Java Practice Test
Friday 13th December 2013 3-00pm – 6-00pm
THREE HOURS
(including 15 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.

The maximum total is 25.

Credit will be awarded throughout for clarity, conciseness, useful commenting, and appropriate use of assertions.

Important note: THREE MARKS will be deducted from solutions that do not compile. You should comment out any code which you cannot get to compile.

Do Not Melt The Snowman!

Do Not Melt The Snowman! is a board-based puzzle game involving *lasers*, *mirrors*, *walls*, a *target* and *snowmen*. The aim of the game is to rotate a laser *emitter* and the mirrors, such that the laser bounces off the mirrors and strikes the target. However the emitter cannot be shut off, and so care must be taken when rotating the pieces to ensure that the laser does not accidentally strike and heat up a snowman – doing so would cause the snowman to melt, making the player lose the game!

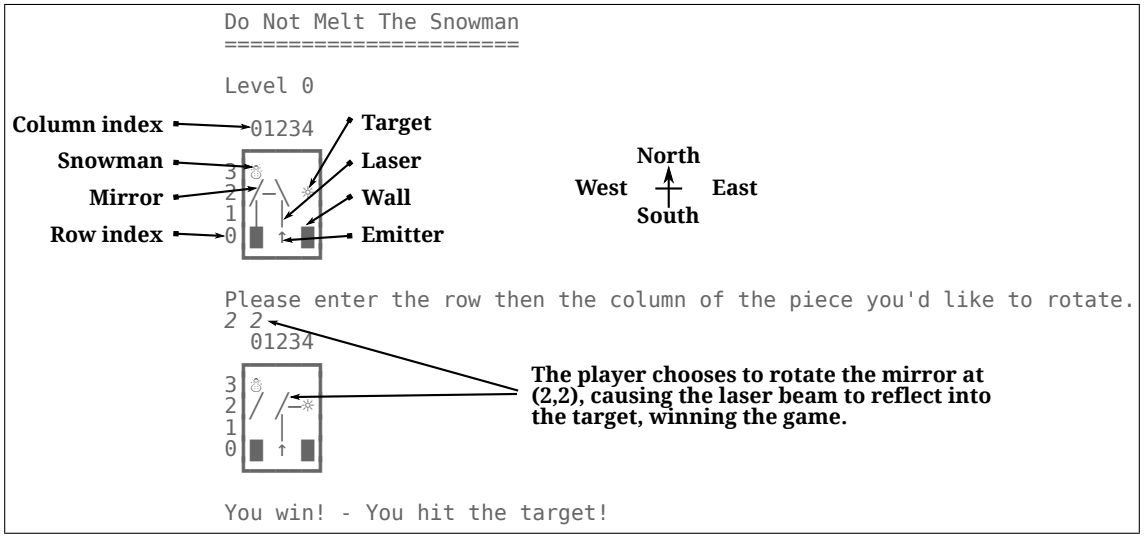


Figure 1: An example *Do Not Melt The Snowman* game being won.

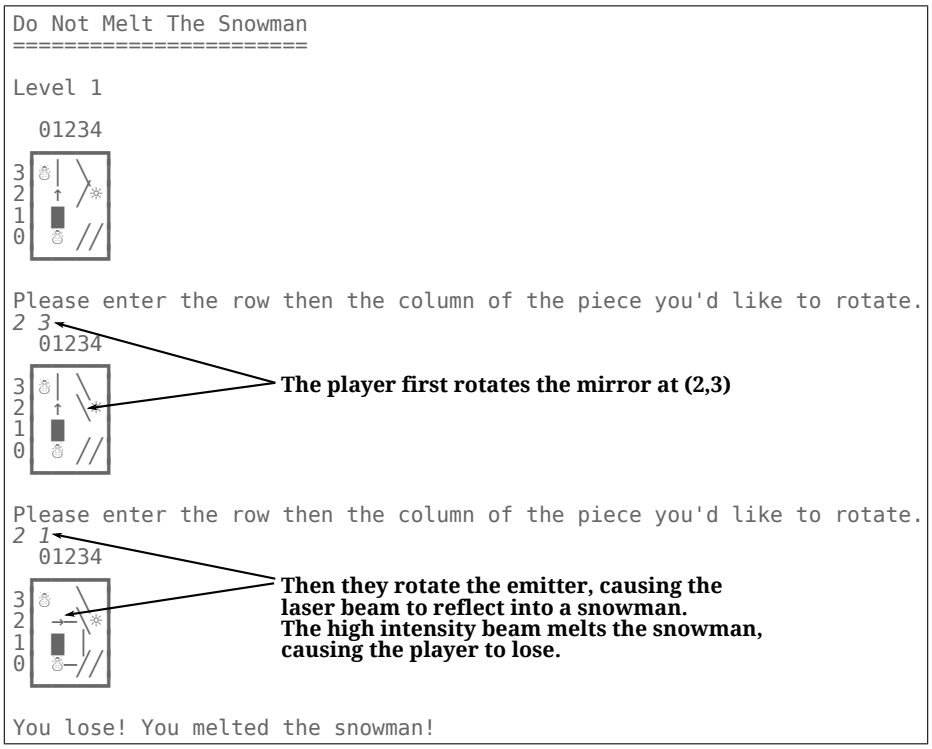


Figure 2: An example *Do Not Melt The Snowman* game being lost.

The initial arrangement of the pieces on a rectangular board is called a *level*. Each level may require a board of a different size, from 1×2 up to 10×10. The number of mirrors, walls and

snowmen per level may vary, but there will always be only a single emitter and a single target. To help you test your implementation, we have provided 11 sample levels (numbered 0 through 10).

After a level has been loaded, the game runs in a loop, repeatedly performing the following two steps:

- The laser is fired from the emitter, bouncing off the mirrors.
  - If the beam leaves the board, or hits a wall or the emitter, then a *miss* occurs. Play may then continue.
  - If the beam hits a snowman, then the snowman melts and the player loses.
  - If the beam hits the target, then the player wins.
- Assuming a *miss* occurred, the player may then select a single mirror, or the emitter, to rotate. The piece is rotates clockwise by 90 degrees.

Getting Started

In the `DoNotMeltTheSnowman` directory in your Lexis home directory you will find ten `.java` files.

Six of these files you should not need to edit, but will need to make use of during this exercise. They provide:

- `IOUtil.java`: contains the input/output utility methods you have been using in your labs and tutorials.
- `Coordinate.java`: contains a class `Coordinate` which stores two `int`s, representing an x, y location on the board. It provides two methods, `int getX()` and `int getY()` to access the two locations.
- `Piece.java`: contains an enum `Piece` which represents the 13 different types of pieces that can be on the board. The full list of pieces can be found in the description of Part I a, below.
- `Result.java`: contains an enum `Result` which represents the three possible results from a laser firing. The elements of `Result` are `HIT_TARGET`, `MELT_SNOWMAN` and `MISS`.
- `Level.java`: contains a class `Level` which stores a description of a level.
- `Levels.java`: provides a single static method `getLevels`, which returns an array of the 11 predefined `Level` objects.

Three of the remaining four files you should edit, filling in the stub methods as explained below. Once these are completed you should be able to load and play the levels of *Do Not Melt The Snowman!*

- `PieceUtils.java`: contains stubs for static utility methods which you will need to implement as part of this exercise. It also contains three given methods that you may need to use later on:
  - `boolean isEmitter(Piece p)`, which returns `true` if, and only if, the given piece represents an emitter.
  - `Piece hideLaser(Piece p)`, which converts laser pieces into the empty piece, but leaves other pieces unchanged.
  - `Piece rotate(Piece p)`, which will rotate any `Piece` by 90 degrees clockwise.

- `Board.java`: contains the class `Board` – instances of this class will store a two-dimensional array of `Piece` for the board, and the `Coordinate` of the emitter in the board. Method stubs are provided which, when completed, will allow a `Board` to fire the laser. You are also given methods that rotate pieces and clear all the laser tiles on the board.
- `DoNotMeltTheSnowman.java`: contains an empty `main` method, which you should complete to enable the user to play a game of *Do Not Melt The Snowman!*

You should not change the signatures of any of the provided methods: auto-testing of your solution depends on the methods having exactly their original signatures. However you may feel free to add additional methods and classes (e.g. for testing) as you see fit. Any new Java files should be placed in the `DoNotMeltTheSnowman` directory.

To help you test your work as you go, we have provided a test suite for all of Part I and Part II in `Tests.java`. The tests are checked using `assert`, so only the first failure will be reported. If you attempt the questions in a different order you may need to comment out or reorder the method calls in the `main` method of this file.

What to do

Part I: PieceUtils.java (total: 13 marks)

This class contains some static utility methods that are used elsewhere in the game. You will need to complete these to make implementing the rest of the game easier.

a. Parsing Pieces (4 marks)

The descriptions of the levels in the game are provided in an array of `chars`. Each ASCII character in the array corresponds to one piece, as detailed in the following table:

`Piece`	`char` form	Graphic
EMITTER_NORTH	'^'	↑
EMITTER_EAST	'>'	→
EMITTER_SOUTH	'v'	↓
EMITTER_WEST	'<'	←
LASER_VERTICAL	'|'	│
LASER_HORIZONTAL	'-'	─
LASER_CROSSED	'+'	┼
MIRROR_SW_NE	'/'	╱
MIRROR_NW_SE	'\\'	╲
WALL	'#'	▉
TARGET	'o'	✳
EMPTY	' '	
SNOWMAN	'@'	☃

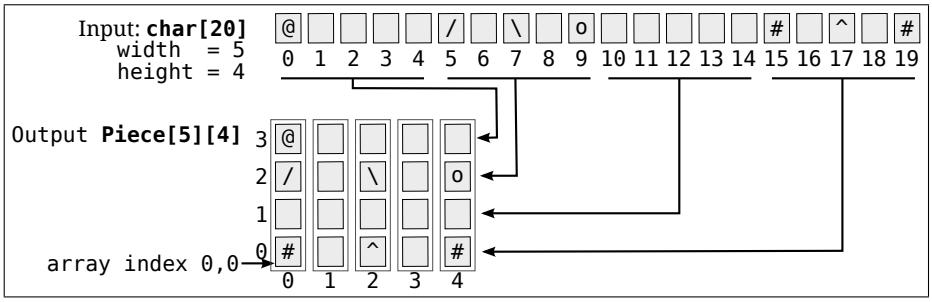
1. Implement a static method `Piece charToPiece(char c)` – which converts an ASCII character into the corresponding `Piece`.

   You do *not* need to create separately named constants for each of the ASCII characters in this method.

2. Hence, implement a static method `Piece[][] charsToPieces(char[] description, int width, int height)` – which given the ASCII representation of a level, and it's associated width and height (this method should

assume `description` is `width`×`height` in length), builds a rectangular two-dimensional array of `Piece`. The outer array should have length `width`, and the inner arrays should each have length `height`.

See the following diagram for an example input and output. Notice that in the output array index $(0, 0)$ is the *bottom left* corner of the board. This means the first character in the input array ends up in the *last* row of the output array.



b. Finding the Emitter (3 marks)

Later on, the `Board` class will need a way to find the emitter on a two dimensional grid. The two following static methods will allow it to do so.

- You are given a static method `boolean isEmitter(Piece p)` – which given a `Piece`, returns `true` if, and only if, that piece is an emitter facing north, east, south or west.
- Hence, implement a static method `Coordinate findEmitter(Piece[][] pieces)` – which returns the `Coordinate` of the single emitter in a two dimensional array of `Piece`s. If this method fails to find an emitter piece, it may return `null`.

c. Working with Lasers (6 marks)

Later, the `Board` class will need to add and remove laser tiles on the board. By doing this, it will be able to show the user the path the laser is currently taking. It will also need to know where the laser beam travels to in one step when leaving a piece.

To help with this, in `PieceUtils` you are given a static method `Piece hideLaser(Piece p)` – which, when given a horizontal, vertical or crossed laser piece will return an empty one. Otherwise it returns the argument it was given.

1. Implement a static method `Piece addLaser(Piece p, boolean isHorizontal)` – which adds a laser beam to a given piece. The `isHorizontal` argument specifies if the beam to be added is travelling horizontally or vertically.

   For example, adding a horizontal/vertical laser to an empty piece gives a horizontal/vertical laser piece. Adding a horizontal laser to a vertical laser (or vice-versa) gives a crossed laser piece. Adding a laser to any other type of piece will return the original piece.

2. Implement a static method `Coordinate move(Piece p, Coordinate c, int xo, int yo)` – which works out the next co-ordinate the laser beam will visit, assuming the argument `p` is located at co-ordinate `c`. The arguments `xo` and `yo` will be used to describe the velocity the laser beam is currently heading in. You may assume $-1 \leq \text{xo} \leq 1$ and $-1 \leq \text{yo} \leq 1$.

   There are several cases to consider:

   i. `p` is an emitter. In this case, the output co-ordinate is one unit further on in the direction the emitter is facing.

ii. `p` is empty space, possibly with an existing laser beam already in it. In this case the beam moves one step according to `xo` and `yo`.

iii. `p` is a mirror. In this case the beam reflects and moves one step according to the modified velocities in the following Table:

Mirror	Input velocity	Output velocity
`MIRROR_SW_NE`	$-1, 0$	$0, -1$
	$1, 0$	$0, 1$
	$0, -1$	$-1, 0$
	$0, 1$	$1, 0$
`MIRROR_NW_SE`	$-1, 0$	$0, 1$
	$1, 0$	$0, -1$
	$0, -1$	$1, 0$
	$0, 1$	$-1, 0$

iv. `p` is a wall, snowman or target. In this case the beam does not move and the original co-ordinate should be returned.

Part II: Board.java (total: 9 marks)

You will now need to complete the `Board` class, which manages the state of the puzzle board during a game. Instances of a `Board` have two fields – `Piece[][] board`, and `Coordinate emitter`. A constructor for `Board` is provided that initializes these pieces (in Part III you will need to use this constructor to build game boards). This class also provides the implementations of the following public instance methods:

- `void rotatePiece(Coordinate c)`, which rotates the piece located at `c` by 90 degrees clockwise.
- `void clearLasers()`, which removes all laser pieces on the board.
- `void renderBoard()`, which prints out the current board.

a. Querying Coordinates (4 marks)

The `Board` will need a way to query the `board` at certain locations, to determine if the given location stops the laser, and what (if anything) the laser has hit there. These two methods will be useful in b. Firing the Laser, below.

1. Implement the instance method `boolean laserEnds(Coordinate c)`, which returns `true` if and only if, the given co-ordinate would stop the laser.

   The laser stops if `c` is off the board, or if the piece at the location indicated by `c` is not empty, a laser or a mirror.[1]

   Hint: to work out the height of a board, it may be useful to remember that it is guaranteed to be rectangular, and at least 1×2 in dimension.

2. Implement the instance method `Result calculateResult(Coordinate c)`, which returns the result of the laser's action - i.e. missing, hitting the target or melting a snowman.

   If the `c` is inside the board, then if the `Piece` at location `c` represents the target then the `HIT_TARGET Result` is returned. Otherwise if the `Piece` represents a snowman then the

[1] This means that, even though they will be melting, snowmen *will* stop a laser.

result is `MELT_SNOWMAN`. In all other situations (including when `c` is outside the board), then a `MISS` occurs.

b. Firing the Laser (5 marks)

Implement the instance method `Result fireLaser()` – which simulates the firing of the laser, tracing its path. While following the path of the laser, this method will replace empty pieces that it traverses over with laser pieces in the correct orientation. When the laser finally stops, it returns the result of doing so.

To implement this, you will need to create a `Coordinate` variable, call it `current` that starts at the emitter's known location. You will also need to keep track of the laser's current velocity. This can be stored in two `int`s (for the x and y direction), both are initially 0.

Then, in a loop (whose body must run at least once), you will need to:

- Use `PieceUtils.addLaser` to update the board at the current location with a laser.
- Use `PieceUtils.move` to work out the co-ordinate the laser moves to. Call this location `next`.
- Calculate the difference between `next` and `current` in both the x and y components to update the laser's current velocity.
- Make `current` equal to `next`.

The loop should end when `current` reaches a location that stops the laser. Once the laser's stopping location is determined, you need to return the `Result` based on that location.

Part III: DoNotMeltTheSnowman.java (total: 3 marks)

a. Making the Game (3 marks)

You should now have the necessary code to implement the main game loop, and allow a user to play the game.

Implement the static `void main(String[] args)` method to create the program. Your `DoNotMeltTheSnowman` program will typically be invoked with a single argument specifying a level number to load. You may find the built in static method `Integer.parseInt(String s)` useful to turn a command line `String` argument into an `int`. For example, `Integer.parseInt("0")` will return the `int` 0.

You should then use the `Levels.getLevels()` method to get the array of levels, and index that to find the appropriate `Level` object. Use the information contained within the `Level` to build a new `Board` object. (Hint: don't forget to use `PieceUtils.charsToPieces` as part of this).

Next, greet the user, and then start the main game loop. In that loop you will need to:

- Fire the laser, remembering the result.
- Render the state of the board.
- If the result from firing ends the game, print out a message and stop.
- Otherwise ask the user for a row then column of a piece to rotate (Hint: `IOUtil.readInt()`), and rotate it.
- Then clear the lasers.

For example, a sample interaction follows showing the program running on the first level. You do not need to match the output precisely, but to help with testing please keep the input format the same:

```
> java -ea DoNotMeltTheSnowman 0
Do Not Melt The Snowman
======================

Level 0

  01234
 ┌─────┐
3│☃    │
2│/─\ ✳│
1│| |  │
0│█ ↑ █│
 └─────┘

Please enter the row then the column of the piece you'd like to rotate.
2 2
  01234
 ┌─────┐
3│☃    │
2│/ /─✳│
1│  |  │
0│█ ↑ █│
 └─────┘

You win! - You hit the target!
```

b. Playing the Game (For fun!)

Once you have implemented your program, why not have a go and try and solve the 11 provided levels (indexes 0 through 10 in the `Levels.getLevels` array).

Total across all parts: 25 marks