.

for (String n : nodes.get("a")) { // iterate through elements of nodes.get("a")

// code that uses n

}

If you want to iterate over a HashMap, you can do the following:

for (String n : nodes.keySet()) { // iterate over key of HashMap

LinkedList<String> s = nodes.get(n); // get corresponding successor lists

}

It is also possible to avoid the call to get by iterating over the entrySet of the HashMap. If you're interested, you can find out how to use that approach by searching online.

Part 2: A Different Graph API

The Graph interface we gave you above is just one possible interface for graphs. For example, here is a class that implements an immutable representation of graph edges:

public class Edge {

private String src, dst; // source, destination

Edge(String src, String dst) {

this.src = src; this.dst = dst;

}

String getSrc() { return src; }

String getDst() { return dst; }

}

We can then use this class to define a different interface for graphs:

public interface EdgeGraph {

boolean addEdge(Edge e);

boolean hasNode(String n);

boolean hasEdge(Edge e);

boolean removeEdge(Edge e);

List<Edge> outEdges(String n);

List<Edge> inEdges(String n);

List<Edge> edges();

EdgeGraph union(EdgeGraph g);

boolean hasPath(List<Edge> l);

}

with the following methods:

* boolean addEdge(Edge e) adds an edge to the graph, returning true if the edge was not already in the graph or false if not. Note that, in this API, nodes are not added separately from edges. Node n is automatically added to the graph if an edge containing n is added.
* boolean hasNode(String n) returns true if and only if some edge has been added to the graph (and not removed) with n as either the source or destination fo the edge.
* boolean hasEdge(Edge e) returns true if edge e has been added to the graph (and not removed).
* boolean removeEdge(Edge e) removed edge e from the graph, returning true if it was previously in the graph and false otherwise. If this method removes the last edge to or from a given node in the graph, that node should also be removed. Note: This method does not throw an exception even if one or the other end of the Edge is not in the graph.
* List<Edge> outEdges(String n) returns a list of all edges that start at node n.
* List<Edge> inEdges(String n) returns a list of all edges that end at node n.
* List<Edge> edges() returns a list of all the edges in this graph, in some unspecified order.
* EdgeGraph union(EdgeGraph g) returns a new graph that includes all the nodes and edges of the current graph and all the nodes and edges of g. Nodes identified by the same string in both graphs are coalesced to be the same node in the returned graph. *Note:* You may **not** assume that g is implemented by an EdgeGraphAdapter, i.e., code that casts g to an EdgeGraphAdapter will receive no credit.
* boolean hasPath(List<Edge> l) returns true if the path l (a List<Edge>) is in the graph, meaning if l = e1, e2, ..., en then all edges ei are in the graph. The method should also check if the argument is a path, i.e., if ei.getDst() == e(i+1).getSrc() for every edge in the middle of the list. If this is not the case, this code should raise the exception BadPath, which is defined in file BadPath.java. Note that every graph includes the empty path (since if a graph contains a path, it should contain every sub-path).

Your task for the second part of the project is to implement an EdgeGraph. But, like any good software engineer, you don't want to start from scratch. You already have perfectly good Graph implementation. So, for this part of the project, you will write an [*adapter* (Links to an external site.)](https://en.wikipedia.org/wiki/Adapter_pattern) that, given a Graph, will adapt it to be an EdgeGraph. More specifically, implement EdgeGraphAdapter, which looks like the following:

public class EdgeGraphAdapter implements EdgeGraph {

private Graph g;

EdgeGraphAdapter(Graph g) { this.g = g; }

// methods of EdgeGraph

}

So, to implement the methods of EdgeGraphAdapter, you will write code that *delegates* graph operations to g. You'll notice that for some methods of EdgeGraph, you can call corresponding methods of g with no change. With other methods, you'll need to change the arguments a bit. And with still other methods, you'll need to implement new functionality that's not part of Graph.

Notice also that your code will work with *any* implementation of Graph, not just ListGraph. Cool!

Part 3: Write and Share a Test Case

To help you test your code, we've created a simple method main in class Main so you can run some test cases. The body of main looks like this:

public static void main(String[] args) {

test1();

test2();

}

It just calls a couple of simple tests we wrote. Though it won't count in the grading of your project, you should right a bunch more tests (test3, test4, etc., or call them whatever you want since we won't evaluate these methods) and add calls to them in main.

Here is the definition of test1:

public static void test1() {

Graph g = new ListGraph();

assert g.addNode("a");

assert g.hasNode("a");

}

There are a bunch of ways to write and design tests, which we'll talk about later in the semester, but this method just creates a new graph, adds a node to it, and checks that the node is in the graph. To check that methods are returning the correct values, it uses Java's assert statement, which takes a boolean argument and raises an exception if the argument doesn't evaluate to true.

**Important:** To run the code with assertions enabled, you need to invoke java -ea Main, i.e., you need to pass the -ea (or -enableassertions) argument to the JVM. Otherwise, by default, the JVM will ignore the assertion statements completely and not run them.

Although you can't generally share code for this class, we will make one exception for this project: You **must** write one test case, in the style of test1 above, and post it publicly to Piazza to share with the class. **Your test case must be for Part 1 only.** You may not share test cases for Part 2. Your test case can test any functionality of Part 1, as much or as little as you like. It need not be substantively different than others' test cases, but you must come up with it on your own. In fact, you should come up with a test case (or many test cases) right now, before you've even started implementing the project. This is called [Test-driven development (Links to an external site.)](https://en.wikipedia.org/wiki/Test-driven_development), which is a popular software engineering approach.

What to turn in

Put all your code in the code skeleton we have supplied, and upload your solution using [Gradescope (Links to an external site.)](https://www.gradescope.com/" \t "_blank). **Important:** Be sure all of your .java files are at the top level in your submission. If you submit them inside of a directory, the autograder won't find your code and compilation will fail. For this and all future projects, you may not change any public API we specify, including changing the list of exceptions that may be thrown by a method. Also for this and all future projects, unless otherwise specified, your code may only use standard Java libraries; it may not use any libraries that require changing the CLASSPATH or compilation options.

[Previous](https://canvas.tufts.edu/courses/37277/modules/items/665869)[Next](https://canvas.tufts.edu/courses/37277/modules/items/665871)

**Submission**

[Submission Details](https://canvas.tufts.edu/courses/37277/assignments/223598/submissions/80402)

Grade: 104 (106 pts possible)

Graded Anonymously: no

**Comments:**

No Comments

## Project 02: Design Patterns in Java

Reminder: you'll need to submit your test case for this project no later than **11:59 pm ET on Monday February 14.**

Introduction

In this project, you will implement a chess game simulator. Your simulator will take as input a file that describes the initial locations of pieces on the board and a series of moves of those pieces. Your simulator will then set up the board and execute those moves, checking that all the moves are legal and, at the end, printing the state of the game board.

The real goal of implementing this project, however, is to gain experience with design patterns. More specifically, as part of this project you will implement the following patterns: Singleton, Factory, External Dependency Injection, Observer, and (Internal) Iterator. Here is a [code skeleton](https://canvas.tufts.edu/courses/37277/files/4270561?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4270561/download?download_frd=1)to get you started.

This project will also give you a little practice with the essential software engineering skill of solving programming problems by finding library code to do what you want. We'll give you some hints, but fewer than in the last project. So you'll need to spend some time searching on Google and poking around the [JDK 16 API (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/index.html). Most of the methods you want are probably in the following:

* [java.base (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/module-summary.html)
  + [java.io (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/io/package-summary.html)
  + [java.lang (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/package-summary.html)
  + [java.util (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/package-summary.html)
  + [java.nio (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/package-summary.html)

But, if you want to get fancy, you can use other parts of the API, e.g., [java.util.regex (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/regex/package-summary.html" \t "_blank). In this and all other projects, you are allowed to use any part of the JDK 16 API you like.

This project doesn't break down into separate steps quite the way the previous one does, but it does break down into modules. So this writeup is organized as a description of the modules you'll need to write. The description is written in an order that's sensible for presentation, but you can implement the different modules in whatever order you want.

**Important:** Do not add throws clauses or otherwise change any of the public API methods you are implement, or the autograder will not compile against your code!

*Hint:* The hardest part of the project, in a technical sense, is writing the code to model the moves of all the chess pieces. The rest of the project has much less code, but you will need some time to understand all the design patterns we've crammed into the project. And yes, there are too many design patterns here, but hopefully not as many as this [factorial implementation that's been design patterned to death. (Links to an external site.)](https://chaosinmotion.blog/2011/01/25/how-not-to-write-factorial-in-java/)

*Note:* You are free to add any helper methods, additional classes, etc, that you'd like to for this project. Just be sure to upload a complete, working program to Gradescope.

Overview

Recall that the game of chess is played on and eight-by-eight board with the following initial layout:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** |
| **8** | ♜ br | ♞ bn | ♝ bb | ♛ bq | ♚ bk | ♝ bb | ♞ bn | ♜ br |
| **7** | ♟ bp | ♟ bp | ♟ bp | ♟ bp | ♟ bp | ♟ bp | ♟ bp | ♟ bp |
| **6** |  |  |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |  |  |
| **3** |  |  |  |  |  |  |  |  |
| **2** | ♙ wp | ♙ wp | ♙ wp | ♙ wp | ♙ wp | ♙ wp | ♙ wp | ♙ wp |
| **1** | ♖ wr | ♘ wn | ♗ wb | ♕ wq | ♔ wk | ♗ wb | ♘ wn | ♖ wr |

Using standard notation, we've labeled the chess board's rows 1-8 and the columns a-h, and using slightly non-standard notation we're describing each piece with two characters: the color (black (b) or white (w)) and the kind (king (k), queen (q), knight (n), bishop (b), rook (r), and pawn (p)).

Your simulator will be run via the command

java Chess layout moves

where layout is the name of a file describing the initial locations of pieces on the chessboard, and moves is the name of a file describing a sequence of moves. We've given you one example each of the layout and move files, layout1 and moves1, respectively, where layout1 contains the standard setup of chess pieces on the board. For ideas of other ways you could lay out chess pieces initially, search the web for *chess puzzles*or [chess problems (Links to an external site.)](https://en.wikipedia.org/wiki/Chess_problem).

Parsing Layout and Move Files

You will need to write code in Chess.java that (a) reads data from the layout file and sets up the board and then (b) plays the sequence of moves in the moves files. Of course, you can create whatever additional methods and classes you need.

If you look in Chess.java, you'll see a main method of the usual type. You'll want to add your code toward the bottom of this method. Notice that the names of the files you need to read are given on the command line, which means they will be stored in the args parameter of main. The layout file name will be in args[0], and the moves file name will be in args[1].

To write this code, you'll need to figure out how to open and read files in Java. We won't tell you how to do this, but if you search the web, look in the JDK documentation, and/or look through the Java textbooks linked from the class web page (see the [resourcesLinks to an external site.](https://www.cs.tufts.edu/comp/180/" \l "resources" \t "_blank) at the bottom of the page), you can figure it out.

There are actually a few different ways to access files in Java  (*hint: the "newer" ways to do it may be slightly easier than the older ways to do it)*. You just have to find one that works. Please don't ask your fellow students for help with this. Pretend that you're working at a company, you need to work with files in Java, and none of your colleagues knows how to do it. This kind of situation happens often, so now is a good time to practice finding the information on your own!

As you process the layout and moves files, you **must enforce the following rules about the file**. If any of the rules is violated, throw an exception (any exception will do):

* **Layouts**
  + x is a column (a-h)
  + n is a row (1-8)
  + c is a color (b or w)
  + p is a piece kind (k, q, n, b, r, or p)
  + A layout file is a possibly empty sequence of lines
  + If a line begins (0th character) with a #, that line is a comment and should be ignored
  + Otherwise, a line should have the form xn=cp, indicating that position xn starts out with piece cp, where
  + All files we supply will not have any extra whitespace at the beginning or end of a line, so you can allow or disallow extra whitespace as you like.
  + A layout file may not place two pieces in the same location.
* **Moves**
  + A moves file is a possibly empty sequence of lines
  + If a line begins (0th character) with a #, that line is a comment and should be ignored
  + Otherwise, a line should have the form xn-ym, indicating that the piece at position xn moves to position ym, where
    - x and y are columns (a-h)
    - n and m are rows (1-8)
  + All files we supply will not have any extra whitespace at the beginning or end of a line.
  + All moves must be valid according to a subset of the chess rules we discuss below.

Chess Pieces

There are six classes representing different kinds of chess pieces: King, Queen, Knight, Bishop, Rook, and Pawn. Each of these classes is a subclass of Piece, which is a class rather than an interface because it has some code in it. You need to add code to all seven of these classes (the six chess piece classes and their superclass) to match the following design:

* Each chess piece has a color, which is either Color.BLACK or Color.WHITE. This typesafe enum is defined in Color.java. The color is passed to the piece's constructor. You should implement the color() method in Piece, rather than in the subclasses. To do so, you'll probably want to add a field to Piece.
* Each piece's toString() method should return the piece name in the form it appears in the layout file, e.g., a black king should return bk, a white pawn should return wp.
* Each piece has a method List<String> moves(Board b, String loc) that returns a list containing the locations the piece could legally move to starting from the position loc, taking the positions of other pieces on the board into account. (However, there should **not** be any check by this method that the piece is actually at position loc.) The loc is given in the same notation as the layout file, e.g., "a3" is a location. Each piece has a different set of moves. We will use the following movement rules only, which are slightly simplified from the [rules of chess (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Basic_moves):
  + The king can move to one adjacent space in any direction (left/right, up/down, or diagonal) and may or may not capture a piece at the end of its move. We will not worry about [check (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Check) or [checkmate (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Checkmate), and we will not implement [castling (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Castling).
  + The queen can move any number of vacant spaces in any direction, left/right, up/down, or diagonal. The queen can stop moving before capturing a piece, or can capture a piece at the end of the move.
  + The knight moves in an L-shape: two squares horizontally and then one square vertically, or two squares vertically and one square horizontally. See the [rules of chess wikipedia page (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Basic_moves) for a picture. The knight jumps over other pieces. It can either land on an unoccupied space, or it can capture a piece by landing on an occupied space.
  + The bishop is like a queen that can only move diagonally. It moves any number of vacant spaces along any diagonal, and may or may not capture a piece at the end of its move.
  + The rook is like a queen that can only move horizontally or vertically. It moves any number of vacant spaces either horizontally or vertically, and may or may not capture a piece at the end of its move. We will not implement [castling (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Castling).
  + The pawn can move one space vertically forward, only toward the opponent's side of the board, i.e., a black pawn can move toward row 1, and a white pawn can move toward row 8. However, the pawn can move two spaces forward if it is in its home row (row 2 for white, row 7 for black) and the intervening space is vacant. The preceding two moves can only be made to vacant spaces; they cannot capture pieces. A pawn can capture an opponent's piece by moving one square diagonally toward the opponent. It cannot move diagonally if it is not capturing a piece. We will not implement *[en passant](https://en.wikipedia.org/wiki/Rules_of_chess" \l "En_passant" \t "_blank)*[(Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess" \l "En_passant" \t "_blank) or [pawn promotion (Links to an external site.)](https://en.wikipedia.org/wiki/Rules_of_chess#Pawn_promotion).
  + Pieces can only capture the opponent's pieces (i.e., of the opposite color).

Aside: Notice that Piece is an abstract class, with two abstract methods. This means that Piece is somewhere between an interface an a class: It has some methods (and possibly fields) that are inherited by subclasses, and it has some methods that must be implemented by subclasses.

Chess Piece Factories

Given a piece's symbol (like "r"), your code will need to construct an instance of the corresponding piece (like Rook). You could implement this using a switch statement, but then the set of piece kinds would be somewhat hard-coded in your program, making it harder to add more kinds of pieces (something we won't do in this project, but we can envision doing). Instead, we will use the factory pattern to create pieces. Your code will store a mapping from piece symbols to objects containing the appropriate factory method. Then, when asked to create a piece, your code will look up the piece symbol in the map and call the corresponding method. Here are the details of the design:

* Implement void Piece#registerPiece(PieceFactory pf)so it stores (in a mapping you will need to create) a map from the symbol of the PieceFactory to the PieceFactroy object. You'll notice that in Chess.java, there is a sequence of calls that use this method, e.g., calling Piece.registerPiece(new PawnFactory()) etc. for all kinds of pieces that currently exist. If we decided to add a new kind of piece, we would just need to register that new piece kind and not change anything else!
* For each subclass *C* of Piece, there is a corresponding class *C*Factory that has two methods (which we've written for you): char symbol(), which returns the piece's one-letter symbol (e.g., 'p' for Pawn) and Piece create(Color c), which returns a new instance of the corresponding piece with the given color, e.g., PieceFactory p = new PawnFactory(); p.create(Color.WHITE) returns a new white pawn. So you don't need to write the actual factories.
* Implement method Piece Piece.createPiece(String name) to create an instance of the piece described by its argument in the same format as the layout file, e.g., Piece.createPiece("br") should look up "r" in the map, call the resulting constructor, and return a black rook. If you do everything right, then if we add a new kind of piece the method will still work!
* (You do need to implement the constructors of the underlying chess piece classes, e.g., you need to implement the Pawn(Color c) constructor.)

Whew, that's pretty complicated! And probably unnecessary for chess. But again, it does have the nice (?) feature that it would be easy to create new kinds of chess pieces and add them to the board without having to change too much code.

The Chess Board

Next up, you need to write code for class Board, which stores the locations of the pieces, among other things. To give you practice with another pattern, we've decided that the Board should be a singleton class. Hence you need to implement a method theBoard that returns the singleton Board instance.

The board itself stores the piece locations in field pieces, which is a two-dimensional array of Piece. Notice that, very often, you will need to convert coordinates like "a3" into an access into this array. It's up to you how you do this. You could implement a full-scale adapter pattern. You could write a utility method or two to convert. Or you could duplicate code all over the place. Only the last choice is not recommended!

You should implement four *mutators* for Board:

* Piece getPiece(String loc) returns the piece at the given location (in the usual notation, e.g., "a3"), or null if the location is empty. It should raise an exception (any exception) if loc is invalid.
* void addPiece(Piece p, String loc) adds piece p to the board at the given location. It should raise an exception (any exception) if the location is already occupied or is invalid.
* void movePiece(String from, String to) moves the piece at location from to location to. This method should check that there is a piece at from (and throw an exception if not), and it should check that the move is valid for that piece. If the move is valid, then location from becomes vacant and the piece is placed at position to. It is possible there was another piece at to, in which case it has been captured; see below. If the move is invalid or from or to are invalid locations, movePiece should raise an exception (any exception).
* void clear() should remove all pieces from the board.

The Chess Board Observer

The Board class also supports the observer pattern, so that listeners can be called back when key events happen. More precisely, a BoardListener is an observer that implements two methods: void onMove(String from, String to, Piece p), which is called whenever a move is made on a board; and void onCapture(Piece attacker, Piece captured), which is called whenever a piece is captured. If a piece is captured, there should be a call to onMove first and then a call to onCapture.

You must implement the observer pattern by modifying Board.java as follows:

* void registerListener(BoardListener bl) should add a listener that should subsequently be called at the appropriate times. (So you will need to modify movePiece also.) There can be any number of listeners at any time (zero or more). We will not test the case of the same listener being registered more than once.
* void removeListener(BoardListener bl) removes a listener so it will subsequently not be notified of events. Your code can behave however you like at an attempt to remove a listener that wasn't registered, or an attempt to remove a listener that was registered twice.
* void removeAllListeners() removes all listeners.

In Chess.java, the main method registers a simple observer that listens for updates to the board and prints out what happened. This could be helpful for debugging.

The Chess Board Iterator

But wait, there's more! (We promise this is the last part...) The Board class also supports internal iteration. The BoardInternalIterator interface (sorry for the terribly long name) defines a method void visit(String loc, Piece p). You must implement the iterate method of Board so that it takes a BoardInternalIterator and, for *every square* on the board, calls the internal iterator's visit method, passing that square's location and the piece at the location, or null if the location is vacant. The iteration order is up to you, but it must visit every square of the board. (So, visit will be called 64 times.)

Testing Your Code

To help you test your code, we've given you a file Test.java with another main method that, like the previous project, runs a test case. You can invoke the tests with

java -ea Test

We've included one sample test case. (Note this test case is a little advanced; you'll have to do a fair amount of work before it passes.)

As part of this project, **you must write a test case for one Piece's moves method and share it with the class on Piazza**. More specifically, your test case should set up a board by calling addPiece some number of times and then call one appropriate moves method, testing the result. **Be sure your test case ignores the order of moves in the returned list** because different implementations might return moves in different orders.

Of course, you will want to write many other test cases for different parts of the project!

[Previous](https://canvas.tufts.edu/courses/37277/modules/items/665891)[Next](https://canvas.tufts.edu/courses/37277/modules/items/665893)

**Submission**

[Submission Details](https://canvas.tufts.edu/courses/37277/assignments/223599/submissions/80402)

Grade: 99 (99 pts possible)

Graded Anonymously: no

**Comments:**

No Comments

Project 03: Testing

Introduction

In this project, you will develop a number of useful testing framework components. First, you will implement your own version of JUnit. Second, you will extend your JUnit to support [QuickCheck (Links to an external site.)](https://en.wikipedia.org/wiki/QuickCheck" \t "_blank)-style random test generation (though to make this gradable we won't use randomness).

You will undoubtedly need to use [java.lang.reflect (Links to an external site.)](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/reflect/package-summary.html" \t "_blank) in this project. Moreover, be sure to look pay attention not only to the classes and interfaces you find when you click links in the documentation, but also subclasses and subinterfaces. At certain places in this project, the documentation will say you will get an X back, and if you check, you'll discover that at run-time you actually have a Y that implements or subclasses X, and Y is really what you need.

Note that **there are no shared test cases for this project**.

Here is a code skeleton for this project: [p3.zip](https://canvas.tufts.edu/courses/37277/files/4098821?wrap=1)[Download p3.zip](https://canvas.tufts.edu/courses/37277/files/4098821/download?download_frd=1)

Part 1: Implementing JUnit

Your first task is to implement the following method in class Unit:

public static HashMap<String, Throwable> testClass(String name);

Given a class name, this method should run all the test cases in that class. The return value is a map where the keys of the map are the test case names, and the values are either the exception or error thrown by a test case (indicating that test case failed) or null for test cases that passed.

Here are the rules your test execution engine should follow:

* Test cases are those methods annotated with @Test. We've defined this annotation for you already. Such methods take no arguments.
* Test cases are executed in alphabetical order (as defined by String's compareTo method).
* If there are any methods annotated as @BeforeClass, they should be executed once before any tests in the class are run. If there are multiple @BeforeClass methods, they are executed in alphabetical order.
* If there are any methods annotated as @Before, they should be run before each execution of a test method. Multiple @Before methods should be run in alphabetical order.
* Methods annotated @AfterClass and @After are analogous to @BeforeClass and @Before, except they are run after the test methods.
* The @BeforeClass and @AfterClass annotated methods should be run even if there are no @Test methods in the class. (But not @Before or @After methods.)
* @BeforeClass and @AfterClass can only appear on static methods (note this is different than JUnit). If they appear on an instance method, testClass should throw an exception (any exception).
* A method can have only one annotation among {@Test, @BeforeClass, @Before, @AfterClass, @After}. If a method has more than one such annotation, testClass should throw an exception (any exception).
* testClass should catch all throwables from invoking test methods and return them in its result. However, it should not catch any throwables raised in methods annotated as @BeforeClass, @Before, @AfterClass, or @After. (Such exceptions can be caught and then rethrown, wrapped in a runtime exception.)
* You can assume that any methods annotated with @Test, @Before, and @After are public instance methods that take no arguments. You can assume that @BeforeClass and @AfterClass only appear on public methods, and you should check that they only appear on static methods.

After implementing the core JUnit functionality, your next step is to implement a [fluent (Links to an external site.)](https://en.wikipedia.org/wiki/Fluent_interface) assertion API, in which conjunctions of assertions about objects are written using method chaining. For example, using this interface, we should be able to write

String s = ...;

Assertion.assertThat(s).isNotNull().startsWith("CS");

meaning that s is not null and begins with "CS".

We've started you off by creating a class Assertion defining several assertThat methods, but you'll need to create other classes to represent the result of calling the different assertThat methods. Here's how your interface should work:

* assertThat(Object o) should return an object that you can invoke the following methods on.
  + isNotNull() - raise an exception (any exception) if o is null, otherwise return an object such that more of the methods in this chain can be called. The rest of the methods have the same return pattern, so below we only state the conditions in which an exception is raised.
  + isNull() - raise exception if o is not null.
  + isEqualTo(Object o2) - raise exception if o is not .equals to o2.
  + isNotEqualTo(Object o2) - raise exception if o is .equals to o2.
  + isInstanceOf(Class c) - raise exception if o is not an instance of class c.
* assertThat(String s)
  + isNotNull(), isNull(), isEqualTo(o), and isNotEqualTo(o) as with assertThat(Object o). The isEqualTo(o) and isNotEqualTo(o) methods should take Object as an argument.
  + startsWith(String s2) raises an exception if s does not start with s2.
  + isEmpty() raises an exception if s is not the empty string
  + contains(String s2) raises an exception if s does not contain s2.
* assertThat(boolean b)
  + isEqualTo(boolean b2) raises an exception if b != b2.
  + isTrue() raises an exception if b is false.
  + isFalse() raises an exception if b is true.
* assertThat(int i)
  + isEqualTo(int i2) raises an exception if i != i2.
  + isLessThan(int i2) raises an exception if i >= i2.
  + isGreaterThan(int i2) raises an exception if i <= i2.

Part 2: QuickCheck for JUnit

QuickCheck is an automated program testing technique developed originally for Haskell. The idea of QuickCheck is that the programmer specifies a test case (called a *property*) that takes some parameters. The QuickCheck infrastructure runs that test case repeatedly with random choices of parameters. For example, using the implementation for this project, we will be able to write the following property:

@Property

public boolean absNonNeg(@IntRange(min=-10, max=10) Integer i) {

return Math.abs(i.intValue()) >= 0;

}

If such a property is defined inside a class whose name is passed to the Unit method

public static HashMap<String, Object[]> quickCheckClass(String name);

then your code will call absNonNeg with many different input integers ranging from -10 to 10, inclusive. For the first one for which absNonNeg returns false, quickCheckClass will add a mapping from the method name ("absNonNeg") to the array of arguments for which the method returned false or threw a Throwable. Otherwise, if the property runs until termination with only true return values, the "absNonNeg" will be mapped to null. Then quickCheckClass will run the next property in the class.

Here is the basic setup for quickCheckClass:

* It should run all the @Property methods in class name, in alphabetical order. Note that this is completely separate from test cases. You can assume, without checking, that no method is annotated as both @Test and @Property.
* A property is deemed to have failed if it returns false or throws any Throwable. Since you'll be invoking methods using reflection, to get the actual exception you'll probably need to unwrap an InvocationTargetException
* Unlike testClass, the quickCheckClass method returns a map that has the arguments that lead to the failure, but does not indicate what the failure was (i.e., it doesn't include a Throwable in the map).
* The first failing/throwing arguments to the property are stored in the map. After the first list of failing/throwing arguments is found, the property is not executed again.
* All properties are run **at most 100 times**. They may run fewer times depending on limits of the search, as described below.
* All property argument types must be annotated in some way, as follows.
  + Integer arguments must be annotated with @IntRange(min=i1, max=i2), indicating the minimum integer value and the maximum integer value (both inclusive) for the argument. You can assume max is greater than or equal to min.
  + String arguments must be annotated with @StringSet(strings={"s1", "s2", ...}), indicating the set of strings for the argument. You can assume the set of strings in non-empty, and that there are no duplicates in the list of strings.
  + List arguments must be annotated with @ListLength(min=i1, max=i2), indicating the minimum and maximum (inclusive) list lengths for the argument. The type T must itself be annotated appropriately for its range. For example, @ListLength(min=0, max=2) List<@IntRange(min=5, max=7) Integer> indicates an argument with lists of length 0 to 2 containing integers from 5 to 7, e.g., [], [5], [6], [7], [5,5], [5,6], [5,7], [6,5], [6,6], [6,7], [7,5], [7,6], [7,7].You can assume max is greater than or equal to min, and that the number of possible elements for T is greater than or equal to the number of elements specified by max.
  + Object arguments must be annotated with @ForAll(name="method", times=i), where method is the name of the (public, no argument, instance) method in the property's class that will be called to generate i values for the argument. (You can assume the method exists and that the times count is positive.) For example:
  + @Property public boolean testFoo(@ForAll(name="genIntSet", times=10) Object o) {
  + Set s = (Set) o;
  + s.add("foo");
  + return s.contains("foo");
  + }
  + int count = 0;
  + public Object genIntSet() {
  + Set s = new HashSet();
  + for (int i=0; i<count; i++) { s.add(i); }
  + count++;
  + return s;
  + }

* + No other argument types for properties are allowed. (Note that in the previous bullet point, Object really means an argument explicitly notated as Object, and not anything else.)

When running a property, your quickCheckClass method should call it will **all possible combinations of arguments** up to a maximum of 100 runs. The order in which you run the tests does not matter. (We will almost always be testing your code with properties for which there are fewer than 100 possible argument combinations.)

What to turn in

Put all your code in the [code skeleton](https://canvas.tufts.edu/courses/37277/files/4098821?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4098821/download?download_frd=1)[Links to an external site.](https://www.cs.tufts.edu/comp/121/p3.tar.gz) we have supplied, and upload your solution using Gradescope.

[Previous](https://canvas.tufts.edu/courses/37277/modules/items/665925)[Next](https://canvas.tufts.edu/courses/37277/modules/items/665927)

**Submission**

[Submission Details](https://canvas.tufts.edu/courses/37277/assignments/223602/submissions/80402)

Grade: 70 (97 pts possible)

Graded Anonymously: no

**Comments:**

No Comments

Project 04: Java on Rails

Reminder: you'll need to submit your test case for this project no later than **11:59 pm ET on April 4.**

Introduction

In this project you will implement a web server that uses a model-view-controller architecture. We'll call the system *jrails*, because it uses ideas from [Ruby on Rails (Links to an external site.)](https://rubyonrails.org/). However, because Java is, well, Java, the framework will be noticeably cruftier than Rails. On the other hand, we'll only implement a tiny fraction of what goes into a real web server framework.

The [code for this project](https://canvas.tufts.edu/courses/37277/files/4098778?wrap=1)[Download code for this project](https://canvas.tufts.edu/courses/37277/files/4098778/download?download_frd=1)is structured a bit differently than for previous projects. The MVC framework source code, which is what you'll be editing, is in the jrails/ directory. The code in that directory will be compiled to jrails.jar, which can then be added to the classpath when compiling a web app that uses the framework.

To make this process easier, we've given you a build setup for [gradle (Links to an external site.)](https://gradle.org/" \t "_blank), which is a popular and powerful build tool for Java. It's a bit complex to set up the build, but you shouldn't need to modify the build process at all. Here's how to use the build process:

* ./gradlew tasks - list all the build tasks
* ./gradlew classes - build jrails and the app
* ./gradlew run - run the app
* ./gradlew testClasses - build tests
* ./gradlew test - run tests

To help you more easily understand the structure of jrails, we've included an example app spread across four files in the books app directory: Book.java, BookController.java, BookView.java, and Main.java. (These files are in books/src/main/java/books.) Once you implement jrails, this particular app will let users keep track of a list of books. When you run

./gradlew run

you will eventually see something like this on the console:

> Task :books:run

Starting server...point your web browser to http://localhost:8000

<===========--> 87% EXECUTING [7s]

> :books:run

(Note that the text prints slightly out of order.) At this point, the web server has started on port 8000 on the local machine. You can connect to it by firing up a web browser and going to

http://localhost:8000/test

This should display a simple web page in your browser, and the console should show that the request was received. If you want to play around a bit more with the server, if you go to any other URL that's not routed (see below), the web server will dump some information about the request onto the console. For example, I noticed that when I go to a web page on this server, my web browser tries to get /favicon.ico in addition to the page I requested. It's fine for the server to just drop such requests on the floor.

**Important:** For this project, you need to upload a zip file of the entire p4 directory, so that directories like books/, gradle/, etc are at the top level of the directory. If you zip and your zip file's top-most file is the p4/ directory, that **will not work**. We recommend uploading your code early to make sure you aren't dealing with uploading issues at the last minute.

Also note that even though gradle makes it easy to include third-party libraries, your code must still only use the standard Java libraries.

Models

Like most MVC web app frameworks, jrails includes *models* that represent the database. More specifically, a model is any class that subclasses jrails.Model, which uses reflection to provide primitive database functionality. For example, here is the model from the book app:

import jrails.\*;

public class Book extends Model {

@Column public String title;

@Column public String author;

@Column public int num\_copies;

}

Because Book extends Model, conceptually there is a database table corresponding to the Book class. That table has three columns, title, author, and num\_copies, with the corresponding types String, String, and int, respectively.

Here is some code that uses the above model, with inline comments to explain what's happening:

Book b = new Book();

b.title = "Programming Languages: Build, Prove, and Compare";

b.author = "Norman Ramsey";

b.num\_copies = 999;

// The book b exists in memory but isn't saved to the db

b.save(); // now the book is in the db

b.num\_copies = 42; // the book in the db still has 999 copies

b.save(); // now the book in the db has 42 copies

Book b2 = new Book();

b2.title = "Programming Languages: Build, Prove, and Compare";

b2.author = "Norman Ramsey";

b2.num\_copies = 999; // hm, same as other book

b2.save(); // a second record is added to the database

assert(b.id() != b2.id()); // every row has a globally unique id (int) column, so we can tell them apart

Book b3 = Model.find(Book.class, 3); // finds the book with id 3 in the db, if any

List<Book> bs = Model.all(Book.class); // returns all books in the db

b.destroy(); // remove book b from db

Your job for this part of the project is to implement the Model class in jrails to achieve this functionality. Here are the details:

* In general, **jrails should throw an exception if the app code has an error.** (Warning: as a general principle, it's okay for web apps to show stack traces locally, but they should not send them in response to a request that crashes the server, since that could reveal information to an adversary.)
* Below, we'll write "model" to refer to a subclass of Model.
* A model class has zero or more public fields annotated with @Column. The only possible types for @Column fields are String, int, and boolean. It is an error for any other type to be used with @Column.
* When save is called on a model, you should write the contents of the model to a file on disk. This file will persist on disk, even between runs of your program. (The autograder will call a reset method, discussed below, to reset the file.) The exact name and format of this file is up to you. We won't look at it directly. I'd suggest a text file in comma-separated value format, where each line represents a database row. Be careful when writing strings to the file in case they themselves have commas. Note that a String @Column could be null, and null should be serialized differently than an empty string. It's up to you whether to have one global db file or several different ones for individual tables. You probably need some logic to create the db file if it does not already exist.
* Initially, a model that has been created but not saved has an id of 0. Just before a model is written to disk, is gets a globally unique id assigned to it, which can subsequently be accessed by calling the id() method of the object and is also written to disk. You'll need to keep track of ids on disk somewhere. If you prefer to have unique ids per table, that's okay too.
* When saveing a model with a zero id field, you should *add* a new row to the db file on disk. When saveing a model with a non-zero id field, you should *replace* the previous record in the database. It is an error to try to save a model with a non-zero id that is not already in the database. It will likely be somewhat annoying to implement replacement, because you'll need to change text in the middle of a file, which is always a bit awkward. You can safely assume that the entire db will fit in memory, though, so you can always read the whole file into some in-memory format and then write it out.
* The int id() method returns the id of the model.
* The <T> T find(Class<T> c, int id) method looks through the db for a row of the given id for the model c. If one exists, it *materializes* the row by creating a fresh instance of the model and initializing its fields according to the database entry. You can assume the model has only the no-argument constructor. If there is no db entry with the given id, then find returns null. The fancy type signature here means that find returns an instance of whatever class is passed as the first argument.
* The <T> List<T> all(Class<T> c) class method returns a List (possibly empty) of all the (materialized) db rows for the model. Similarly to find, the fancy type means all returns a list of instances of whatever class is passed as the first argument.
* Calling the void destroy() method removes the receiver from the db. If the receiver has not been saved to the db previously, this method raises an exception.
* Calling Model.reset() should remove all rows from the database. We will use this method to make grading your project easier.

Views

The *views* for jrails are [HTML (Links to an external site.)](https://en.wikipedia.org/wiki/HTML) pages that are sent back to the client web browser. Ruby on Rails uses HTML with embedded Ruby code, but developing such a system is a bit too complex for this project. Instead, we will create HTML by invoking methods in Java. More specifically, here is an example view from the book project:

public static Html show(Book b) {

return p(strong(t("Title:")).t(b.title)).

p(strong(t("Author:")).t(b.author)).

p(strong(t("Copies:")).t(b.num\_copies)).

t(link\_to("Edit", "/edit?id=" + b.id())).t(" | ").

t(link\_to("Back", "/"));

}

Here the show method takes a Book and returns an Html, which is a data structure representation of HTML. The View superclass provides a variety of methods for constructing HTML, e.g., the p(...) method constructs HTML with paragraph tags <p>...</p>, where the child ... is more HTML that is created by calling some other methods inherited from View. Moreover, an Html object has as instance methods all the same tag methods as View, which are used to sequence HTML. For example, calling p(a).p(b) returns in <p>a</p><p>b</p> for some a and b.

Your next task is to implement View and Html such that calling toString on an Html returns a string containing the corresponding HTML. **These methods should not modify the current HTML object, i.e., they should be purely functional.** Here are the tags you need to support:

* Standard HTML tags br, p, div, strong, h1, tr, th, td, table, thead, tbody, textarea. For textarea, don't forget to include the name argument as an attribute (see the example form in JServer.java).
* There should be no spaces between the <, >, and output HTML tags. The tags should all be in lowercase. Self-closing tags, specifically br, should be rendered with the slash, i.e., <br/>. There should only be one space before any attributes. The value of any attributes should be quoted with double quotes.
* t(Object o) should call o.toString() and convert the result into an Html.
* empty() should return HTML corresponding to an empty string. This method is not part of Html.
* seq(Html h) should sequence the HTML h after this. This method is not part of View.
* link\_to(String text, String url) should return HTML corresponding to a link to URL url with text text, e.g., link\_to("Show", "/show?id=42") should produce <a href="/show?id=42">Show</a>.
* form(String action, Html child) should return HTML corresponding to a form that sends a POST request in [UTF-8 (Links to an external site.)](https://en.wikipedia.org/wiki/UTF-8) to URL action and contains child as its body. For example, form("/create", html) should produce <form action="/create" accept-charset="UTF-8" method="post">HTML</form>, where HTML is the contents of child.
* submit(String value) should return an HTML submit button with the given value, e.g., submit("Save") should produce <input type="submit" value="Save"/>.

For example, the following code:

Book b = new Book();

b.title = "Programming Languages: Build, Prove, and Compare";

b.author = "Norman Ramsey";

b.num\_copies = 999;

String s = BookView.show(b);

results in the following HTML (spaces and line breaks added for clarity):

<p><strong>Title:</strong>Programming Languages: Build, Prove, and Compare</p>

<p><strong>Author:</strong>Norman Ramsey</p>

<p><strong>Copies:</strong>999</p>

<a href="/edit?id=...">Edit</a> | <a href="/">Back</a>

Controllers

When a web request comes in, it's eventually passed to a *controller*, which handles the request and returns an HTML page in response. For example, here is a controller from our example app:

public static Html show(Map<String, String> params) {

int id = Integer.parseInt(params.get("id"));

Book b = (Book) Book.find(id);

return BookView.show(b);

}

The input to *all* controller methods is a hash of parameter names to values, both of which are strings, and every controller method is public and static and returns an Html. In this particular case, the router (discussed next), will map the URL http://localhost:8000/show?id=42 to a call to show where id is mapped to "42". Then the body of show does some computation, in this case retrieving a book from the database, and returns the corresponding HTML page by calling one of the view methods.

If you look through the controller methods for the example app, you'll see that they support a basic [CRUD (Links to an external site.)](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) interface, i.e., create, read, update, and delete.

There's actually no code you need to implement to support controllers. There is a superclass, Controller, but for purposes of this project it can be empty. (For a more full-featured web framework, we'd probably want to add some functionality to it.)

The Router

As we just saw, different HTTP requests are handled by different controllers. Among other things, each HTTP request has a [verb (Links to an external site.)](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods) (e.g., GET, POST) and a *path* (e.g., /show from the URL localhost:8000/show?id=42). The job of the *router* is to map such requests to controller methods.

The router has to be configured on a per-app basis, which is done with a series of calls to addRoute. For example, here is the routing for the example app:

JRouter r = new JRouter();

r.addRoute("GET", "/", BookController.class, "index");

r.addRoute("GET", "/show", BookController.class, "show");

r.addRoute("GET", "/new", BookController.class, "new\_book");

r.addRoute("GET", "/edit", BookController.class, "edit");

r.addRoute("POST", "/create", BookController.class, "create");

r.addRoute("POST", "/update", BookController.class, "update");

r.addRoute("GET", "/destroy", BookController.class, "destroy");

For example, the second call to addRoute tells the router to map a GET request for show to the BookController's show method.

You need to implement the following behavior for JRouter:

* A JRouter should maintain a list of routes in internal state (so you'll need to add at least one field).
* Calling void addRoute(String verb, String path, Class clazz, String method) should add a route from HTTP verb for path to the clazz class's method.
* Calling String getRoute(String verb, String path) should return a string of the form "clazz#method" if such a route exists, or null otherwise. For example, after the sequence of addRoute calls above, calling r.getRoute("GET", "/show") should return "BookController#show".
* Calling Html route(String verb, String path, Map<String, String> params) should look up the controller method corresponding to verb and path, call it with the given params, and return the result. If no such route exists, this method should raise an UnsupportedOperationException.

The HTTP Server

Finally, jrails include a class JServer with a method void start(JRouter r) that starts up an HTTP server on port 8000, listens for requests, and routes any requests received through r, sending the result back to the web browser. We've written this class for you, and you shouldn't need to modify it.

Then each application sets up its routes and calls start to launch the web server. We've put all the necessary code for the example app in a class called Main, which you can run as described above.

Errors

In general, when web servers encounter errors, they often send some nice web page back to the web browser indicating something went wrong. But for this project, we're not going to do that. Instead, your web server is just going to crash with an exception. That's okay for this project, though it wouldn't be great for a real-world system.

How to Test Your Code

Although on the surface this project seems hard to test—e.g., you might think you have to write tests that connect over the network to a web server—in fact all the model/view/router code is designed to be tested locally. You'll notice that the only code involving networking is in JServer, which you didn't write and therefore you don't have to test.

Thus, as part of this project, you must write at least one test case and post it on Piazza to share with the class. (Warning: We don't guarantee that posted test cases are correct!) Since there are several different parts of the project, we'll use the following rules to split up the tests:

|  |  |
| --- | --- |
| **Last digit of Tufts ID** | **Class for your test** |
| 0-3 | Model.java |
| 4-7 | View.java and Html.java |
| 8-9 | Router.java |

To use the testing framework for your project, put your tests in the appropriate file in jrails/src/test/java/jrails/. We've gotten you started with a few basic tests already.

What to turn in

Put all your code in the [code skeleton](https://canvas.tufts.edu/courses/37277/files/4098778?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4098778/download?download_frd=1)we have supplied, and upload your solution using Gradescope.

[Previous](https://canvas.tufts.edu/courses/37277/modules/items/685155)[Next](https://canvas.tufts.edu/courses/37277/modules/items/665948)

**Submission**

[Submission Details](https://canvas.tufts.edu/courses/37277/assignments/223603/submissions/80402)

Grade: 92 (100 pts possible)

Graded Anonymously: no

**Comments:**

No Comments

Project 05: A Concurrent MBTA Simulator

Reminder: you'll need to submit your test case for this project no later than **11:59 pm ET on April 18.**

Introduction

Object-oriented programming was pioneered by the [Simula (Links to an external site.)](https://en.wikipedia.org/wiki/Simula" \t "_blank) programming language, which was designed for doing simulations. Object-orientation is useful for many different kinds of programming, but for this project you will go back to basics and implement a simulation using Java. As usual, we have a [code skeleton](https://canvas.tufts.edu/courses/37277/files/4098810?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4098810/download?download_frd=1)to get you started.

More specifically, you will implement a multi-threaded simulation of the T. In your simulation, passengers will ride trains between stations, boarding and deboarding (getting on and off) to complete their journey. Your simulation will generate a log showing the movements of passengers and trains, for example:

$ ./sim sample.json

Passenger Alice boards red at Davis

Passenger Bob boards green at Park

Train green moves from Park to Government Center

Train red moves from Davis to Harvard

Train red moves from Harvard to Kendall

Train green moves from Government Center to North Station

Train green moves from North Station to Lechmere

Train green moves from Lechmere to East Sommerville

Passenger Alice deboards red at Kendall

Train green moves from East Sommerville to Tufts

Passenger Bob deboards green at Tufts

Your simulation will be multi-threaded, with a thread for each passenger and each train. That means if you run the simulation multiple times, you may get different results depending on the scheduler! To make testing and debugging easier, you will also build a *verifier* that checks that the simulation result is sensible, e.g., passengers can only deboard trains at the stations the trains are at, trains must move along their lines in sequence, etc.

Building and Running the Code

This project uses JSON. For convenience working with JSON, we have included [Gson (Links to an external site.)](https://github.com/google/gson" \t "_blank), a library for working with JSON in Java. Since we're doing that, we've also thrown in JUnit for testing. To keep things simple, rather than provide a full-fledged build system, we've instead provided several scripts:

* ./build - run javac \*.java with Gson and JUnit in classpath
* ./sim *config\_filename* - run java Sim *config\_filename* with Gson in classpath
* ./verify *config\_filename* *log\_filename* - run java Verify *config\_filename* *log\_filename* with Gson in classpath
* ./test *test\_classname* - run java org.junit.runner.JUnitCore *test\_classname* with Gson and Junit in classpath

You may alternatively add the appropriate jar files to your CLASSPATH instead and not use these scripts.

Part 1: Simulation Configuration

The input to the simulation describes the initial configuration: the set of train lines and the set of passenger journeys. For this project, that input will be specified in [JSON (Links to an external site.)](https://en.wikipedia.org/wiki/JSON). For example, here is sample.json, a sample configuration included with the project:

{

"lines": {

"red": [ "Davis", "Harvard", "Kendall", "Park", "Downtown Crossing",

"South Station", "Broadway", "Andrew", "JFK" ],

"orange": [ "Ruggles", "Back Bay", "Tufts Medical Center", "Chinatown",

"Downtown Crossing", "State", "North Station", "Sullivan" ],

"green": [ "Tufts", "East Sommerville", "Lechmere", "North Station",

"Government Center", "Park", "Boylston", "Arlington", "Copley" ],

"blue": [ "Bowdoin", "Government Center", "State", "Aquarium",

"Maverick", "Airport" ]

},

"trips": {

"Bob": [ "Park", "Tufts" ],

"Alice": [ "Davis", "Kendall" ],

"Carol": [ "Maverick", "Government Center", "Tufts" ]

}

}

The top-level is a JSON object (JSON terminology for a map) with keys lines and trips. The key lines maps to another JSON object, which maps train names (there is only one train per line in this simulation) to a list of station names. The key trips maps passenger names to a list of station names. The rules for the simulation will be described fully below.

For the first part of the project, you will work with the classes Train, Station, and Passenger, instances of which represent the corresponding entities, and class MBTA, which represents the state of the simulation.

First, taking a brief trip back to the land of design patterns, implement the following methods:

* Train.make(String name), Station.make(String name), and Passenger.make(String name). These factory methods should return instances of the corresponding classes, where **if one of these methods is called multiple times with the same name, it must always return the same object.** For example, Train.make("red") == Train.make("red") (note the physical equality test). Thus, you'll need to implement some kind of caching. You'll have a few design choices about exactly how to set up the caching; we'll leave those choices to you, and we'll only test for functionality.

Second, implement the following methods (you will need to add fields to MBTA, plus another class, as described below):

* MBTA#addLine(String name, List<String> stations) - Add a new train line to the simulation, where name is the name of the line and stations is an ordered list of stations on the line. This method will be handy for writing test cases.
* MBTA#addJourney(String name, List<String> stations) - Add a new train line to the simulation, where name is the name of the passenger and stations is an ordered list of stations the passenger wants to visit. This method will be handy for writing test cases.
* MBTA#reset() - Reset the MBTA state so it contains no lines and no journeys. Again, this method will be handy for writing tests.
* MBTA#loadConfig(String filename) - Load a JSON simulation configuration (as specified above), adding the lines and journeys listed in the JSON file. Let's discuss how to do this next. This will take a lot of words, but you won't actually have to write much code.

To load the JSON configuration file, you will definitely want to use the class [Gson (Links to an external site.)](https://www.javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/com/google/gson/Gson.html" \t "_blank), specifically one of the fromJson methods. You can use any of them that work, but here's our suggestion. The Gson library's key feature is that it can take any Java object and turn it into a JSON string, and vice-versa. For example, consider the following code (included in the code skeleton):

public class C {

public List l;

public Map m;

public static void main(String[] args) {

Gson gson = new Gson();

C c = new C();

c.l = List.of("a", "b", "c");

c.m = Map.of("k1", "v1", "k2", "v2");

String s = gson.toJson(c);

System.out.println(s);

C c2 = gson.fromJson(s, C.class);

System.out.println(c2.l);

System.out.println(c2.m);

}

}

This code first creates an instance of C and then converts it to a JSON string. The first print statement produces:

{"l":["a","b","c"],"m":{"k2":"v2","k1":"v1"}}

Then, the code uses one form of fromJson to take that string and produce a C from it. Notice that the JSON parser needs to know what kind of object to produce, so we pass the type of the object, in this case C.class, to fromJson. The last two lines print out the fields of c2 to show they are correct.

To implement MBTA#loadConfig, look at the JSON type for simulation configurations and then create a new class that, when turned into JSON, will look exactly like the configuration format. You can experiment with this by using toJson to look at how your new class is encoded. Once you have the right class, all you need to do is call fromJson and, magically, you'll be able to turn an on-disk JSON file into an in-memory representation! Once you have the in-memory representation, you can then traverse it and call addLine and addJourney as appropriate to set up the simulation.

If you want to see a worked-out example of this, you can look in classes Log and LogJson, which include code to turn simulation logs (discussed below) to and from JSON. Note that process is more complicated than what you'll need to do for configurations. So the code you have to write will be much simpler than the code for logs.

Part 2: Verifying Simulation Output

The next part of the project is to write a verifier that, given the initial configuration and a *log* of a simulation, ensures that the log follows all the simulation rules. We've already built the basic logging infrastructure for you. The class Log stores a List<Event>, where an Event is one of the following three classes:

* MoveEvent(Train t, Station s1, Station s2) - Train t moves from s1 to s2
* BoardEvent(Passenger p, Train t, Station s) - Passenger p boards t at s
* DeboardEvent(Passenger p, Train t, Station s) - Passenger p deboards t at s

Your job is to implement:

* MoveEvent#replayAndCheck(MBTA mbta), BoardEvent#replayAndCheck(MBTA mbta), and DeboardEvent#replayAndCheck(MBTA mbta) - Check that the change indicated by the event is valid. If not, raise an exception (any exception). If it is valid, modify mbta so that the change indicated by the event is incorporated into it. For example, if train green is at Tufts and the event is MoveEvent(green, Tufts, East Sommerville), then the event is valid (no exception raised), and after the method returns, green will now be at East Sommerville.
* MBTA#checkStart() and MBTA#checkEnd() - Check that the simulation is in the correct initial and final states, respectively.

We've already written the following code in class Verify to use these methods to verify a log against a configured MBTA:

public static void verify(MBTA mbta, Log log) {

mbta.checkStart();

for (Event e : log.events()) {

e.replayAndCheck(mbta);

}

mbta.checkEnd();

}

The simulation is initially configured with a set of lines (e.g., Red, Green, Blue, Orange), each of which has an ordered list of stations (e.g., Davis, Porter, Harvard, Central, Kendall). Stations are identified by names, and if the same station name is on two lines, that's considered just one station. You can assume all lines, stations, and passengers are named uniquely. There is exactly one train on each line, whose name is the same as the line. The initial configuration also includes a set of passengers, each of whom has a journey they want to take (e.g., Alice wants to go from Kendall to Davis).

Here are the rules your verifier needs to check:

1. At the start of the simulation,
   1. Each train starts at the first station in the line's ordered list of stations.
   2. Each passenger starts at the initial station of their journey.
2. Trains move along their line until they reach the end, at which point they change direction and follow the line in reverse until they reach the beginning of the line, then go forward on the line, etc. Trains never skip stations.
3. At most one train can be at a station at a time. Trains may not be in-between stations; they must wait to leave their current station until the station they are moving to is available.  Trains and stations can hold an unbounded number of passengers. You can assume the initial configuration does not place two trains at the same station.
4. A passenger can only board a train that is at the same station as the passenger; they can only deboard a train at the station it is currently at.
5. A passenger exits from a train only if they have arrived at the next station on their journey. You can assume that passenger journeys include all the necessary stops to change lines, e.g., if a passenger wants to go from Maverick to Tufts, they journey will probably be Maverick, Government Center, Tufts.
6. The simulation ends when all passengers have arrived at their final stops. It is an error for the simulation to end early. It is fine if trains run a bit past the point where all passengers have finished their journeys.
7. You can assume all specified passenger journeys are possible on the given set of lines. You don't need to check if the journeys themselves are sensible.

Part 3: The Simulation

Finally, you must implement the simulation. More specifically, implement

* Sim#run\_sim(MBTA mbta, Log log) - runs the simulation starting with the already configured mbta, logging events using log.

and

* In design.txt, *briefly* explain how your simulation uses threads and locks. The purpose is to give the TAs a quick overview of your code so that they can easily check that you followed the rules below. It is not necessary to write a lot of text.

Your simulation must obey the rules checked by the verifier, plus the following rules:

* Your simulation must use the following methods to log events as trains and passengers move:
  1. log.train\_moves(Train t, Station s1, Station s2) when train t leaves station s1 and enters station s2.
  2. log.passenger\_boards(Passenger p, Train t, Station s) when passenger p boards train t at station s.
  3. log.passenger\_deboards(Passenger p, Train t, Station s) when passenger p exits train t at station s.

There methods will both store the event in the log and print the event to the console (and flush the output after doing so). The second part will be useful if your simulation deadlocks.

* Use multiple threads in the simulation. More specifically, each passenger and each train should have a corresponding thread. You may use additional threads as well, but no fewer.
* When a train moves to a new station, it should stay at that new station (i.e., not attempt to move) for 0.5 seconds (call Thread.sleep).
* **Do not use a single global lock.** You should aim to let passengers and trains move as independently as possible, though of course there will be some mutual exclusion to ensure there are no data races. You may assume that the simulation configuration does not itself cause deadlocks (for example, a cycle in the station map such that trains are all waiting for each other to move before they can move).
* **Avoid data races.** Your code should not have any.
* **Do not use busy waiting or spinning.** You will want to use await/signalAll, wait/notifyAll, or classes from [java.util.concurrent (Links to an external site.)](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/package-summary.html" \t "_blank) instead.
* It is possible that a passenger might miss their train, even if it arrives at their station, or might miss their stop. In these cases the passenger remains at the station or on the train, respectively.
* You maybe use ReentrantLock and/or the synchronized keyword and/or data structures from java.util.concurrent).

In class Sim, we've already written a useful main method for you that uses loadConfig (part 1) to set up the initial conditions; calls run\_sim (part 3); writes the resulting log file to disk as log.json; and then runs verify (part 2) on the result to check that all the rules are followed. You can run this main method from the command line using the command ./sim *config\_filename*. If you want to run the verifier separately on the saved log.json, you can run the command ./verify *config\_filename* log.json. Note that log.json gets overwritten each time you run the simulation, so be sure to save a copy somewhere else if you need it.

Part 4: Test Cases

We've given you a script ./test *test\_classname* you can use to run JUnit tests in the given class (and we've given you a class Tests with a trivial test inside it). For this project, **you must write at least one test case for Part 2 (the verifier), only, and share it on Piazza**. You can use the methods in MBTA and Log to create an initial configuration (without using JSON, even) and sample log, and then directly call Verify#verify to check that the verifier passes (returns normally) or throws an exception, depending on the test.

What to turn in

Put all your code in the [code skeleton](https://canvas.tufts.edu/courses/37277/files/4098810?wrap=1)[Download code skeleton](https://canvas.tufts.edu/courses/37277/files/4098810/download?download_frd=1)we have supplied, and upload your solution using Gradescope.

[Previous](https://canvas.tufts.edu/courses/37277/modules/items/685159)[Next](https://canvas.tufts.edu/courses/37277/modules/items/665968)

**Submission**

[Submission Details](https://canvas.tufts.edu/courses/37277/assignments/223604/submissions/80402)

Grade: 141 (146 pts possible)

Graded Anonymously: no

**Comments:**

No Comments