HW3 DB/Sudoku

Thanks to Nick Parlante for much of this handout

Homework 3 makes extensive use of the Java Collection classes while exercising OOP themes -- assembling a large solution out of modular classes, giving each class a good API for its clients, testing, and some GUI coding. The whole thing is due Wed Oct 22nd.

Part A - Database

First, we will discuss the basic roles and concepts of the three DB (database) classes you will create. The design and code of the classes -- their ivars and methods -- will be up to you.

A `DBBinding` encapsulates a string `key` and string `value`. The text format of a `DBBinding` is the key, followed by a colon (:), followed by the value, possibly with extra whitespace around the key and the value. So for example the text "`name:Midnight Run`" represents the binding (key="name", value="Midnight Run"). The text "` year : 1979 `" represents the binding (key="year", value="1979"). The text in a `DBBinding` does not change after it has been created.

A `DBrecord` is simply a collection of `DBBinding`s. The text format of a `DBRecord` is simply the text of its bindings separated by commas, like this…

```
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant
```

Finally, a `DBTable` is just a collection of `DBRecords`. The text format of a `DBTable` is simply the text of its `DBRecords`, one per line. Here is our sample "`movies.txt`" table…

```
name:Alien, stars: Yaphet Kotto, stars:Sigourney Weaver, stars: Harry Dean Stanton
name:Repo Man, stars: Emilio Estevez, stars:Harry Dean Stanton
name:The Truth About Cats and Dogs, stars: Janeane Garofalo, stars:Uma Thurman
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant
name:Midnight Run, stars: Yaphet Kotto, stars: Charles Grodin, stars: Robert Deniro
```

Each `DBRecord` is on a single line of text, although the wrapping of the text in a display may make it look like some records span multiple lines. The key and value strings are guaranteed to not be the empty string, and to not contain colons, commas, or newlines, so our text format will work. You code may assume that the inputs are correctly formatted. The bindings in a record and the records in a table should print out in the same order they had when originally read in. Also, the data should print out retaining the capitalization it had when read in.

Notice that it is valid for a `DBRecord` to contain multiple bindings with the same key — multiple "`stars`" bindings for example. We're doing something a little more flexible than a conventional SQL database table.

Select

Each record also encapsulates a single "selected" `boolean` that is initially `false`. The main function of all the database classes is running "select" tests against all the records in a table, setting the selected booleans within certain records. When printed, a record that is selected prints with a "`*`" at the start of its line (shown below). When reading in a `DBRecord` from its text form, skip over the leading "`*`" if present (i.e. new records always have a `false` selected `boolean`).

Conceptually, a "select" operation tests every table record vs. a single fixed "criteria" record. For each record in the table, if the select test is `true`, then the record's selected `boolean` is set to `true`. If the test is `false`, the record's selected `boolean` is left unchanged.

Select operates in two modes — OR mode and AND mode.

In AND mode, the select test is `true` for a given record if all of the bindings in the criteria record are present in the given record. So for example, a select AND on the movies.txt table with the following criteria record...

```
stars: Yaphet Kotto, stars: Harry Dean Stanton
```

will select only the *Alien* record in the movies table. It will not select the *Repo Man* or *Midnight Run* records since they contain one or the other of the criteria bindings, but not both. The use of "for all" here mirrors the definitions of mathematical sets (CS103), so a select with an empty criteria record is always `true`. When defining boundary behaviors in your API, falling back on mathematical definitions is a good default strategy.

In OR mode, there must exist at least one binding from the criteria that is present in a record to select it. So with the same criteria as above but in OR mode, all three movies will be selected: *Alien*, *Repo Man*, and *Midnight Run*. So a select with an empty criteria record is always `false`.

As a convenience for the client, the criteria `value` strings do not need to match exactly: the criteria text can just be a substring within the table record value text, and they do not need to match upper/lower case. The `key` string must match exactly however. So the above criteria could be shortened to...

```
stars: kotto, stars: anton
```

and give the same result, as would the criteria record

```
stars   : aNton  , stars   : kOTTO
```

Command Line Interface

To drive the DB classes, we provide a simple console input/output class that presents a command-line interface to the user, parses what the user types, and then calls your DB classes to do all the actual work. This interface reminds us of why GUIs were invented, but it's good enough for this part of the homework. The command line has a single `DBTable` object, and each typed command triggers operations on that table…

r	reads (prompts for a filename and then adds the records in that file to the table)
p	prints (prints all the records, selected records start with *)
sa	selects in AND mode (prompts for criteria record)
so	selects in OR mode (prompts for criteria record)
da	deletes all records
ds	deletes selected records
du	deletes unselected records
c	clears all the selected booleans, so no records are selected

Below is a transcript of the command line interface in action to give you a sense of how the DB works. The command line prompt is "db:", and the typed commands are shown in bold. Look at the *CommandLineInterface.java* source file to see what's going on under the hood.

```
0 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
```

```
db:r
read
Filename:movies.txt

5 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:p
print
name:Alien, stars:Yaphet Kotto, stars:Sigourney Weaver, stars:Harry Dean Stanton
name:Repo Man, stars:Emilio Estevez, stars:Harry Dean Stanton
name:The Truth About Cats and Dogs, stars:Janeane Garofalo, stars:Uma Thurman
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant
name:Midnight Run, stars:Yaphet Kotto, stars:Charles Grodin, stars:Robert Deniro


5 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:sa
select and
Criteria record:stars:kotto, stars:stanton

5 records (1 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:p
print
*name:Alien, stars:Yaphet Kotto, stars:Sigourney Weaver, stars:Harry Dean Stanton
name:Repo Man, stars:Emilio Estevez, stars:Harry Dean Stanton
name:The Truth About Cats and Dogs, stars:Janeane Garofalo, stars:Uma Thurman
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant
name:Midnight Run, stars:Yaphet Kotto, stars:Charles Grodin, stars:Robert Deniro


5 records (1 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:c
clear selection

5 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:so
select or
Criteria record:stars:kotto, stars:stanton

5 records (3 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:p
print
*name:Alien, stars:Yaphet Kotto, stars:Sigourney Weaver, stars:Harry Dean Stanton
*name:Repo Man, stars:Emilio Estevez, stars:Harry Dean Stanton
name:The Truth About Cats and Dogs, stars:Janeane Garofalo, stars:Uma Thurman
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant
*name:Midnight Run, stars:Yaphet Kotto, stars:Charles Grodin, stars:Robert Deniro


5 records (3 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:ds
delete selected

2 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:p
print
name:The Truth About Cats and Dogs, stars:Janeane Garofalo, stars:Uma Thurman
name:Sense and Sensibility, stars:Emma Thompson, stars:Hugh Grant


2 records (0 selected)
r read, p print, sa select and, so select or, da ds du delete, c clear sel
db:q
quit
```

Your Mission

- Understand conceptually what the DB classes store and what the select operation computes.

- Design and implement `DBBinding`, `DBRecord`, and `DBTable` classes -- ivars, constructors, public methods, and private helper methods. We will allow a lot of latitude about your specific design, so long as it gets the job done and has a reasonable OOP design. Your design should follow the ideas of OOP encapsulation, receiver-relative coding, and good API design.

- The `DBRecord` builds on (is a client of) the `DBBinding`, and the `DBTable` is a client of both. The command line class is the ultimate client of the whole DB system. Therefore, collectively, the DB classes should expose constructors and methods so the command line interface can get its job done most easily. The DB classes do not need to support any more functionality than that required by the command line client and unit tests.

- Modify the provided command line class code so that it calls your DB code to do the actual work. The command line starter code just has the command line printing and parsing done, with empty slots for the actual DB manipulation. The DB code in the command line should be very short, since with good API design, the actual DB functionality should be in the DB classes.

- Write `JUnit` test classes for `DBBinding`, `DBRecord`, and `DBTable`. The tests are not required to be very long -- say just three or four test methods per class -- but of course it may be in your interest to make them longer. The DB classes should also think of the unit tests as a client, and may include features to support the unit tests. In particular, it's handy to be able to create `DBRecord`s from string constants. Therefore, the `DBRecord` should have a constructor that takes a single `String` line like "`name:Alien, stars:Yaphet Kotto, stars:Harry Dean Stanton`" and constructs that `DBRecord`.

- As part of your class design, write "*javadoc*" comments for your `DBRecord` class (as shown in lecture). Your comments do not need to be very lengthy, but should capture the basic interface of each method. Consider starting each comment with an "s" form verb, like "adds", "removes", "finds". Use the `@param` and `@return` tags -- notice that Eclipse autocomplete then uses that information. Use the default *Generate Javadoc* command in Eclipse to turn in your project with a "doc" directory of Javadoc. We are only doing Javadoc for the `DBRecord` class. The command may ask you the location of the "javadoc" tool which does the doc generation. For Windows users, it's in your Java folder. If you can't find it there, you might need to install the JDK. For Mac OS X, it's in /System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Commands/javadoc

DB Implementation Ideas

Implement `DBRecord` and `DBTable` using the `Collection` class `ArrayList` for storage. Store the pointer to your `ArrayList` objects as type `Collection<DBBinding>` and `Collection<DBRecord>`, so a different implementation may be substituted later. Use the basic `Collection` methods `add()`, `size()`, `foreach`, `iterator()`, and `iterator.remove()`. Use `iterator.remove()` to remove elements from the collection during iteration (somewhat confusingly, the collection itself also responds to a `remove()`, but that version searches the whole collection, so it's an order of magnitude slower). To support text output in the DB classes, override `toString()`. It's natural to put in debugging print statements while building the code, but please comment these out for your final turn-in. The output of the command line interface should be exactly as shown here, so we can run its output into automated grading scripts (we look at your source code too, but it's nice to be able to check your output for correctness easily).

The "select" algorithm is a bit tricky. Use decomposition and receiver-relative coding to try to break the problem up a bit, and hit your select code with a few unit tests. The select code should be smart enough to

stop computing when it has figured out the answer for a particular record and criteria. Use unit-tests to push on the various boundary cases. Most of the complexity of the DB is in the `DBRecord` while `DBTable` is pretty simple.

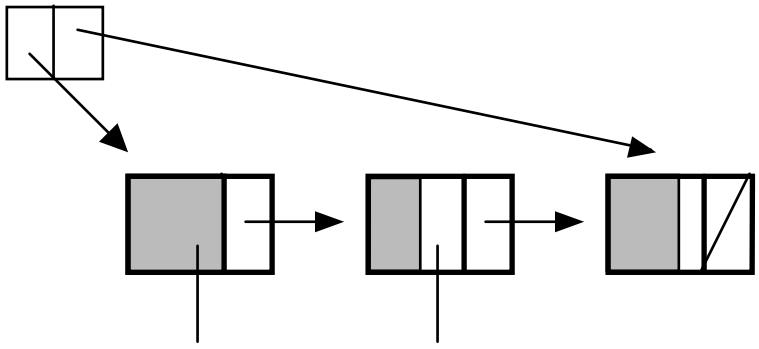
It could make sense to implement the record using a `HashMap` to store each binding by its `key`, however we are not doing that in this version.

Part 2 -- ChunkList

For Part 2, we'll add a `ChunkList` to take care of the storage. The themes are: isolating complexity inside an object, and testing.

ChunkList

The `ArrayList<E>` is nice and all, but what you really want is a nice `ChunkList<E>` class that implements the `Collection` interface. A `ChunkList` is like a regular linked list, except each node contains a little fixed size array of elements instead of just a single element. Each node also contains its own "`size`" int to know how full it is. For CS108, the `ChunkList` is a great example of both OOP encapsulation and unit testing.



Full chunk Partially full chunk

The `ChunkList` will have the following features...

- The `ChunkList` object contains a `head` pointer to the first chunk, a `tail` pointer to the last chunk, and an `int` to track the logical size of the whole collection. When the size of the list is 0, the `head` and `tail` pointers are `null`.

- Each chunk contains a fixed size `T[]` array, an `int` to track how full the chunk is, and a `next` pointer. The constant `ARRAY_SIZE = 8;` in the chunk class defines the fixed size of the array. Elements should be added to the array starting at its 0 index. The elements in each little array should be kept in a contiguous block starting at 0 (this will require shifting elements around in the little array at times). (You may want to do some testing with `ARRAY_SIZE` set to smaller values, but turn it in set to 8.)

- `ChunkList` should be a subclass of `AbstractCollection` which provides some basic facilities built on top of your `add()`, `iterator()`, `size()` etc. primitives. `Chunk` should be an inner class defined inside of `ChunkList`. Only `ChunkList` will see or use the `Chunk` class directly. It's stylistically ok if `ChunkList` accesses the state (.next, .data, ...) of the `Chunk` objects, since `Chunk` essentially is just an integrated part of `ChunkList`. Add utility methods to `Chunk` where it helps to keep things receiver-relative (`remove` for example). `ChunkIterator` should be a private inner class as in the lecture example.

- `ChunkList` must support the messages: constructor, `add()`, `size()`, and `iterator()`. The `iterator` must support `hasNext()`, `next()`, and `remove()`. It's valid for the client to add `null` as an element, so you cannot use `null` as some sort of special marker in the array -- use your size `int` in each chunk. Make sure that `size()` and `hasNext()` are consistent about the size of the collection.

- The empty collection should be implemented as `null` head and tail pointers. Only allocate chunks when actually needed.

- The `add()` operation should add new elements at the end of the overall collection -- i.e. new elements should always go into the `tail` chunk. If the `tail` chunk is full, a new chunk should be added and become the new `tail`. We are not going to trouble ourselves shifting things around to use empty space in chunks in the middle of the list. We'll only look at the `tail` chunk.

- Do not use a dummy node because (a) it does not help the code much, and (b) dummy nodes are lame.

- Keep a single `size` variable for the whole list that stores the total number of client data elements stored in the list, so you can respond to `size()` in constant time. Similarly, keep a separate `size` in each chunk to know how full it is. Likewise, `add()`, `hasNext()`, `next()`, and `remove()` should all run in **O(1)** constant time (they may do computations proportional to `ARRAY_SIZE`, but not proportional to the overall collection length).

- When using `iterator.remove()` to remove an element from a `Chunk`, overwrite the pointer to that element with a `null` to help the garbage collector. This is not a requirement, but it's a nice touch.

- When an `iterator.remove()` operation causes a chunk to contain zero elements, that chunk should be removed from the list immediately. The code for deletion is quite tricky. You may store a `previous` pointer in each chunk if you wish. It is possible to code it without previous pointers by keeping extra pointers in the `iterator` to remember the two previously seen chunks during iteration. `Chunk` deletion will need to work for many boundary cases -- the first chunk, the last chunk, and so on. This is probably the trickiest part of the whole thing.

- When called with correct inputs, `ChunkList` should return the correct result. However, when called with incorrect inputs -- e.g. `iterator.remove()` without first calling `next()` -- `ChunkList` does not need to do anything in particular. It's fine if your code just gives the wrong output or throws a null pointer or other exception. This is a slight relaxation of the formal `Collection` interface which guarantees to throw particular exceptions for particular errors.

ChunkList Advice — Tight Code

The `ChunkIterator` is a tricky bit of code. There are about 4 different cases to get right, depending on if the removed chunk was first or last in the list. Don't have a separate copy of the remove code for each special case. If you find yourself copying and pasting code, you're doing it wrong. Clean up the solution so there is one copy of the `remove` code and then a few lines to deal with the special cases. This is just general coding style advice – avoid proliferating copies of the code for slightly different cases. Try to factor the code down so one copy of the code deals with many cases. My `remove()` method is about 15 lines long.

Testing

The `ChunkList` is extremely well suited to unit-testing. This is good, since the `ChunkList` has such a large number of tricky little boundary cases where it can go awry. We will unit-test the `ChunkList` aggressively, since it's the best way to get the code right.

1. ChunkList Basic Testing

To get started, write some basic unit tests that build up a `ChunkList` of `Strings` with `add()`, and then iterate over it and remove a few elements, checking that the list look right before and after the removes. These tests will get the most basic `add()`/`iterator()`/`remove()` code working.

2. ChunkList Super Test

I'm providing the code for an extremely aggressive test on the `ChunkList`. Only try this after your basic tests are working. Because the `ArrayList` (known to be correct) and the `ChunkList` both implement the `Collection` interface, we can use a unit-test strategy where we do the same operation on both an `ArrayList` and `ChunkList`, and then verify that they get the same output after each operation.

The `SuperTest` creates both an `ArrayList` and a `ChunkList`. We'll call an "operation" either a single addition or an iteration down the collection doing a few random removes. Do the same random operation to both the `ArrayList` and `ChunkList`, and then check that the two look the same, element by element. Then loop around and do that 4999 more times, checking the two again after each random operation. We want to push on the `ChunkList` to get every weird combination of list length and this or that chunk being full or empty at the time of add or series of removes in some position. Take a look at the `SuperTest` code. It creates a `Random(1)` for its random number generation, object passing 1 as the seed. In this way, the series of "random" operations is the same every time the test is run.

3. ChunkSpeed Test

The unit-tests should work on the correctness of the `ChunkList`, and that's the most important thing. Look at the source of the provided `ChunkSpeed` class and try running it. It runs a timing test on the `ArrayList`, `LinkedList`, and `ChunkList` classes. The test is a crude simulation of collection add/remove/iterate use. Chun©kSpeed prints the milliseconds required to do the following representative mix of operations...

- Use `add()` to build a 50,000 element collection

- Create an iterator to iterate halfway through the collection

- Use the iterator to remove 10,000 elements at that point

- Creates and runs iterators to run over the whole collection 5 times

The speeds seen depend on many factors, including the `ARRAY_SIZE`, the specific hardware, the system load, JVM version, etc., and some runs will be way off because the GC thread runs during part of the test. Broadly speaking, on average the `ChunkList` should be roughly as fast or faster than the `LinkedList`, and a lot faster (a factor of 10 or so) than the `ArrayList`. The speed numbers have a lot of noise in them, so don't worry about a single run. The `ChunkList` might be slower than the `LinkedList` on occasion. However, if your `ChunkList` is consistently slower than the `LinkedList` or `ArrayList`, there is probably something wrong with your `ChunkList` algorithm -- some operation that is supposed to be **O(1)** is **O(n)**. The most important methods to be fast are `hasNext()` and `next()`, since they are the most commonly called by the client.

Onward to the DB

When your `ChunkList` is working well, change both your `DBRecord` and `DBTable` to use `ChunkList` instead of `ArrayList`. (This requires that `DBTable` and `DBRecord` only use the standard collection methods: `add()`,. `iterator()`, etc. which is as it should be.) This should be a one-line change like this…

```
// records = new ArrayList<DBRecord>();      // old
records = new ChunkList<DBRecord>();         // new
```

With the `ChunkList` installed, your DB should still work perfectly. If it does not, you may want to look at your `ChunkList` unit testing again.

Puzzle

As the final deliverable for Part A, use your working DB classes and the command line to solve the following little puzzle. Find the correct person from within the *people.txt* file, delete all the other records, and print the one solution record. Make a transcript of that interaction, with the print of the solution record as the last step before quitting. Include the transcript in your submit directory as the file "`puzzle-soln.txt`" -- just copy and paste from the console output window, or use the "`script`" command on Unix using the "`exit`" command to end the script.

This is just a little puzzle that acts as a final test for both your DB classes and your `ChunkList`: Someone has taken the last of the donuts from the instructor lounge! As the result of some pretty complicated and difficult to explain detective work, you know the following about the guilty party. Consider the sets of attributes A, B, and C below. The guilty party has none of the attributes listed in group A. The guilty party has at least one attribute in B and at least one attribute in C. Once you have a candidate guilty party, it's fairly easy to verify that your candidate has the right relationship with A, B, and C by hand, just to double-check the correctness of your DB and `ChunkList`. Identify that guilty person! The text below in the text file "`puzzle.txt`" so you can copy/paste the text instead of typing it. Please include a `puzzle-soln.txt` transcript with your submission.

```
// Nothing from A, something from B and C
// A
likes: dennis-ritchie, dislikes: imelda-marcos, hair: gray, is: assertive, dislikes: winnie-the-pooh,
dislikes: blue, is: quixotic, likes: the-michelin-woman

// B
hair: blonde, hobby: tetris-playing, is: sweet, dislikes: nice, hobby: puddle-jumping, likes: malcom-x,
dislikes: sneezy, is: sensitive, likes: krusty-the-clown

// C
is: quixotic, hobby: simpsons-watching, dislikes: assertive, is: appealing, hobby: movie-going, dislikes:
tax-deductible, likes: lorena-bobbit, hobby: snipe-hunting, likes: winnie-the-pooh
```

Part B - Sudoku

For this part of the project, you will build code to solve Sudoku puzzles. Our approach will concentrate on OOP and API design, and give us a chance to start doing some GUI coding. You do not need to be good at Sudoku to build this code. I'm quite slow at them. In fact, this whole project is perhaps cheap revenge against the Sudoku puzzles I've struggled with.

Sudoku is a puzzle where you fill numbers into a grid. The history is that it originated in the Dell puzzle magazine in the 1970's, and later became very popular in Japan, possibly filling the niche that crossword puzzles play in English newspapers, as the Japanese language is not suited to crossword puzzles. Sometime around 2005 it because a worldwide sensation. (See the Wikipedia page for the full story.)

The Sudoku rules are: fill the empty squares in the 9x9 grid so that each square contains a number in the range 1..9. In each row and each column across the grid, the numbers 1..9 much appear just once ("Sudoku" translating roughly as "single"). Likewise, each of the nine 3x3 squares that make up the grid must contain the just numbers 1..9.

Here is an easy Sudoku puzzle. I can solve this one in about 5 minutes. Look at the topmost row. It is only missing 1 and 7. Looking down the columns, you can figure out where the 1 and 7 go in that row. Proceed in that way, looking at rows, columns, and 3x3 squares that are mostly filled in, gradually figuring out the empty squares. A common technique is to write the numbers that might go in a square in small letters at the top of the square, and write in a big number when it's really figured out. Solve this puzzle to get a feel for how the game works (the solution is shown on the next page).

	3	5	2	9		8	6	4
	8	2	4	1		7		3
7	6	4	3	8			9	
2	1	8	7	3	9		4	
			8		4	2	3	
	4	3		5	2	9	7	
4		6	5	7	1			9
3	5	9		2	8	4	1	7
8			9			5	2	6

1	3	5	2	9	7	8	6	4
9	8	2	4	1	6	7	5	3
7	6	4	3	8	5	1	9	2
2	1	8	7	3	9	6	4	5
5	9	7	8	6	4	2	3	1
6	4	3	1	5	2	9	7	8
4	2	6	5	7	1	3	8	9
3	5	9	6	2	8	4	1	7
8	7	1	9	4	3	5	2	6

Sudoku Strategy

There are **many** ways to solve Sudoku. We will use the following approach which is a sort of OOP interpretation of classic recursive backtracking search. Call each square in the puzzle a "spot". We want to do a recursive backtracking search for a solution, assigning numbers to spots to find a combination that works. (If you are rusty with recursion, see the practice recursion problems at javabat.com).

- When assigning a number to a spot, never assign a number that, at that moment, conflicts with the spot's row, column, or square. We are up-front careful about assigning legal numbers to a spot, rather than assigning any number 1..9 and finding the problem later in the recursion. Assume that the initial grid is all legal, and make only legal spot assignments thereafter.

- There are 81 spots in the game. You could try making assignments to the blank spots in any order. However, for our solution, first sort the spots into order by the size of their set of assignable numbers, with the smallest set (most constrained) spots first. Follow that order in the recursive search, assigning the most constrained spots first. Do not re-sort the spots during the search. It works well enough to just sort once at the start and then keep that ordering. The sorting is just a heuristic, but it's easy and effective.

- We will set a max number of solutions of 100 -- if the recursive search gets to a point where it has 100 or more solutions, it can stop looking and just return how many have been found so far.

Sudoku OOP Design

For this project, the starter code has some routine code and data, and the rest of the design is up to you. Your goal is to design classes and APIs so that the `solve()` method (below) is clean expression of the strategy described above, and the `main()` and GUI clients are clean.

We will take an OOP approach to the search by treating each spot as its own capable little object. Create a `Sudoku` class that encapsulates a Sudoku game and give it a `Spot` inner class that represents a single spot in the game. Constant factor efficiency is not a big concern -- we're going for correctness, clarity, and a reasonably smart strategy.

Concentrate on OOP design around the `Spot` class -- push complexity into the `Spot`, making things easy for clients of the `Spot`. For example, the `Spot` has its own access to the `grid` (remember, it's an inner class of Sudoku). Consider these two examples of client code:

```
// Bad
grid[spot.getRow()][spot.getCol()] = 6;

// Good
spot.set(6);
```

Your code may be designed however you like, within the following requirements...

- `Sudoku(int[][] grid)` -- constructor takes the initial `grid` state, and we assume that the client passes us a legal grid. Empty spots are represented by 0 in the grid. You may assume that the grid is 9x9. You do not need to use `int[][]` as the internal representation; it's just an input/output format.

- `Sudoku(String text)` -- takes in puzzle in text form -- 81 numbers. (Starter file provides some parsing code.)

- `String toString()` -- override `toString()` to return a `String` made of 9 lines that shows the rows of the grid, with each number preceded by a space (use the `StringBuilder` class which replaces the old `StringBuffer`). (Essentially, the reverse of the text constructor.) For example, here is the `toString()` of the "medium 5 3" puzzle...

```
 5 3 0 0 7 0 0 0 0
 6 0 0 1 9 5 0 0 0
 0 9 8 0 0 0 0 6 0
 8 0 0 0 6 0 0 0 3
 4 0 0 8 0 3 0 0 1
 7 0 0 0 2 0 0 0 6
 0 6 0 0 0 0 2 8 0
 0 0 0 4 1 9 0 0 5
 0 0 0 0 8 0 0 7 9
```

- `int solve()` -- tries to solve the puzzle using the strategy described above. Returns the number of solutions and sets the state for `getSolutionText()` and `getElapsed()` (below). The original grid of the sudoku should not be changed by the solution (i.e. `toString()` is still the original problem). The included puzzles have 1 solution each.

- `String getSolutionText()` -- after a solve, if there was one or more solutions, this is the text form of the first one found (otherwise the empty string). Which solution is found first may vary, depending on quirks of your `solve()` implementation.

- `long getElapsed()` -- after a solve, returns the elapsed time spent in the solve measured in milliseconds. See `System.currentTimeMillis()`. In particular, it's interesting to get visibility into the timing effects of some of your code changes.

- You do not need to write unit tests, although you may want to anyway. (You may make `Spot` public for the purpose of writing unit tests for it.)

- You do not need to write a javadoc.

It's fine to use `Integer` or `int` or whatever to track the grid state -- whatever you find most convenient. It's good to leverage `Set<Integer>/HashSet<Integer>` and their built in methods `contains()`, `addAll()`, `removeAll()` to help solve the problem.

Do not pre-compute and store the possible numbers for a spot. Pre-computation worked very well in tetris, but it does not work well here. Each spot assignment changes the possible numbers for 20 other spots. However, in the search, you only care about the possible numbers for the one spot you look at next. Therefore, doing the computation for all 20 spots ahead of time is a bad strategy -- better to compute the possible numbers for a spot at the moment you need them, based on the grid state at that moment.

Deliverable main()

Your Sudoku `main()` should be as below, using your code to print the problem and solution for the "hard 3 7" puzzle. As usual, comment out your other extraneous printing before turning in, so we can run your code to see just your clean output.

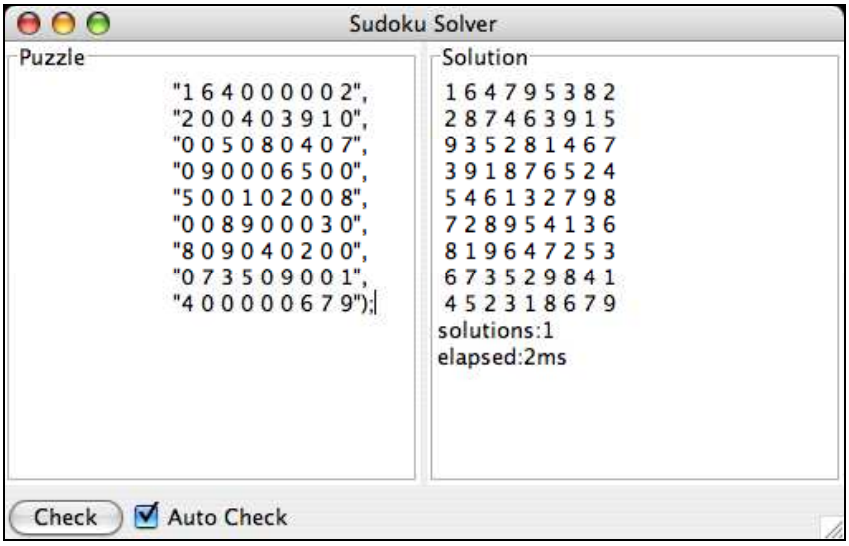
```
public static void main(String[] args) {
    Sudoku sudoku;
    sudoku = new Sudoku(hardGrid);
    System.out.println(sudoku); // print the raw problem

    int count = sudoku.solve();
    System.out.println("solutions:" + count);
    System.out.println("elapsed:" + sudoku.getElapsed() + "ms");
    System.out.println(sudoku.getSolutionText());
}
```

Deliverable GUI

Finally, with the core logic of the Sudoku done, it's time to nest it inside a `SudokuFrame` to make your hard algorithmic work available to a grateful public. The idea is that someone building a Sudoku puzzle could use this to play around with a puzzle they are building. The included puzzles all have a single solution. However, as you start adding 0's, they get more solutions. For example, changing the 7 of the hard 3 7 puzzle should yield 6 solutions. (This is easiest to play with when you have the GUI working.)

We'll make a simple layout like this: Use a `BorderLayout(4,4)` -- the "4" is a little spacer between the areas. Create 15x20 `JTextArea` in the center to hold the "source" puzzle text. Create a second 15x20 `JTextArea` in the east to hold the results. Create a horizontal box in the south to hold the controls. The code -- `textarea.setBorder(new TitleBorder("title"))` -- puts the little titled border around any component.



When the *Check* button is clicked, construct a Sudoku for the text in the left text area and try to solve it:

- If the text is mal-formed in any way so the construction of the Sudoku throws an exception, just write "*Parsing problem*" in the results text area. (Use a `try/catch`.)
- Otherwise, after the `solve()`, if there is at least one solution, write its text into the results text area.

- If there is a solution, write the "solutions:xxx\n" "elapsed:xxx\n" at the end of the results text area.

- Finally, the "*Auto*" checkbox should make it so that every keystroke add/delete in the text area automatically does a "Check". To implement this, get the "document" object from the text area. The document supports a `DocumentListener` object which gets notifications whenever the text changes. The Auto feature should work only if the auto checkbox is checked, which it should be by default.