This assignment requires you to write a program to validate chess piece moves. There are constraints placed upon the type of checking you must do, in order to simplify your task. You are given a starting Eclipse project with the interfaces you must implement and a small set of tests that indicate the type of tests that we will run on your solution.

# The problem

The problem is a fairly simple one. As part of a chess game, you would need to ensure that when a player attempts a move that it is a valid one. This is represented in the *can Move()* method of the **MoveValidator** class. All of the tests will call this method, giving it a **ChessBoard** instance and two **Square** instances. The method returns *true* if the move can be made, *false* if the move violates the rules of the game, and throws a **CMVException** with an appropriate message that indicates the error (e.g., one of the squares has an invalid coordinate or there is no piece on the square where the pieces should be moved from).

If you are not familiar with the rules of chess, there are many tutorials and set of basic rules that are available via a Web search. A very succinct description may be found at <https://www.chess.com/learn-how-to-play-chess>. We do not use the special moves in Step 3 of this description (en passant, pawn promotion, and castling) for this assignment. We only use the basic information provided in Step 2. We also do not take into account a King moving into check. This is just the basic movement of the various pieces

# Starting code

There are a few classes and interfaces in the starting code in the Eclipse project. Please read the comments on the classes to understand what you may and may not change.

You should be able to download the zipped archive and import it into Eclipse directly. Once you have imported the file, rename the project by changing the name. The name in the starter is **ChessMoveValidator-gpollice**. Change **gpollice** to your WPI login name.

Here are some things that may help you understand the starting:

* You may not change **ChessBoard.java**. This class provides a working board that provides all of the functionality you should need from the board. The constructor takes a Map<Square, ChessPiece> that provides the configuration of the pieces on the board. You can see how this is used in the sample tests in the project.
* There are two factory classes. These are classes with static methods that produce instances of classes the way a factory produces things.
  + The first is the **SquareFactory** that has a method to produce squares when you specify the column and row. Examples of invoking this method are in the sample tests. We use the factory in order to allow you to provide a subclass of the **Square** if you decide to implement your solution with a subclass. The tests we run will only use the public methods in the **Square** class in the starting code.
  + The second factory is the **ChessPieceFactory** that produces instances of the class(es) that you create that implement the **ChessPiece** interface.

## Expectations

You will implement the solution using the best design you can think of. Your code should be as “great” as you can make it. It should be understandable, obvious, elegant, and the intention clear. Your code should be documented with Javadoc comments on every class and non-private method. You should use a design that is flexible and is easily modified when changes are introduced. This is a problem where the use of Java lambdas can be quite useful to help you develop a flexible solution.

You should test your code using JUnit. Put your tests into the **test** folder, just like the sample tests are. You can add to the starting test file or create your own. You should test for any exception conditions, like bad Square coordinates or other exceptional conditions and throw the **CMVException** in this case. You will see an example of this in the starting code. You do not need to use Test-Driven Development for this assignment, but I recommend that you try to use it. This is the only assignment this term where you will not be required to use TDD for your coding.

Running your tests should give you about 95% test coverage (except possibly for enumerations and exceptions) of your **sac** folder.

Note that the staring code is formatted according to my coding style. You should either follow that style (indentation, placement of braces, and so on) or format the code to your style. If you have the formatting preference you use in Eclipse, you can select all text in a source file, or any part of the file, and hit CTRL-SHIFT-F to achieve the consistency.