Capstone Courseware, LLC
1 Washburn Place
Brookline, MA 02446

877-227-2477
capstonecourseware.com

Java Training Coding Assessments

Here are instructions for three coding exercises. Do as much on as many of these as you can in the time available, taking them in order and completing each one before moving ahead. It's absolutely not critical that you complete or even touch all three of these, and much more important to be thorough in completing what you can.

Starter code and some other resources, as mentioned in the instructions for each exercise, can be found in folders that sit alongside this PDF.

When done with an exercise, send your source files (and only your source files, please! not project or workspace files as managed by your IDE) to the instructor for review.

This is an individual assessment. You're free to use any and all internet resources, but you should not collaborate or discuss your work with others in the classroom. On the other hand, don't hesitate to ask about the requirements of a given exercise, or about logistical and up-and-running sorts of issues; and if you find yourself stuck on a given problem for more than maybe 15 minutes, check with the instructor. It's almost always better if the instructor can help you push past a specific problem so you can get moving again, as opposed to losing a lot of time on one niggling thing.

Specific exercise instructions and details are found on the following pages.

Exercise 1 – Certificate of Deposit

Here you will complete an application that can calculate future values of certificates of deposit at a bank. A CD has a face value, an interest rate, and a term: so you might invest $10,000 at 5% for 1 year and get $10,500 back. Interest is assumed to compound annually, too, so that same CD with a 3-year term would be worth $11,576.25 at maturity. Further, it's possible for the compounding period to be shorter than a year: it might be semi-annual, quarterly, or monthly.

In your IDE of choice, create a Java project called **CD**, and in that project create a package called **finance**. Copy the file **CD/src/finance/TestProgram.java** into that folder/package. This is just a starter with specific data for four test cases; but it doesn't do anything with that starting data. Your job is to implement a **CertificateOfDeposit** class that can be instantiated with specific values for face value, rate, term, and optionally compounding period (with a default of annual compounding), and that can report the value of the CD when it matures.

It's a good idea to get everything but the compounding period working, and then refine your class with that feature. The first two cases in the test program support this. Then deal with non-annual compounding and the third and fourth test cases.

Expected output from **TestProgram** is something like this:

```
$ 1,000.00 at  0.050% for 10 years -> $ 1,628.89
$   800.00 at  0.045% for  8 years -> $ 1,137.68
$ 1,000.00 at  0.050% compounding  2 time(s) per year for 10 years -> $ 1,638.62
$   800.00 at  0.045% compounding 12 time(s) per year for  8 years -> $ 1,145.89
```

Exercise 2 – Files and Folders

In this exercise you will implement support for a representation of a file system, with various sorts of nodes. Create a second Java project called **Files** and in it create a package **files**. Copy the starter sources in **Files/src/files** into your package. **TestProgram.java** builds up a tree of folders and files, using helper functions that so far don't do anything. It uses a prepared **Node** class, but this is incomplete, and the program compiles, but fails at runtime.

You will support four types of files and folders, each with certain attributes and behaviors:

- The **file** has a name, size in bytes, and last-modified date. Like all nodes in the tree it can report its name and can produce an HTML representation of itself via **asHTML()**. It can report its size and last-modified date as well.
- The **hidden file** has the same properties but shows a different HTML representation: grey text instead of black, and with the tag "[hidden]" at the end, in italics.
- The **folder** has a name, but no size or mod date of its own. Instead it can hold any number of files or folders as children. It reports its size by summing up all of its children's' sizes, and its HTML representation includes indented representations of all of the children.
- The **remote folder** is like a folder but represents as having zero size, regardless of its contents; it also shows in all italics.

See the provided **sample.html** for an idea of what the output of the program should be. The HTML in this sample is "pretty-printed" for readability, and you don't need to worry about matching the formatting – just focus on producing the correct HTML content. You can check the output in the console, and/or open the **files.html** that your code produces, to see the rendered HTML.

... and, no: don't use **java.io.File**, or any built-in file-system APIs for this! Create your own code. The exercise is not about connecting to your actual, local file system, but rather about modeling an abstract file system given the information provided.

Exercise 3 – Treasure-Hunt Game

The starter project includes complete implementations of the logic and UI for a treasure-hunt game, and an application that creates a hard-coded game layout and plays it with a series of hard-coded moves by a team of two players. Here are the game rules:

- The game board represents an island on which there can be found a treasure chest. Like all islands in the real world, this one is a perfect rectangle! and locations on the island are addressed by two dimensional coordinates that we call row and column. Also found on this island are a wizard and some number of precious coins. There will never be more than one of these things at the same row-column location (which we sometimes call a "square"): each will have treasure, a wizard, a coin, or nothing.
- The object of the game is to claim the treasure. But finding it isn't enough: to claim it you must know a magic spell.
- Only the wizard can tell you the spell. But finding the wizard isn't enough: he will want to be paid some number of coins.
- You enter the island with a team of two players, each starting at opposite corners of the remarkably rectangular island: so, one starts at row=0, column=0, and one starts at row=H-1, column=W-1, where H is the height or number of rows, and W is the width or number of columns.
- On each turn of the game, each player can (a) search the square it occupies, (b) act on anything it finds – this includes picking up a coin, paying the wizard for a magic spell, or claiming the treasure – and (c) move one square up, down, right, or left.
- The player can "see" what's in a newly-occupied square, but can't act on it until the next turn.
- All players can move during one turn, in any order.
- Players can't pool their coins in order to pay the wizard: one player must accumulate enough coins to satisfy the wizard's greed (and there will always be enough coins on the island to assure that you don't get stuck with all players having too few coins).
- Players can share their knowledge of the magic spell that's needed to claim the treasure – so for example one player can pay the wizard and the other can immediately claim the treasure, if both players are located on the right squares.

Your job is to write the logic to play the game, and to play it reasonably well with a two-player team. Not all the rules above are enforced by the provided classes, so you must implement logic to follow the rules.

Start by creating a Java project called **Treasure**, with a package **treasure**, and copying in source files and other resources into it. Reviewing the **Island** class, which represents the game board and implements the state of what items are where, as well as the logic of taking coins, paying the wizard, and claiming the treasure.

The look at the **Game** class. This is a graphical user interface that shows the state of play on one **Island**. It is passive: it will start with a blank grid, and must be told to reveal the contents of one square at a time (appropriate to the searching nature of the game play), and can also show icons representing the players as they move around. There is also a message area at the bottom of the GUI that can be useful for status updates and other diagnostics as you develop.

The **Game** has a method **onEachTurnCall()** that takes a **Supplier<Boolean>**. So you can pass it a reference to a class that implements this interface, and its **get** method will be expected to (a) play one turn of the game and (b) return **true** to continue play, or **false** to end the game.

You can see this API in use in **ExampleGame**, where an **Island** is set up with a 3x4 grid and two players, a **Game** is created with that island as its model, and then **onEachTurnCall()** is given a reference to a game-playing function. That function just hard-codes a series of moves, paying no attention to the state of the board but just relying on the programmer's foreknowledge of what will be found where. This function wins the game in six turns, by directing one player to gather coins and pay the wizard, and the other to go to the treasure and claim it once the spell is known.

The **RandomGame** class initializes a random, 6x6 game board, and has spots in its **main()** method for you to iniitalize your own logic and to provide your game-playing function. A good solution to this exercise will win the random game every time, and a very good solution will do so in as few turns as reasonably possible.

How you implement the setup and playing logic is up to you – but remember to observe the rules and the spirit of the game, especially that you have to have players move around the island to find things and then make decisions based on what you find. A solution that, for example, just immediately loops over the full grid, finds everything, and then tells players where to go, is not playing by the rules! The example game logic is meant to show you how to use the **Island** and **Game** APIs – it is not really exemplary in its decision-making as there is none and it just carries out a script of moves that we know ahead of time will win. Your playing logic should take the information that players learn by traversing the island as it comes.

You'll notice a unit test for the **Island** class, in a separate **test** folder. Feel free to write unit tests for your classes, but it's not a requirement of this exercise to do so, and of course time is limited.