*Last Updated: 2015-04-24 Fri 09:53*

**CS 211 Project 6: You Must Fit It. Fit It Good.**

* **Due: Sunday 4/26/2015 by 11:59 pm**
* *Approximately 5.7% of total grade*
* Submit to [**Blackboard**](https://mymasonportal.gmu.edu/)

**CODE DISTRIBUTION**

* Tests: [P6Tests.java](http://cs.gmu.edu/%7Ekauffman/cs211/P6Tests.java) [TESTFILES.zip](http://cs.gmu.edu/%7Ekauffman/cs211/TESTFILES.zip) (Updated Fri Apr 24 09:52:35 EDT 2015)
* Project Code: [p6pack.zip](http://cs.gmu.edu/%7Ekauffman/cs211/p6pack.zip)

**CHANGELOG:**

Fri Apr 24 09:52:30 EDT 2015

Minor update to the tests to correct some string constants in space\_fitsX() tests and the impossible2.txt data file. When a fit is found, print FOUND FIT in FitIt.main() to match the tests rather than the original FIT FOUND. The test\_main() tests have been updated to simply verify that a fit is found or not rather than check for a specific solution.

Thu Apr 23 18:57:27 EDT 2015

The test files have been updated to fix a few minor bugs and add the honors section tests. Honors section students will need to uncomment their tests starting at line 3824 in P6Tests.java and ending near the bottom of the file. There are

* 113 Tests for non-honors students
* 123 Tests for honors students

Thu Apr 23 16:59:48 EDT 2015

In order to support testing the main() method, it should take an optional second command line argument. If present, this argument is the name of an output file which should be printed into rather than printing to the screen. For example

> java FitIt problem.txt output.txt

should fill up the file output.txt with what would normally print to the screen. The [section on the FitIt](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#fit-it) class has been amended to mention this. The behavior is identical to writing to output files in Project 4 so refer to your solution there if in doubt.

Thu Apr 23 14:24:02 EDT 2015

Test cases are now available. Download the two files P6Tests.java and TESTFILES.zip, unzip the zip into your project directory, and start testing. The only tests missing are those for the honors section which will be added sometime Friday.

Tue Apr 21 14:19:11 EDT 2015

* Fixed minor typo that in Space: getHeight() reports number of rows, getWidth() reports columns.
* Added setup section to clarify project directories and zips should be named via the usual convention: ckauffm2-205-p6.
* Clarified that all shapes should be initialized to CW0 rotation when produced from FitIt.makeShape(..).

Tue Apr 21 03:29:00 EDT 2015

Added the honors section materials (adds a "search for all fits" method, which also must be discussed in the design document).

Thu Apr 16 17:50:08 EDT 2015

Minor update to indicate how long answers should be in the [Design Doc](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#design-doc) under the manual inspection criteria.

Clarified in the [Submission Section](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-11) that you SHOULD submit the additional java files you create to solve complete this project as per [Piazz @1610](https://piazza.com/class/i4w9yjkwnrl2by?cid=1610).

Wed Apr 15 09:52:28 EDT 2015

Removed reference to old exception name from the FitIt prototypes; all exceptions should be FitItExceptions.

**Table of Contents**

* [1. Design Freedom](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-1)
* [2. Two-dimensional Packing](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-2)
  + [2.1. Spaces and Shapes](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-2-1)
  + [2.2. Placing Shapes in Spaces](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-2-2)
  + [2.3. Fitting Shapes into a Space](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-2-3)
  + [2.4. Shape Rotations](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-2-4)
* [3. Implementation Issues](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-3)
  + [3.1. Representation of Spaces and Shapes](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-3-1)
  + [3.2. Recursive Search for Fit](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-3-2)
  + [3.3. Backtracking and Recursive Search](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-3-3)
* [4. Required Design Elements](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4)
  + [4.1. Top-level Methods: FitIt.java](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4-1)
  + [4.2. Shape Interface](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4-2)
  + [4.3. Space Interface](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4-3)
  + [4.4. Rotation Enumeration](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4-4)
  + [4.5. Exception Classes](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-4-5)
* [5. Layout String and File Formats](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-5)
  + [5.1. Layout String Formats](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-5-1)
  + [5.2. Space Layout Strings](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-5-2)
  + [5.3. Problem Input Formats](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-5-3)
* [6. Sample Runs of FitIt.main()](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-6)
* [7. Honors Section (15%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-7)
  + [7.1. Finding All Fits](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-7-1)
  + [7.2. Sample Runs](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-7-2)
* [8. Grading](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-8)
  + [8.1. Automated Tests (50%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-8-1)
  + [8.2. Manual Inspection (50%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-8-2)
* [9. Manual Inspection Criteria](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9)
  + [9.1. Correct Project Setup (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-1)
  + [9.2. Coding Style and Readability (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-2)
  + [9.3. design.txt Design Documentation (10%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-3)
  + [9.4. Adherence to Interface Boundaries within Space (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-4)
  + [9.5. Adherence to Interface Boundaries within searchForFit() (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-5)
  + [9.6. Adherence to Interface Boundaries within main() (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-6)
  + [9.7. Elegance of searchForFit(space,shapeList) (10%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-7)
  + [9.8. Elegance of Space.fitString() (5%)](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-8)
  + [9.9. Honors 15% Inspection](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-9-9)
* [10. Setup](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10)
  + [10.1. Project Directory](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10-1)
  + [10.2. Identity Text File](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10-2)
  + [10.3. Provided Files](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10-3)
* [11. Submission](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-11)
* [12. Setup](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-12)
  + [12.1. Project Directory](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-12-1)
  + [12.2. Identity Text File](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-12-2)
  + [12.3. Provided Files](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-12-3)

**1 Design Freedom**

This will be the final large project for the semester. By now, if you have worked hard on the preceding projects, you should have some skill at constructing programs. Designing programs is another matter: rarely will clients give a detailed break-down of what classes should be used to solve a problem and specifically how those classes should work. Clients are much more likely have a problem with no conception of the code required to solve it. It is important to practice understanding a problem first and then begin experimenting with the classes and methods required to solve the problem. This project will present an opportunity in which a rough design is required (a few classes and interfaces are required) but you will otherwise have a free hand to create whatever classes you feel are necessary to solve the given problem.

Rather than describing a specific route to solving the problem below, this specification discusses the problem itself and some of the complexities you will need to surmount in any solution to the problem. The [Required Design Elements](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#required) section outlines the minimum number of classes and methods that must appear but you should expect there to be somewhat more to the project than this minimal interface.

Freedom comes with some responsibility: you will need to describe the classes you build in a design.txt document that discusses how you implemented certain functionality. This document does not have to be long but does account for significant credit on the project. It must be present and is described in the [section on manual inspection](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#design-doc).

**2 Two-dimensional Packing**

Packing shapes into fixed size containers is an old problem with continued applications, particularly in shipping and handling services. For example, online retailers may seek to pack as many items in an order as possible into a single box to reduce shipping costs. On a larger scale, [packing shipping containers onto oceanic container ships](http://en.wikipedia.org/wiki/Container_ship#/media/File:Panama_Kanal_01_%2840%29.jpg) travel around the world devolves to the same basic problem of packing a set of objects into a fixed volume.

This project will focus on a problem in this class which has the following properties.

**2.1 Spaces and Shapes**

The **space** into which shapes will be packed is a rectangle comprised of blocks which are empty or filled. Each block in a space is referenced by its row/column coordinate which starts at 0. The space may be initialized with some permanently filled blocks. Consider the following space.

........

.......|

.....|||

The dots (periods) represent empty blocks into which shapes may be put while the filled blocks are drawn as vertical lines (pipes). For clarity, here is the same space with numbers around it to indicate the rows and columns.

01234567

0 ........ 0

1 .......| 1

2 .....||| 2

01234567

The space has height 3 and width 8. All blocks are empty except the following filled blocks: (1,7), (2,5), (2,6), (2,7).

There is no requirement that permanently filled blocks must exist on the edges of the space or be contiguous. The following is another valid space demonstrating the arbitrariness of filled block placement.

|.......

..|||..|

..|.|...

........

A **shape** is a rectangular grid of **blocks** which are filled or empty. For example here are a few all-too familiar shapes.

SHAPE T

TTTT

SHAPE e

..e

eee

SHAPE t

.t.

ttt

SHAPE r

rr

rr

SHAPE i

i..

iii

SHAPE s

.ss

ss.

As with a space, a shape represents empty blocks with a period but filled blocks are represented using any other character aside from newlines. By convention, shapes will usually have a single **display character** associated with them which will be used for their filled blocks.

**Note**: Shapes will be restricted to contain a non-empty border: the 0th row, 0th column, last row, and last column must all contain at least one filled block. For more information refer to the section on [illegal layout strings](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#illegal-layout).

**2.2** **Placing Shapes in Spaces**

Shapes may be **placed** into a space if there are no conflicts with any filled blocks (including other shapes). The placement locations indicates where the upper left (0,0) block of the shape would be located in the space. Consider the space and shape below.

SPACE

|.......

..|||..|

..|.|...

........

SHAPE i

i..

iii

Placing the shape at block (2,5) would lead the space to look like the following

Shape at (2,5)

01234567

0 |....... 0

1 ..|||..| 1

2 ..|.|i.. 2

3 .....iii 3

01234567

On the other hand, the shape might be placed at location (2,0).

Shape at (2,0)

01234567

0 |....... 0

1 ..|||..| 1

2 i.|.|... 2

3 iii..... 3

01234567

Note that the block (2,2) is filled in the space. This is not a problem as the shape has an empty block in that position so it fits at (2,0) nicely. However, attempting to place the shape at (0,0) would lead to several filled blocks in the shape conflicting with filled blocks in the space and should result in an error.

Even if block (0,0) of a shape is empty, it is still used to indicate shape placement. Consider the space and shape below.

SPACE

|.......

..|||..|

..|.|...

........

SHAPE t

.t.

ttt

Block (0,0) of the shape is empty and can overlap with other filled blocks. The following are **some** of the valid locations where the shape may be placed. Note the row/column locations indicate where the upper left empty block of the shape will be placed.

Shape at (2,4)

01234567

0 |....... 0

1 ..|||..| 1

2 ..|.|t.. 2

3 ....ttt. 3

01234567

Shape at (2,5)

01234567

0 |....... 0

1 ..|||..| 1

2 ..|.|.t. 2

3 .....ttt 3

01234567

Shape at (1,5)

01234567

0 |....... 0

1 ..|||.t| 1

2 ..|.|ttt 2

3 ........ 3

01234567

Shape at (2,2)

01234567

0 |....... 0

1 ..|||..| 1

2 ..|t|... 2

3 ..ttt... 3

01234567

**2.3 Fitting Shapes into a Space**

The central problem you will need to solve is whether a give list of shapes can **all** be fit together into a given space. For example consider the following space and shapes.

SPACE

|.......

..|||..|

..|.|...

........

SHAPE T

TTTT

SHAPE e

..e

eee

SHAPE r

rr

rr

SHAPE i

i..

iii

SHAPE s

.ss

ss.

One possible fit of the shapes in the space is as follows.

|TTTT.ss

rr|||ss|

rr|e|i..

.eee.iii

There may be other fits but you will primarily be responsible for determining if at least one exists and producing one. This is not always possible. For example if the shape

SHAPE t

.t.

ttt

is added to the above, there is no way to fit all the shapes in the given space without changing the shapes (rotations are discussed in the next section).

Packing shapes into a space constitutes a computationally difficult problem and is almost certainly in the NP-hard complexity class. In simple terms, this means that to solve the problem, one may be reduced to searching every possible arrangement of shapes in a space to see if any works. While this is feasible for small numbers of shapes and spaces, it becomes intractable for large spaces and long lists of shapes as there are so many possibilities.

**2.4** **Shape Rotations**

Shapes can be rotated in 90 degree increments to open up more potential packing solutions. Your implementation will be required to allow shapes to **rotate clockwise in increments of 90 degrees**. There are therefore 4 possible rotations of each shape referred to as follows.

* CW0: no rotation
* CW90: 90 degree rotation clockwise
* CW180: 180 degree rotation clockwise
* CW270: 270 degree rotation clockwise

Consider the following shape and its four valid rotations.

SHAPE t

.t.

ttt

SHAPE t

height: 2; width: 3; rotation: CW0

.t.

ttt

SHAPE t

height: 3; width: 2; rotation: CW90

t.

tt

t.

SHAPE t

height: 2; width: 3; rotation: CW180

ttt

.t.

SHAPE t

height: 3; width: 2; rotation: CW270

.t

tt

.t

Notice that the height and width of the shape changes with rotations and the positions of the filled blocks change with each rotation.

If the initial orientation of the shape were different, then the rotations would be altered as is the case with the shape below.

SHAPE x

x.

xx

x.

SHAPE x

height: 3; width: 2; rotation: CW0

x.

xx

x.

SHAPE x

height: 2; width: 3; rotation: CW90

xxx

.x.

SHAPE x

height: 3; width: 2; rotation: CW180

.x

xx

.x

SHAPE x

height: 2; width: 3; rotation: CW270

.x.

xxx

It should be clear however that Shape t and Shape x are equivalent under rotations in terms of how they can be fit into a space.

Some shapes do not change on every rotation. This may lead to some redundancy in during search and among solutions to fitting problems but you are not required to deal with this in any special way. Examples:

SHAPE T

TTTT

SHAPE T

height: 1; width: 4; rotation: CW0

TTTT

SHAPE T

height: 4; width: 1; rotation: CW90

T

T

T

T

SHAPE T

height: 1; width: 4; rotation: CW180

TTTT

SHAPE T

height: 4; width: 1; rotation: CW270

T

T

T

T

SHAPE r

rr

rr

SHAPE r

height: 2; width: 2; rotation: CW0

rr

rr

SHAPE r

height: 2; width: 2; rotation: CW90

rr

rr

SHAPE r

height: 2; width: 2; rotation: CW180

rr

rr

SHAPE r

height: 2; width: 2; rotation: CW270

rr

rr

The closing example of the last section involved the following space and shapes for which there was no fit without rotations.

SPACE

|.......

..|||..|

..|.|...

........

SHAPE T

TTTT

SHAPE e

..e

eee

SHAPE t

.t.

ttt

SHAPE r

rr

rr

SHAPE i

i..

iii

SHAPE s

.ss

ss.

However, with the introduction of rotations, a solution does exist. Some additional detail is given at the bottom indicating the state of each shape found.

FOUND FIT

SPACE:

height: 4 width: 8

|tTTTTss

tt|||ss|

it|.|err

iiieeerr

6 shapes placed

Shape at (0,2)

SHAPE T

height: 1; width: 4; rotation: CW0

TTTT

Shape at (2,3)

SHAPE e

height: 2; width: 3; rotation: CW0

..e

eee

Shape at (2,0)

SHAPE i

height: 2; width: 3; rotation: CW0

i..

iii

Shape at (2,6)

SHAPE r

height: 2; width: 2; rotation: CW0

rr

rr

Shape at (0,5)

SHAPE s

height: 2; width: 3; rotation: CW0

.ss

ss.

Shape at (0,0)

SHAPE t

height: 3; width: 2; rotation: CW270

.t

tt

.t

**3 Implementation Issues**

**3.1 Representation of Spaces and Shapes**

One of your primary tasks in solving this problem is to design classes that represent the notions of shapes and spaces. Unlike previous projects, you are not required to use any specific internal structure of the data for these classes. The section [Required Design Elements](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#required) contains some required methods as specified in a few interfaces, but aside from the presence of the public methods mentioned there, you are free to implement your solution using any number of classes you feel is necessary. The public interfaces required give some guidance as to what should be done but you will need to do some planning about how to surmount the problems faced by these classes. Your solutions will need to be described in your [design.txt document](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#design-doc). Consider the following issues while designing your classes.

* Shapes and Spaces need a way to keep track of their size (height and width) along with which blocks in their grid are filled and empty. There are several ways to do this such as using arrays, using lists of pairs, or other classes.
* Shapes will need to be rotated (Spaces do not rotate). On rotation, the Shapes size (height/width) may change and its configuration of filled and empty blocks may also change. You only need to accommodate single clock-wise rotations by 90 degrees but this may happen multiple times. You will need to do some research to figure out how to transform the blocks in a Shape to reflect the rotation that was done. Some internet research may be helpful for this but **make sure to include a comment string with any sources drawn from**. The test cases may also eventually contain some code which is useful for rotation so you are encouraged to examine them.
* Spaces will need a way to detect whether a Shape can fit at a given location or if any filled blocks in the shape would be out of bounds, overlap with a permanently filled block, or overlap with the filled blocks of another shape that has been placed in the space. Code-wise this will look the following.
* Shape shape = ...;
* Space space = ...;
* boolean fits = space.shapeFitsAt(1,3,shape);
* // true if the shape would fit, false otherwise
* Spaces will need a way to add shapes to themselves. Adding is done via placement of the shape in the space by passing in a shape along with its placement location. This uses the shape's current rotation.
* Shape aShape = ...;
* Space space = ...;
* space.placeShapeAt(1,3,aShape);
* // space now contains aShape
* Spaces can produce string representations of themselves e.g. using the fitString() method and list out their currently placed shapes using the placedShapeInfo(). Placed shapes must be shown sorted by their display character.
* Spaces can remove shapes by their display character. For example:
* Shape aShape = ...; // display character is a
* Space bShape = ...; // display character is b
* space.placeShapeAt(1,3,aShape);
* space.placeShapeAt(4,0,bShape);
* space.removeShapeByDisplayChar(aShape.getDisplayChar());
* // space no longer contains aShape
* space.removeShapeByDisplayChar('b');
* // space no longer contains bShape
* Shapes and Spaces will be initialized using a **layout string** that is described in more detail in the [Layout String Formats](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#layout-strings) section.

**3.2 Recursive Search for Fit**

The primary objective of your implementation is to search for a fit of all of a given list of shapes into a space. You will specify a method in the FitIt class called

Space searchForFit(Space space, List<Shape> unplaced)

which will return either null if no fit exists or a space which is filled with the given list of shapes.

searcForFit() is best implemented recursively. It solves a search problem involving **backtracking**: while trying out candidate placement for shape X, another shape Y may not also be able to fit, in which case the method must back up and attempt to try another placement of shape X.

As an example, consider the following problem.

SPACE

..

..

.|

SHAPE a

aa

SHAPE b

bbb

An obvious strategy for an algorithm is to try placing Shape a at all possible locations with all possible rotations, then attempt the same for Shape b. For example a first attempt might place Shape a at (0,0) with no rotation (CW0).

Trying a rot:CW0 at (0,0)... Fits

aa

..

.|

Unfortunately, this does not allow Shape b to be fit into the space. Consider all the possible location/rotation combinations which do not work.

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

At this point, there is no choice but to give up on the original placement of Shape a

Giving up on shape b in

aa

..

.|

and try a different placement of it. Importantly, this is the notion of **backtracking**: one solution path did not work out so we must back up and try another direction to see if it leads to more success.

Trying a rot:CW90 at (0,0)... Fits

a.

a.

.|

To most humans, this placement for Shape a is equally as bad and will result in no fit being found for Shape b. This is because most humans are good at small constraint problems like this and recognize where Shape a should be placed. However, any such intuition would need to be encoded into an algorithm and it is far easier to simply enumerate all possible placements of shapes.

Eventually, the following placement of Shape a is made.

Trying a rot:CW90 at (0,1)... Fits

.a

.a

.|

This is the choice most humans would make right off the bat. After this, there is not much left in terms of search.

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... Fits

ba

ba

b|

All shapes placed

FOUND FIT

SPACE:

height: 3 width: 2

ba

ba

b|

2 shapes placed

Shape at (0,1)

SHAPE a

height: 2; width: 1; rotation: CW90

a

a

Shape at (0,0)

SHAPE b

height: 3; width: 1; rotation: CW90

b

b

b

**3.3 Backtracking and Recursive Search**

The notion of backtracking is rather easy to implement in a recursive method if you identify two **base cases**: a base case for success and a base case for failure. Complementing these two base cases is a **recursive** case in which calls for further search via a recursive call. In a recursive implementation of searchForFit(space,unplacedShapes) these cases are roughly as follows.

Success Base Case

The list unplacedShapes is empty so all shapes have been placed and the space currently contains a solution that should be returned.

Failure Base Case

The list unplacedShapes is not empty. A single Shape S is removed from it but there is no place in space to fit S. No solution exists for the current configuration of space so a failure should be signaled by returning null. S should go back into the list of unplacedShapes.

Recursive Case

The list unplacedShapes is not empty. A single Shape S is removed from it and a valid placement is found for S in space. Place S at the valid location in space, remove it from the unplacedShapes list, and recursively call searchForFit(). If the recursive call results in a success, return it. If it results in a failure, try a different placement of S in space.

Do not be seduced into thinking that the above description can be mechanically translated into a code version of searchForFit(): you will need to master the ideas presented there and experiment with the ordering of operations in order to implement the method correctly. This will take some time and effort.

A great boon while coding this effort is the ability to see exactly what is going on at any given moment. You are **highly encouraged** to use debugging print statements that allow you to see the progress of your algorithm. As an example, here is the whole debugging output for the problem in the previous section. You may use this as a model for your own debugging output or use a different style. It is very likely that teaching staff will want to see debugging output if you need help on searchForFit() and you will expedite your help by providing it.

SPACE:

height: 3 width: 2

..

..

.|

0 shapes placed

2 shapes read

SHAPE a

height: 1; width: 2; rotation: CW0

aa

SHAPE b

height: 1; width: 3; rotation: CW0

bbb

Trying a rot:CW0 at (0,0)... Fits

aa

..

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

aa

..

.|

Removing a rot:CW0 from (0,0)

Trying a rot:CW90 at (0,0)... Fits

a.

a.

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

.|

Removing a rot:CW90 from (0,0)

Trying a rot:CW180 at (0,0)... Fits

aa

..

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

aa

..

.|

Removing a rot:CW180 from (0,0)

Trying a rot:CW270 at (0,0)... Fits

a.

a.

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

.|

Removing a rot:CW270 from (0,0)

Trying a rot:CW0 at (0,1)... No fit

Trying a rot:CW90 at (0,1)... Fits

.a

.a

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... Fits

ba

ba

b|

All shapes placed

FOUND FIT

SPACE:

height: 3 width: 2

ba

ba

b|

2 shapes placed

Shape at (0,1)

SHAPE a

height: 2; width: 1; rotation: CW90

a

a

Shape at (0,0)

SHAPE b

height: 3; width: 1; rotation: CW90

b

b

b

**4** **Required Design Elements**

**4.1** **Top-level Methods: FitIt.java**

There are four required methods in the FitIt class which you must implement, though two of these simply call constructors you will write in your own designed classes. These methods are shown below in the overview of FitIt.java. They reference several of the interfaces shown subsequently. The [Layout String and File Format](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#layout-strings) section describes in more detail the type of string that will be passed in to the the makeShape() and makeSpace() method and the file format for input to main().

public class FitIt{

// Factory methd to produce shapes. Call a constructor for one of

// your own classes here. The layout given is be in the format

//

// ..e

// eee

//

// Newlines (\n) show breaks in the rows of the Shape. The period

// (.) is used to represent empty blocks. Any other character is a

// filled block even if it does not match the given displayChar.

// The Shape returned by this method should have the display

// character displayChar even if that character does not appear in

// the layout string. Shapes should be initialized to have rotation

// CW0. If the shape contains any empty border sides, throw a

// FitItException with an informative message.

public static Shape makeShape(String layout,char displayChar);

// Factory method to produce instances of space. Call a constructor

// for one of your own classes here. The layout parameter will be in

// the format

//

// |.......

// ..|||..|

// ..|.|...

// ........

//

// where vertical bars are filled blocks and periods are empty blocks.

public static Space makeSpace(String layout);

// Search for a fit of the given shapes in the given space. Return

// null if no fit exists. If a fit is found, return a space with all

// shapes placed in it. It is very useful for this method to be

// recursive.

public static Space searchForFit(Space space, List<Shape> unplaced);

// Search for a all fits accumulating answers in the answers

// parameter list.

public static void searchForAllFits(Space space, List<Shape> unplaced, List<String> answers);

// Read an input file which contains a fitting problem in it. The

// input file is the zeroth command line argument. The file format

// contains records for SPACE and SHAPE. There should only be one

// SPACE per file but potentially many SHAPEs. SHAPE is followed by

// a display character for the shape. SPACE and SHAPE records

// continue until a black line. Any line that does not start with

// SPACE or SHAPE should be ignored. The main method should read

// this file, initialize spaces and shapes using the methods

// makeShape() and makeSpace(). It should then execute

// searchforFit(space,shapes) and report the results as either

//

// NO FIT FOUND

//

// or

//

// FOUND FIT

// then call the toString() method of space

//

// Main should take one or two arguments. The first argument is an

// input problem file. If there is only one argument, print to the

// screen. Example:

//

// > java FitIt problem.txt

//

// If there are two arguments, the second argument is the name of a

// file that should be printed into rather than printing to the

// screen. Example

//

// > java FitIt problem.txt output.txt

//

// If the first argument refers to a non-existant file, throw any

// kind of exception.

public static void main(String args[]) throws Exception;

}

**4.2 Shape Interface**

The Shape interface is provided in the code pack but like all interfaces, it specifies only what an implementing class can do, not how to do it. You will need to create a class or classes which implements Shape and is returned by the FitIt.makeShape() method. Some methods refer to the Rotation enumeration which is described in a subsequent section. All classes implementing Shape must have the following methods.

// A shape is a rectangular grid of filled and empty blocks. They can

// be rotated clockwise (CW) in increments of 90 degrees and track

// their rotation using the Rotation enumeration. Shapes can be

// displayed in text terminals and have a display character which

// dictates how they are shown.

public interface Shape{

// Return the number of rows of the shape

public int getHeight();

// Return the number of columns of the shape

public int getWidth();

// When drawing this shape in a text terminal, the character

// returned by displayChar will be used

public char getDisplayChar();

// Set how the shape will be displayed in text terminals

public void setDisplayChar(char c);

// Rotate this shape clockwise by 90 degrees. Adjust all internal

// state required to achieve the rotation including height and width

public void rotateCW();

// Return a value from the Rotations enumeration indicating how

// much the shape has been rotated from its original position.

public Rotation getRotation();

// Return true if the shape has a filled block at the given row/col

// position and false if the block is empty. If the position is out

// of bounds, raise a FitItException with an informative message.

public boolean isFilledAt(int row, int col);

// Create a string representation of the shape. The format must

// follow this convention:

//

// SHAPE c

// height: 2; width: 3; rotation: CW270

// ..c

// ccc

public String toString();

}

**4.3 Space Interface**

The Space interface is also provided in the code pack and you will need to implement one or more classes which cooperate to fulfill the interface requirements.

// A Space is a rectangular grid of empty and filled blocks. On being

// initialized, some blocks of a space are permanently filled.

// Instances of Shape may be placed in the space so long as they are

// in bounds and do not conflict with any filled blocks. Space classes

// provide mechanisms to check whether shapes fit at a location in the

// space, place shapes at a location, remove shapes, and display the

// present contents of the space.

public interface Space{

// Return the number of rows of the space

public int getHeight();

// Return the number of columns of the space

public int getWidth();

// Return true if the space has a filled block at the given row/col

// position and false if the block is empty. A block may be filled

// if it is permanently filled or if a shape has been placed in a

// position such that the space's block is filled by the shape's block

// If the position is out of bounds, return true as there is implicit

// fill all around the space. DO NOT THROW EXCEPTIONS from this

// method.

public boolean isFilledAt(int row, int col);

// Determine if the given shape may be placed at the indicated

// row/col position. The row,col indicates where the upper left

// corner [block (0,0)] of the shape would be placed. A shape would

// not fit if one of its filled blocks would conflict with an

// existing filled block in the space or would be out of bounds in

// the space.

public boolean shapeFitsAt(int row, int col, Shape shape);

// Place the given shape at the given row/col position. The row,col

// indicates where the upper left corner [block (0,0)] of the shape

// would be placed. The shape should be removable and when

// displaying info on the placed shapes, they must be shown in order

// of their display characters (shape A before B before C...).

// There may be more than one shape in a space with the same display

// character. If the shape would not fit at the specified row/col

// position, raise a FitItException with an informative message.

public void placeShapeAt(int row, int col, Shape shape);

// Remove the shape with the display character indicated by dc from this

// space. Update all internal state so that blocks in the space that

// were filled by the removed shape are emptied.

public void removeShapeByDisplayChar(char dc);

// Return how many shapes have been placed in this space.

public int placedShapeCount();

// Return a string representing the space and the shapes that have

// been fit into it. The following character conventions must be

// used.

// | : vertical bar for permanently filled blocks

// . : period for empty blocks

// displaychar : any space filled by a shape uses its display char

//

// Example:

// aa.....e

// aacddd.|

// cccdd|||

public String fitString();

// Give a listing of all the placed shapes. This should start with

// the number of placed shapes, show the position of each shape, and

// use the toString() of each shape. Shapes must be reported in

// sorted order based on their display character (shape A before B

// before C...).

//

// Example:

//

// 3 shapes placed

// Shape at (0,0)

// SHAPE a

// height: 2 width: 2 rotation: CW90

// aa

// aa

//

// Shape at (0,2)

// SHAPE b

// height: 1 width: 4 rotation: CW180

// bbbb

//

// Shape at (1,0)

// SHAPE c

// height: 2 width: 3 rotation: CW270

// ..c

// ccc

public String placedShapeInfo();

// Print a verbose string representation of the space. Start with

// the string SPACE: then follow with the fitString() of the space

// and finally the placedShapeInfo() string.

//

// Example:

//

// SPACE:

// height: 3 width: 8

// aabbbbfe

// aacdddf|

// cccdd|||

//

// 6 shapes placed

// Shape at (0,0)

// SHAPE a

// height: 2 width: 2 rotation: CW90

// aa

// aa

//

// Shape at (0,2)

// SHAPE b

// height: 1 width: 4 rotation: CW180

// bbbb

// ...

public String toString();

}

**4.4 Rotation Enumeration**

Rotation is a simple enumeration used to track the four valid clock-wise rotations: 0, 90, 180, and 270 degrees. It has four elements and a single method.

public enum Rotation{

CW0, CW90, CW180, CW270;

// Calling rot.next() will return the next enumeration element

// representing the next 90 degree clock-wise rotation after rot.

public Rotation next();

}

**4.5 Exception Classes**

Certain methods of Space must throw exceptions. In order to identify that you are identifying these situations and responding appropriately, throw the provided FitItException which can be simply initialized with a string message. **Ensure** that you are using messages that are informative and correspond to the situation that has arisen.

// Catch-all exception for Shapes and Spaces so that explicit

// exception throwing can be checked

public class FitItException extends RuntimeException{

public FitItException(String msg){

super(msg);

}

}

**5 Layout String and File Formats**

**5.1** **Layout String Formats**

The makeShape(String layout, char displayChar) method of FitIt (described in the [FitIt section](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#fit-it)) takes a layout string as an argument and should produce a shape. This layout string (and the display character argument) will be passed into a constructor for a class you devise. Layout strings are identical the examples used elsewhere in the specification. Examples are below, each separated by a blank line.

..e

eee

.ss

ss.

b

b

b

b

aaaa

a..a

aaaa

.....z

....z.

...z..

zz..z.

All of these are valid layout strings which should produce valid shapes on a call to makeShape(layout,dc).

Shape layout strings have the following format.

* Newlines (\n) show breaks in the rows of the Shape.
* The period (.) is used to represent empty blocks.
* Any other character is a filled block even if it does not match the given displayChar. By convention, no tests or input files will do this but it may be useful for your own testing.
* All lines may be assumed to have the same number of characters on them. Your code may handle strings that violate this assumption however you wish (produce a shape, throw an exception, otherwise misbehave). An Example of a "ragged" format string that is illegal is
* ppp
* p
* pp

because it does not contain the period (.) to denote blank spaces at the ends of lines. A proper layout would be

ppp

p..

pp.

Layout strings for shapes must have at least one filled block in each of 0th row, 0th column, last row, and last column.

Shapes will be restricted to contain a non-empty border: the 0th row, 0th column, last row, and last column must all contain at least one filled block. This prevents problems when dealing with [rotation](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#rotation) and [placement in spaces](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#placement). The following shapes are illegal due to having at least part of their border empty. This means the following layout strings are all illegal.

..b.

bbb.

...

.x.

...

..r

.rr

.....

.y...

..y..

...y.

If passed to makeShape(layout,dc) as the layout, a FitItException should be raised for such strings.

**5.2 Space Layout Strings**

The method makeSpace(String layout) of FitIt takes a layout string and returns a Space. The string will be formatted similarly to layouts for shapes except that filled blocks are always shown with the pipe/vertical bar (|) character. Examples are as follows (separated by blank lines).

..

..

.|

......

.....|

...|||

|.......

..|||..|

..|.|...

........

Space layout strings have the following format.

* Newlines (\n) show breaks in the rows of the Shape.
* The period (.) is used to represent empty blocks.
* The pipe (|) is used to represent permanently filled blocks.
* All lines may be assumed to have the same number of characters on them.

**5.3 Problem Input Formats**

FitIt.main(String args[]) takes a command line argument which is an input file describing a problem to solve. The input files have the following format.

Space Records

Any line starting with SPACE denotes the beginning of a space. Subsequent lines constitute the layout string for a space ending with a blank line. Example:

SPACE

........

.......|

.....|||

The space layout string should be created by appending subsequent lines together and then passed to the makeSpace(layout) method. The space is used in a call to searchForFit(space,shapeList). If more than one SPACE record appears in an input file, your code may behave in any way (replace previous space, throw an exception, fail utterly).

Shape Records

Any line starting with SHAPE denotes the beginning of a shape record. On the line with SHAPE will be a single character which is the display character for the shape. Subsequent lines constitute the layout string for the shape ending with a blank line. Examples:

SHAPE b

bbbb

SHAPE c

c.

c.

cc

The lines after SHAPE should be appended to create a layout string which, along with the display character, is passed to a the makeShape(layout,displayChar) method. The shapes should be added into a list which is used in a call to searchForFit(space,shapeList).

Other Lines

Any line that that is not part of a Shape or Space record and does not start such a record by beginning with SPACE or SHAPE should be ignored. Lots of comments and solutions can appear in the files. The following lines will be ignored.

SOLUTION

aadddffe

aaddccc|

bbbbc|||

A = 0 0 CW0

B = 2 0 CW0

C = 1 4 CW90

D = 0 2 CW180

E = 0 7 CW0

F = 0 5 CW90

Some random text here

which will be ignored

Here is a complete example of an input file which is provided with the code as problem5.txt

SPACE

..

..

.|

SHAPE a

aa

SHAPE b

bbb

Any text that doesn't start with SPACE or SHAPE should be

ignored. That includes these lines and the SOLUTION below which is for

documentation purposes only.

SOLUTION

ba

ba

b|

**6 Sample Runs of FitIt.main()**

The exact output FitIt.main() will not be closely checked but it gives a good sense of how the program behaves in terms of listing solutions or not. The input files are shown first, then the program is invoked on the problem file. In the instructor solution, a second argument is checked for the string debug in which case more output is produced. This is a pattern you should consider adopting in your own implementation.

lila [p6]% java FitIt

usage: java FitIt input.txt

lila [p6]% cat samples/sample-problem.txt

SPACE

..

..

.|

SHAPE a

aa

SHAPE b

bbb

Any text that doesn't start with SPACE or SHAPE should be

ignored. That includes these lines and the SOLUTION below which is for

documentation purposes only.

SOLUTION

ba

ba

b|

lila [p6]% java FitIt samples/sample-problem.txt

Finished with input

SPACE:

height: 3 width: 2

..

..

.|

0 shapes placed

2 shapes read

SHAPE a

height: 1; width: 2; rotation: CW0

aa

SHAPE b

height: 1; width: 3; rotation: CW0

bbb

FOUND FIT

SPACE:

height: 3 width: 2

ba

ba

b|

2 shapes placed

Shape at (0,1)

SHAPE a

height: 2; width: 1; rotation: CW90

a

a

Shape at (0,0)

SHAPE b

height: 3; width: 1; rotation: CW90

b

b

b

lila [p6]% cat samples/impossible-problem.txt

This problem has no solution

SPACE

..

..

.|

SHAPE a

aaa

SHAPE b

bbb

lila [p6]% java FitIt samples/impossible-problem.txt

Finished with input

SPACE:

height: 3 width: 2

..

..

.|

0 shapes placed

2 shapes read

SHAPE a

height: 1; width: 3; rotation: CW0

aaa

SHAPE b

height: 1; width: 3; rotation: CW0

bbb

NO FIT FOUND

lila [p6]% java FitIt samples/sample-problem.txt debug

Trying a rot:CW0 at (0,0)... Fits

aa

..

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

aa

..

.|

Removing a rot:CW0 from (0,0)

Trying a rot:CW90 at (0,0)... Fits

a.

a.

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

.|

Removing a rot:CW90 from (0,0)

Trying a rot:CW180 at (0,0)... Fits

aa

..

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

aa

..

.|

Removing a rot:CW180 from (0,0)

Trying a rot:CW270 at (0,0)... Fits

a.

a.

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

.|

Removing a rot:CW270 from (0,0)

Trying a rot:CW0 at (0,1)... No fit

Trying a rot:CW90 at (0,1)... Fits

.a

.a

.|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... Fits

ba

ba

b|

All shapes placed

lila [p6]% java FitIt samples/impossible-problem.txt debug

Trying a rot:CW0 at (0,0)... No fit

Trying a rot:CW90 at (0,0)... Fits

a.

a.

a|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

a|

Removing a rot:CW90 from (0,0)

Trying a rot:CW180 at (0,0)... No fit

Trying a rot:CW270 at (0,0)... Fits

a.

a.

a|

Trying b rot:CW0 at (0,0)... No fit

Trying b rot:CW90 at (0,0)... No fit

Trying b rot:CW180 at (0,0)... No fit

Trying b rot:CW270 at (0,0)... No fit

Trying b rot:CW0 at (0,1)... No fit

Trying b rot:CW90 at (0,1)... No fit

Trying b rot:CW180 at (0,1)... No fit

Trying b rot:CW270 at (0,1)... No fit

Trying b rot:CW0 at (1,0)... No fit

Trying b rot:CW90 at (1,0)... No fit

Trying b rot:CW180 at (1,0)... No fit

Trying b rot:CW270 at (1,0)... No fit

Trying b rot:CW0 at (1,1)... No fit

Trying b rot:CW90 at (1,1)... No fit

Trying b rot:CW180 at (1,1)... No fit

Trying b rot:CW270 at (1,1)... No fit

Trying b rot:CW0 at (2,0)... No fit

Trying b rot:CW90 at (2,0)... No fit

Trying b rot:CW180 at (2,0)... No fit

Trying b rot:CW270 at (2,0)... No fit

Trying b rot:CW0 at (2,1)... No fit

Trying b rot:CW90 at (2,1)... No fit

Trying b rot:CW180 at (2,1)... No fit

Trying b rot:CW270 at (2,1)... No fit

Giving up on shape b in

a.

a.

a|

Removing a rot:CW270 from (0,0)

Trying a rot:CW0 at (0,1)... No fit

Trying a rot:CW90 at (0,1)... No fit

Trying a rot:CW180 at (0,1)... No fit

Trying a rot:CW270 at (0,1)... No fit

Trying a rot:CW0 at (1,0)... No fit

Trying a rot:CW90 at (1,0)... No fit

Trying a rot:CW180 at (1,0)... No fit

Trying a rot:CW270 at (1,0)... No fit

Trying a rot:CW0 at (1,1)... No fit

Trying a rot:CW90 at (1,1)... No fit

Trying a rot:CW180 at (1,1)... No fit

Trying a rot:CW270 at (1,1)... No fit

Trying a rot:CW0 at (2,0)... No fit

Trying a rot:CW90 at (2,0)... No fit

Trying a rot:CW180 at (2,0)... No fit

Trying a rot:CW270 at (2,0)... No fit

Trying a rot:CW0 at (2,1)... No fit

Trying a rot:CW90 at (2,1)... No fit

Trying a rot:CW180 at (2,1)... No fit

Trying a rot:CW270 at (2,1)... No fit

Giving up on shape a in

..

..

.|

**7 Honors Section (15%)**

**7.1 Finding All Fits**

In the main class FitIt, we will add one more method, whose purpose is to find all solutions. We will try not to make too many additions, so this method is like searchForFits except it adds an extra parameter, a List<String>, that will be filled with the fitString()'s of all unique fitting solutions.

**public static void searchForAllFits(Space space, List<Shape> unplaced, List<String> answers)**

* keep collecting newly-found solutions in answers.
* the return type is indeed void; whoever called it should just look in answers for the… answers.
* don't allow duplicates.
  + in FitIt.main, we will choose to sort the results (you can just use Collections.sort(), from java.util.Collections). Sorting is not searchForAllFits' job.
* this "all fits" version is called by including the word "all" as the second argument to FitIt.main. For instance:
* java FitIt samples/problem4.txt all

When "all" is present, we must print how many fits were found, and then all the fitString results in *sorted* order. See the sample runs next.

* if the second argument is not "all", then interpret instead as an output file to be printed into rather than on the screen.
* java FitIt samples/problem4.txt output.txt

This invocation should produce one fit and print all output to the file output.txt.

* if the second argument IS the word "all" and there is a third argument, the find all fits and print output to the third argument which is a file for output.
* java FitIt samples/problem4.txt all output.txt

This invocation should find all fits and print all output to the file output.txt.

**7.2 Sample Runs**

demo$ javac \*.java

demo$ java FitIt samples/problem2.txt all

2 FITS FOUND

|eeeTTTT

se|||ii|

ss|t|irr

.stttirr

|tTTTTss

tt|||ss|

it|.|err

iiieeerr

demo$ java FitIt samples/problem4.txt all

4 FITS FOUND

aabbb

acc|b

bbbaa

ccb|a

bccaa

bbb|a

cabbb

caa|b

demo$ java FitIt samples/impossible-problem.txt all

0 FITS FOUND

demo$

**8 Grading**

As before, 50% of your grade will be determined by automated tests while the other 50% will be according to a manual inspection.

**8.1 Automated Tests (50%)**

We're providing a set of *unit tests*, which may be updated as needed until the deadline, which you should use frequently while testing and implementing your code. Grading will be a simple fraction of tests passed. If your code doesn't compile against the tests, you might receive most or all credit from the automated portion of the grade. Small enough fixes might be corrected by your grader, but any significant modifications needed will not be performed - the grader is not tasked with fixing your code.

**8.2 Manual Inspection (50%)**

We'll also manually inspect your code to check for certain commenting, formatting, and coding style. This portion of the grade will also enforce different requirements of the spec that are not easily captured in a JUnit test case. Below is the breakdown of points earned through manual inspection.

**9 Manual Inspection Criteria**

**9.1 Correct Project Setup (5%)**

Correctly setting up the directory structure for each project greatly eases the task of grading lots of projects. Graders get cranky when strange directory structures or missing files are around and you do not want cranky graders. The following will be checked for setup on this project.

* The [Setup](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10) instructions were followed closely
* The [Project Directory](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10-1) is named according to the specification
* There is an [Identity Text File](http://cs.gmu.edu/%7Ekauffman/cs211/p6.html#sec-10-2) present with the required information in it
* Code can be compiled and tests run from the command line

**9.2 Coding Style and Readability (5%)**

This is a larger project. It will require discipline and effort to keep track of how all the pieces fit together. Commenting your own code to keep track of the purpose of fields and methods will pay much higher dividends in this project than was previously the case. Your code will be inspected for clarity in the following categories.

* **Code Cleanliness (1%)**: Indent and {bracket} code uniformly throughout the program to improve readability.
* **Class Documentation (1%)**: Each class has an initial comment indicating its intended purpose, how it works, and how it relates or uses other classes in the project.
* **Field Documentation (1%)**: Each field of a class is documented to indicate what piece of data is tracked and how it will be used, regardless of visibility.
* **Method documentation (2%)**: Each method has a short description indicating its intended purpose and how it gets its job done. (These are also needed for your own helper methods you choose to add).

**9.3** **design.txt Design Documentation (10%)**

For this project, you must describe your overall design to solve the problem in a text file you submit called design.txt. This document should contain the following sections.

DESCRIBE WHAT CLASSES WERE USED TO IMPLEMENT THE Shape

INTERFACE. DETAIL ANY CLASSES YOU CREATED FOR THIS PURPOSE. DESCRIBE

THE FIELDS THESE CLASSES CONTAIN AND HOW WERE THEY USED TO IMPLEMENT

isFilledAt(r,c) AND rotateCW(). BE SPECIFIC ABOUT HOW YOU ACHIEVED

ROTATION.

DESCRIBE WHAT CLASSES WERE USED TO IMPLEMENT THE Space

INTERFACE. DETAIL ANY CLASSES YOU CREATED FOR THIS PURPOSE. DESCRIBE

THE FIELDS THESE CLASSES CONTAIN AND HOW WERE THEY USED TO IMPLEMENT

THE METHODS

- isFilledAt(r,c)

- removeShapeByDisplayChar(dc)

- placeShapeAt(r,c)

BE SPECIFIC WHEN DESCRIBING placeShapeAt(r,c)

Your descriptions should be in the 4-8 sentence range. They are an overview of the design decisions you made in each case so that someone reading your code will have a good sense of what to look for: which classes will be present and what will generally be inside them (arrays, ArrayLists, pairs, additional classes, etc.).

Your descriptions are not a substitute for informative comments within each class and associated with each method. Do not forget these.

**9.4 Adherence to Interface Boundaries within Space (5%)**

The classes you use to implement Space should deal exclusively with objects implementing Shape, not with any specific class you design. No casting should be done. Your code will be checked to guarantee that you are using only methods that appear in the Shape interface while manipulating shapes.

**9.5 Adherence to Interface Boundaries within searchForFit() (5%)**

FitIt.searchForFit(space,shapes) should deal with shapes and spaces using only the methods specified in the interfaces Space and Shape so that if an alternative implementation was provided, the method would still work. No casting should be done. Your code will be checked to ensure that it is compliant and not using special methods of your own classes.

**9.6 Adherence to Interface Boundaries within main() (5%)**

FitIt.main(args) should call the high-level methods makeShape(layout), makeSpace(layout) and searchForFit(space,shapes). No casting should be done. Your code will be checked to ensure that it is compliant and not using special methods of your own implementation classes.

**9.7 Elegance of searchForFit(space,shapeList) (10%)**

Searching for fits is the main purpose of this project. Work hard to make the search method as clean as possible. Hints:

* The method does not need to be very long: if you are exceeding 50 lines, you are probably heading down the wrong line of inquiry.
* There will be a combination of iteration and recursion. Iteration is used to drill through all possible locations and rotations for shapes in a space. Recursion is used to explore a space after placing a shape to see if the placement works out.
* The fact that the space, the shapeList, and all Shapes are mutable might cause some problems as they may change between recursive calls. Be very careful between calls that you are restoring state as required: putting things back in lists, removing shapes from spaces, and rotating shapes the proper number of times will all be necessary.
* Include debug printing statements that indicate what your method is doing at any given step. This will help you to diagnose any problems.

**9.8 Elegance of Space.fitString() (5%)**

Constructing the string representation of a Space requires some work as you must deal with empty blocks, permanently filled blocks, and blocks filled by placed shapes. Your code will be reviewed for taking a reasonable approach to this problem. Some hints:

* Start with a 2D array of characters representing empty (.) and filled (|)
* For every shape that has been placed, fill in the positions in the 2D array with characters corresponding to the filled blocks of the shape.
* Append all of the characters in the 2D array into a final return string.

**9.9 Honors 15% Inspection**

We will rely upon test cases for 10% of the honors grade, and 5% will be dedicated to the approach of your solution to searchForAllFits. You will add a third paragraph (of similar size to the others) to your design document; in it, you must address the following:

* What extra infrastructure, if any, did you need to add in order to make this extension work?
* How did you address duplicates?
* How did you address the extra command line argument versus the other one
* How did you continue searching even when a match was found?

**10 Setup**

**10.1 Project Directory**

Create a directory (folder) with the structure of netID-LabSecNum-pNum, where:

* ***netID*** is your GMU ID (part of your email address, **not** your G-number).
* ***labSecNum*** is your lab section number, such as 201, 202, 2H1, etc. *Don't put your lecture number here by mistake!*
* ***pNum*** represents the project number, such as p6.
* example: msnyde14-2H1-p6
* example: ckauffm2-205-p6
* example: rcarver-201-p6

This is your **project directory.** Everything concerning your project will go in this directory.

**10.2 Identity Text File**

Create a text file in your project directory called ID.txt which has identifying information in it. My ID.txt looks like this:

Mark Snyder

msnyde14

G0071234

Lecture: 001

Lab: 205

It contains my full name, Net ID, G#, lecture section, and lab section.

Failure to submit your work in an appropriately named directory and with the ID.txt file in it with correct information will result in the loss of 5%.

**10.3 Provided Files**

**Download** [p6pack.zip](http://cs.gmu.edu/%7Ekauffman/cs211/p6pack.zip) once it is available and put its contents in your project directory. Use it for testing as with previous projects.

**11 Submission**

Create a zip file of your project directory and submit it to the course Blackboard page. **Do not e-mail the professor or TAs your code**.

Make sure your directory is set up as described in the Setup section. Also make sure that all .java files are present there; .class files (compiled bytecode) and .java~ (auto-save files) will not be accepted, as we cannot inspect the code you have written. There should be these files present inside the directory (no ordering implied, of course) along with any other source code files you create to complete the problem.

* netID\_LabSecNum\_pNum/
  + Shape.java
  + Space.java
  + FitIt.java
  + FitItException.java
  + Rotation.java
  + design.txt
  + P6Tests.java (optional)
  + samples/ (optional)
    - <many \*.txt files…>
  + junit-cs211.jar (optional)
  + Your own .java files which implement the required interfaces

Zip that folder as a .zip archive and submit it to Blackboard:

* Click on the Assignments Link
* Click on the correct submission link for this project
* Scroll down to "Attach a File"
* Click "Browse My Computer"
* Select your zip file

You may submit as many times as you like; all submissions later than 48 hours past the deadline are invalid and thus ignored entirely, so the latest remaining valid submission is the only one to be graded. If your latest valid submission is a late submission, any tokens remaining will be automatically applied.

You should *always* verify that your submission is successful. Go back to the assignment and download your last submission; is there actually a file there? Can you unzip that file, and see .java files in there as you'd expect? Remember, .class and .java~ files are absolutely worthless for grading purposes. Can you run the submitted files? It turns out you can do quite a bit to ensure you've properly submitted your work. Don't get a zero just because of a failure to actually turn in the right files! It is your responsibility to correctly turn in your work.

**12 Setup**

**12.1 Project Directory**

Create a directory (folder) with the structure of netID-LabSecNum-pNum, where:

* ***netID*** is your GMU ID (part of your email address).
* ***labSecNum*** is your lab section number, such as 201, 202, 2H1, etc. *Don't put your lecture number here by mistake!*
* ***pNum*** represents the project number, such as p6.
* example: msnyde14-2H1-p6
* example: ckauffm2-205-p6
* example: rcarver-201-p6

This is your **project directory.** Everything concerning your project will go in this directory.

**12.2 Identity Text File**

Create a text file in your project directory called ID.txt which has identifying information in it. My ID.txt looks like this:

Mark Snyder

msnyde14

G0071234

Lecture: 001

Lab: 205

It contains my full name, Net ID, G#, lecture section, and lab section.

Failure to submit your work in an appropriately named directory and with the ID.txt file in it with correct information will result in the loss of 5%.

**12.3 Provided Files**

**Download** [p6pack.zip](http://cs.gmu.edu/%7Ekauffman/cs211/p6pack.zip) once it is available and put its contents in your project directory. Use it for testing as with previous projects.

*Author: Mark Snyder, Chris Kauffman, Richard Carver (*[*msnyde14@gmu.edu*](mailto:msnyde14@gmu.edu)*,* [*kauffman@cs.gmu.edu*](mailto:kauffman@cs.gmu.edu)*,* [*rcarver@mason.gmu.edu*](mailto:rcarver@mason.gmu.edu)*)   
Date: 2015-04-24 Fri 09:53*