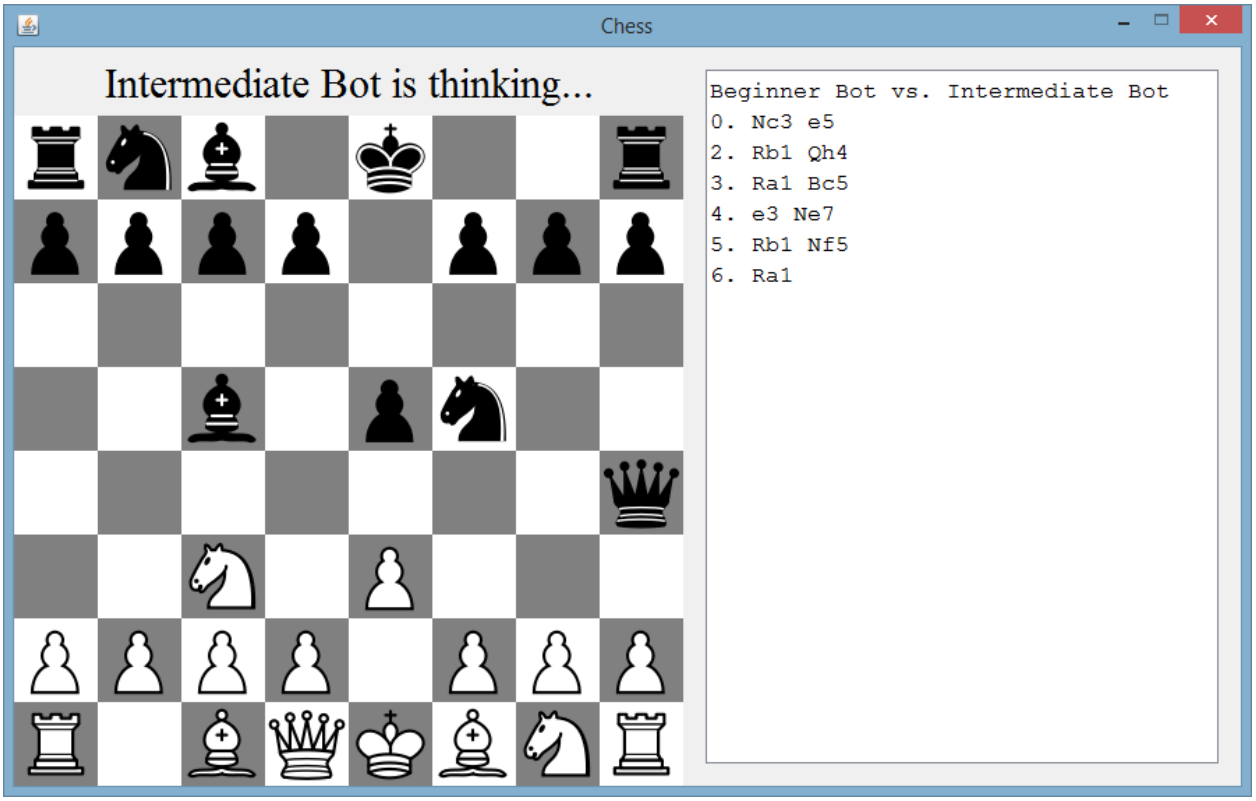
Project 1: Chess Bot

During this project, you will exercise what you have learned so far in AI about representation and search by creating a simple chess-playing program. This project provides a model of the state space of chess, and it is your job to write a program that explores that space efficiently and decides what moves to make to defeat other programs.

This assignment was originally designed by Dr. Stephen Ware, and we are fortunate to make use of the chess framework that he has graciously developed and provided.

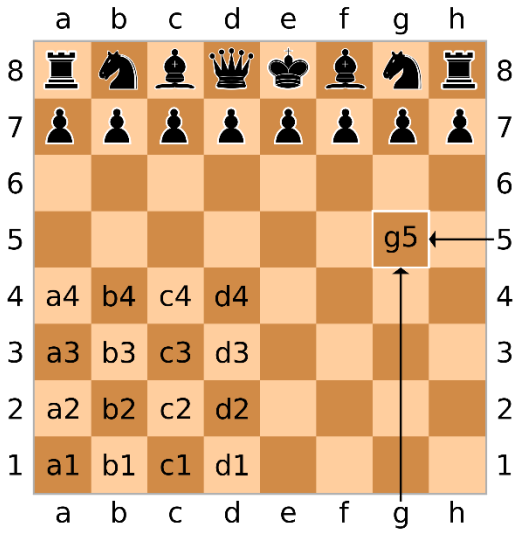


A screenshot of the chess GUI you'll be using to test your bot against a cadre of vicious rivals!

Jump to Section

Chess Background and Java Framework

Knowing the basics of chess, such as the names and movement rules for all of the pieces, as well as the definitions of terms such as stalemate and checkmate, will be helpful for you. Knowledge of more advanced terms, such as *rank* (the rows) and *file* (the columns) to describe chessboard positions, and the material score to evaluate how well a given player is doing is also valuable.



Typical chessboard notation. A row of squares is called a "Rank" and is numbered 1-8, starting on white's side. A column is called a "File" and is labelled a-h, starting on the far left. Spaces are specified "file first" e.g., g5 is "file g, rank 5"

The chess framework you will be using for this project (again, generously provided by Dr. Stephen Ware), is written in Java 9. At the time of this writing, Java 21 is the latest version of Java with long term support. To work on this project, you will need to make sure you have the Java Virtual Machine downloaded (anything after version 9 should be fine), install it on your computer, and ensure that Java is on your system path. How to do this differs between Windows and Mac users. I am more than happy to help you get this set up if you are having trouble (though if you've gone through the UNO curriculum, odds are high you already have what you need set up on your machine).

The starter code is also packaged as an Eclipse project. Although you are welcome to edit your code in any editor you wish, it is highly recommended you use Eclipse to do your programming and export your projects for this class; history has shown this to be the path of least resistance.

Once you have everything installed, download the following elements from the Canvas page:

- `doc.zip`
- `chess.jar`
- `bots.zip`
- `ChessBot.zip`

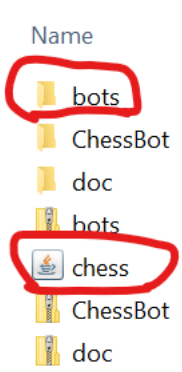
`doc.zip` has all of the documentation for Dr. Ware's chess framework. I understand that it might be overwhelming to look at all of this, but in both this assignment, future projects, and dare I say "real life" after graduation, your success will hinge on not being afraid to look through the documentation. To access it, simply unzip doc.zip, look inside the newly created "doc" folder, and then double click on "index.html" – it should open up the documentation in a browser (or ask you for your choice of browser to use). In case you are new to Java documentation, I'll include some helpful tips for reading through it at the end of this document.

`chess.jar` is the jar file that you will run that will simulate games of Chess.

`bots.zip` is a zip file that contains 6 other jar files: `beginner.jar, greedy.jar, human.jar, intermediate.jar, novice.jar,` and `random.jar.` Each of these represent a different chess-playing program, or "bot" (except for `human.jar`, which is a mechanism that lets YOU make the moves instead of a bot, if you want to play against any of these bots yourself).

`ChessBot.zip` is a zip file that contains a starter Eclipse project with the source code for "Random Bot." It is recommended that you use this as the starting point of your work for this project.

Once you have these files downloaded, extract `bots.zip` into the same directory as `chess.jar`. That is to say, you want a folder that might look something like this:



That is, the unzipped 'bots' folder and the chess.jar file are in the same directory (all of those other folders and files won't hurt anything but don't need to be there). You, of course, are welcome to put *any* of these things *anywhere* you want on your filesystem, but the examples commands that follow assume the above structure.

Once you do that, you have everything you need to have these bots run a **tournament** against each other! To see a sample tournament, open a terminal window, navigate to where you downloaded the files (i.e., the folder depicted above that has both the bots directory and also chess.jar), and execute the following command:

```
java -jar chess.jar 2 bots/random.jar bots/greedy.jar bots/novice.jar
```

This is the breakdown of the above command and its arguments:

- `java -jar chess.jar` ← This part is simply saying "run chess.jar"

- `2` ← This can be any positive integer. It is how many games each bot should play against each other, alternating white and black (i.e., who goes first).

- `bots/random.jar bots/greedy.jar bots/novice.jar` ← This is you specifying which bots you want participating in the tournament. Specifying the `bots/` is assuming the bot jar files are in a directory called `bots`, which itself is in the same directory as chess.jar (which should be the case if you followed the download instructions above).

With that, you should hopefully be able to parse the above command as saying "please run a tournament between Random Bot, Greedy Bot, and Novice Bot, with each bot playing two games against each other." This means that six total games will be played in this sample (two games Random vs. Greedy, two games Random vs. Novice, two games Greedy vs. Novice).

This will bring up a board as seen on the first page of this assignment. A transcript of each game is shown to the right of the board in Portable Game Notation. Though not strictly needed for the assignment, it is still recommend to take a few minutes brushing up on PGN. The basics involve specifying a piece by a letter (R = rook, N = Knight, B = Bishop, Q = Queen, K = King, and the absence of a letter means pawn), and then the file and rank of the square the piece is moving to. So for example, Nc3 would translate to "Knight to c3." Sometimes that is ambiguous (e.g., there are *two* knights that could reach c3 from their current positions). In those situations, they are disambiguated by the file they start on (e.g., Nge2 means "the knight currently in file g moves to e2).

There are typically two moves at a time. The first represents white's move, the second represents the response of black.

So, for example, using the transcript from the image on the first page:

Nc3	e5	White moved a knight to c3, black moved a pawn to e5 (both where they are in the image)
Rb1	Qh4	White moved their rook to b1, where the knight used to be. Black moved their queen to h4 (where you see it in the image)
Ra1	Bc5	White moved their rook back to a1. Black moved their bishop to c5 (where you see it in the image).
e3	Ne7	White moved a pawn to e3 (where you see it in the image). Black moved a knight to e7 (only one knight could have legally gone there, so it isn't ambiguous).
Rb1	Nf5	White moves his rook BACK to b1 again. Black moves a knight to f5. (where you see it in the image)
Ra1		White finally moves his rook BACK to a1 again, where you see it in the image. Black is still figuring out what to do for their turn.

There are additional symbols you might see sometimes in PGN (the extra files and ranks if pieces need disambiguation as described above, and the letter x is present if there is a capture (i.e., a piece is taken by another)), but the above should be enough to get you started.

The transcript ends with the final results of the tournament once all games have been played. If you run the command from above, Novice Bot should come in first with 3.5 games won, Greedy Bot second with 2 games won, and Random Bot last with 0.5 games won. (A game ending in a stalemate awards "0.5" of a win to both players).

Bots are ranked according to these criteria:

- The bot with more wins is ranked higher.
- In case of a tie, the bot that made fewer total moves is ranked higher.
- If there is still a tie, the bot that explored fewer total states is ranked higher.

As mentioned above, `human.jar` allows a human player to take the role of one of the players in the tournament. For example, to play 1 game between two human players:

```
java -jar chess.jar 1 bots/human.jar bots/human.jar
```

To play (as white) against a bot:

```
java -jar chess.jar 1 human.jar bots/novice.jar
```

Again, I cannot stress enough: the chess framework documentation (what you downloaded in doc.zip) will be helpful as you work on this assignment. See the end of the assignment for some tips on that.

If the bots are taking a long time to decide on moves, you can try increasing the amount of memory that the Java virtual machine uses. The `-Xms` command line argument specifies the minimum amount of memory to use, and the `-Xmx` command line argument specifies the maximum amount to use. You can also allocate more space to young generation objects using the `-XX:NewSize` command line argument to save on garbage collection time. To run the first tournament example with exactly 4 gigabytes of memory, 3 of which are used for new objects:

```
java -Xms4g -Xmx4g -XX:NewSize=3g -jar chess.jar 2 bots/random.jar
     bots/greedy.jar bots/novice.jar
```

Intermediate bot in particular likes to take his time figuring things out. I recommend not including him in tournaments until your bot is handily beating all the others. Then when you feel like your bot is ready, have one-on-one tournaments between your bot and Intermediate bot. If your bot can triumph over Intermediate, it should be able to easily crush the others as well.

Assignment

Your assignment is to write your own chess bot. Do this by creating a .jar file which contains (at the top level) any class which is a subclass of `com.stephengware.java.games.chess.bot.Bot`. It must override the `chooseMove()` method.

You can download the source code for Random Bot (the ChessBot.zip file from Canvas), which is packaged as an Eclipse project archive. To open the project:

- Make sure you have Eclipse downloaded and installed.
- Open Eclipse.
- `File > Import`
- Under the `General` category, choose `Existing Projects into Workspace` and click `Next.`
- Choose the `Select archive file` option.
- Browse for `ChessBot.zip`
- Click `Finish`

Once you do this, if you open up MyBot.java you will see what is essentially the source code for `RandomBot`: the `chooseMove()` method has a `State` object argument which represents the 'current state.' It iterates through all of the child states (i.e., every state reachable from the current state by making a single move) and adds each of those child state to an array list. It then simply returns one of those child states at random to represent the next move. It is recommended that you start with this project and modify it to choose moves more intelligently.

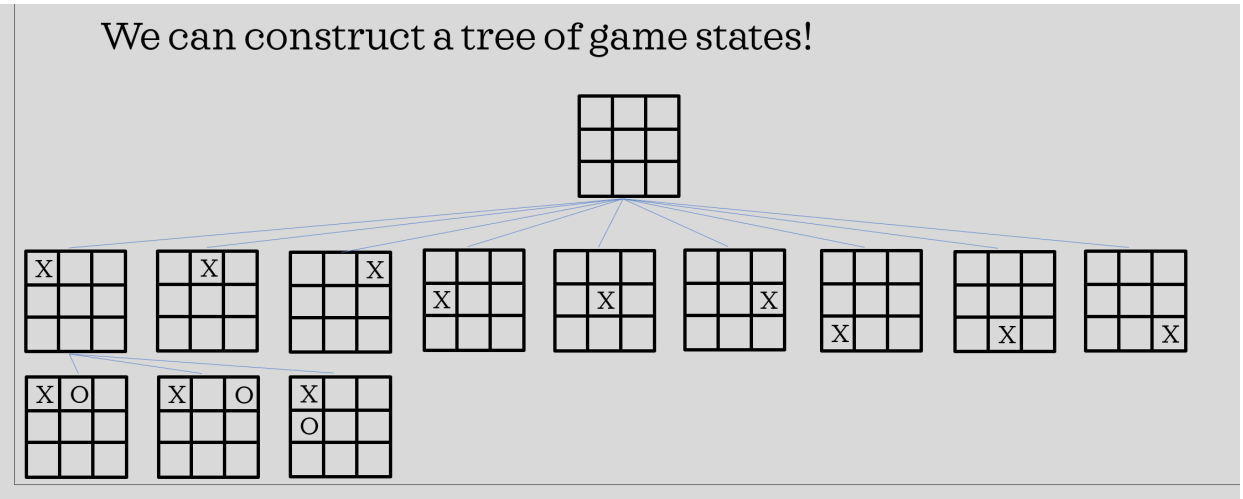
It is also recommended that you rename the Eclipse project to reflect your name:

- Right click on the project.
- Choose `Refactor > Rename`

You are also *required* to change the bot's name from "My Chess Bot" to your student ID (i.e., your university e-mail address). For example, because my university e-mail address is sbanerj1@cs.uno.edu, my ID would be sbanerj1.

Your bot should explore the state space and make intelligent decisions about what moves to make next. You should start in the `chooseMove` method. Here you are provided with a `State` whose root represents the current state of the board and must return one of its child states. Each child state represents a reachable state given the current board configuration. Thus returning one of these child states is essentially expressing which move you want to make.

To help you to visualize this, remember this slide from lecture (the image is for tic-tac-toe instead of chess, but the idea remains the same):



From any given state a 'move' takes you to a child state that represents the effect of taking that move. The chess framework that you are given allows you to access ALL of the child states of any given state (e.g., if given the empty board from the above image, it would produce the nine possible child states you see above – one state for each possible location that X can make a move in).

So again, the framework gives you access to *all* of the potential "next board states" from any given state. This is demonstrated in the Random Bot code you have access to. The heart of this assignment then becomes: *how do you decide *which* of those next board states is best?* The Random Bot base code you are given doesn't think very hard about how any given state might be better than another – but that's where you come in, to teach that bot a thing or two!

Spoiler Alert: Smart Chess bots might need to look ahead by a move or two to make their decision (i.e., if I do this, then he'll probably do this, in which case I'll do this…), that is to say, as you begin working on your bot in earnest, you'll not only looking at the child states of the current state, but the child states *of* those child states (and possibly going down even further). **On any given turn, the number of states you may explore is limited to 500,000.** Because of this, you should try to explore the space as efficiently as possible. If you exceed the search limit, your bot will throw an exception and you will get a very poor grade. The Random Bot sample code has lines of code which demonstrate how you can "abort" if you are in danger of reaching the space exploration limit. This will prevent your bot from throwing an exception and crashing (good!), but likely means your bot didn't get the chance to finish considering the merits of all of the possible moves it can make it the current state and thus is highly likely to make a sub-optimal move (not good!).

While you are working in Eclipse, you can see how your bot is doing by running `Test.java` (another file, with a main method, in the Eclipse project). Note that this *is*

not how your project will be graded. It is only a convenience to save you the trouble of exporting your bot as a JAR file every time you want to test it.

To save yourself time, when running Test.java it is recommended that you only run your bot against one other bot at a time. You can do this by commenting out the lines that add the bots you aren't concerned with to the `Bot[]` array in `Test.java main()` method. Make sure you are adding your bot, though! In general, if your bot can beat a bot, it can beat anything "below" it as well (e.g., if you can beat `NoviceBot`, then you likely will also beat `GreedyBot` and `RandomBot`, too).

There are two things which you must upload to Canvas for this assignment: a .jar file of your bot executable, and a .zip file of your source code.

When you have finished your bot, **you can export it as a .jar file** using Eclipse:

- Right-click on the project and choose `Export`
- Under the `Java` category, choose `JAR file` and click `Next`.
- Choose the destination for the exported file, making sure that it is named with your student ID and ends in `.jar`.
- Click `Finish`.

For example, since my ID is sbanerj1, my bot's name will be sbanerj1`.jar`

You must also **export and submit your source code a zip file**:

- Right click on the project and choose Export
- Under the `General` category, choose `Archive File` and click `Next`.
- Choose the destination for the exported file, making sure that it is named with your student ID and ends in `.zip`

You will need to submit both the bot `jar` file and the `zip` file of your source code on Canvas.

Grading

I am telling you exactly how I will grade your project – these are steps that *you* can do yourself, so in theory there should be no surprises about what your final grade on this assignment will be!

First: I will download your jar file and place it in the same directory as chess.jar. Then I will run the following command, using your id instead of mine:

```
java -Xms4g -Xmx4g -XX:NewSize=3g -jar chess.jar 2 sbanerj1.jar
bots/random.jar bots/greedy.jar bots/novice.jar bots/beginner.jar
bots/intermediate.jar
```

This means that your bot will play in a tournament against the five bots that I provided. Your bot will play two games against each opponent, one as white and one as black.

Your grade will be determined as follows (depending on whether you are enrolled in 4525 or 5525):

Your bot…	CSCI 4525 Grade	CSCI 5525 Grade
Beats Intermediate Bot	A+ (105%)	A (100%)
Beats Beginner Bot	A (100%)	B(80%)
Beats Novice Bot	B (80%)	C (70%)
Beats Greedy Bot	C (70%)	D (60%)
Beats Random Bot	D (60%)	F (50%)
Throws an Exception during play	F (50%)	F(0%)

The following requirements must be observed when submitting your project or it will not be graded:

- Submit your project on Canvas by the deadline.
- Submit your `jar` file, named for your student ID (e.g., sbanerj1`.jar`). If your bot does not load correctly or throws an exception during play, you will receive an F for this project. **Test on your own machine** before submitting!
- Submit the source code for your bot as an Eclipse archive via the method described above. It must be named for your student ID (e.g., bsamuel.zip). If I cannot import your source code, or if you export it incorrectly, I will not grade it. If you are at all concerned about this, come to my office hours or ask me beforehand!
- You must perform your search by expanding and searching the provided state class. Do not attempt to circumvent the 500,000 node search limit by, for example, creating your own State class which does not obey the limit. Circumventing the 500,000 node limit will be considered cheating.
- As outlined in the syllabus, there is a zero tolerance policy for cheating or plagiarism. If I am at all concerned that you submitted code which is not your own (found from the internet, a classmate, or through some reverse engineering of Java bytecode), I reserve the right to 'audit' you and have you explain your code to me. Of *course* what I want the result of that audit to be was that I was wrong and you knew your code inside and out. But if not then we'll need to fill out the academic dishonesty form, which I don't want and you don't want. So let's not come to that.
- Projects are meant to be done individually – you should write all of your code yourself. However, if you find yourself conferring with other students, that's OK – but then *absolutely* mention who you worked with in your submission.

About the Bots

Here is some information about your opponents which you may find helpful when designing your bot:

- **Random Bot** chooses moves at random, bless its heart.
- **Greedy Bot** always choses the move which maximizes its total material score. If it has multiple moves that result in the same score, it chooses at random between them. It does not "look ahead" at all; it (greedily) chooses only among reachable states from the current board configuration.
- **Novice Bot** uses Minimax search with alpha beta pruning to look 1 turn ahead (i.e., 2 ply or 1 move for each player) and choose the move which will maximize its material score.
- **Beginner Bot** uses iterative deepening Minimax search with alpha beta pruning to look 2 turns (i.e., 4 ply) ahead and choose the move which will maximize its material score.
- **Intermediate Bot** uses iterative deepening Minimax search with alpha beta pruning to look 3 turns ahead (when possible). It uses a more advanced utility function that considers positioning. Its search is quiescent. It also uses an opening book to make its first moves when possible.

One path that other students have found successful is to "start small" – i.e., focus first on beating only Random Bot, perhaps by implementing a strategy similar to Greedy Bot yourself (i.e., giving boards a material score). Then, beat greedy bot by implementing what Novice Bot does (i.e., implement a preliminary adversarial search). This strategy will help you focus on one thing at a time, plus it can be fun to see your bot gradually get smarter and smarter as you implement more advanced techniques!

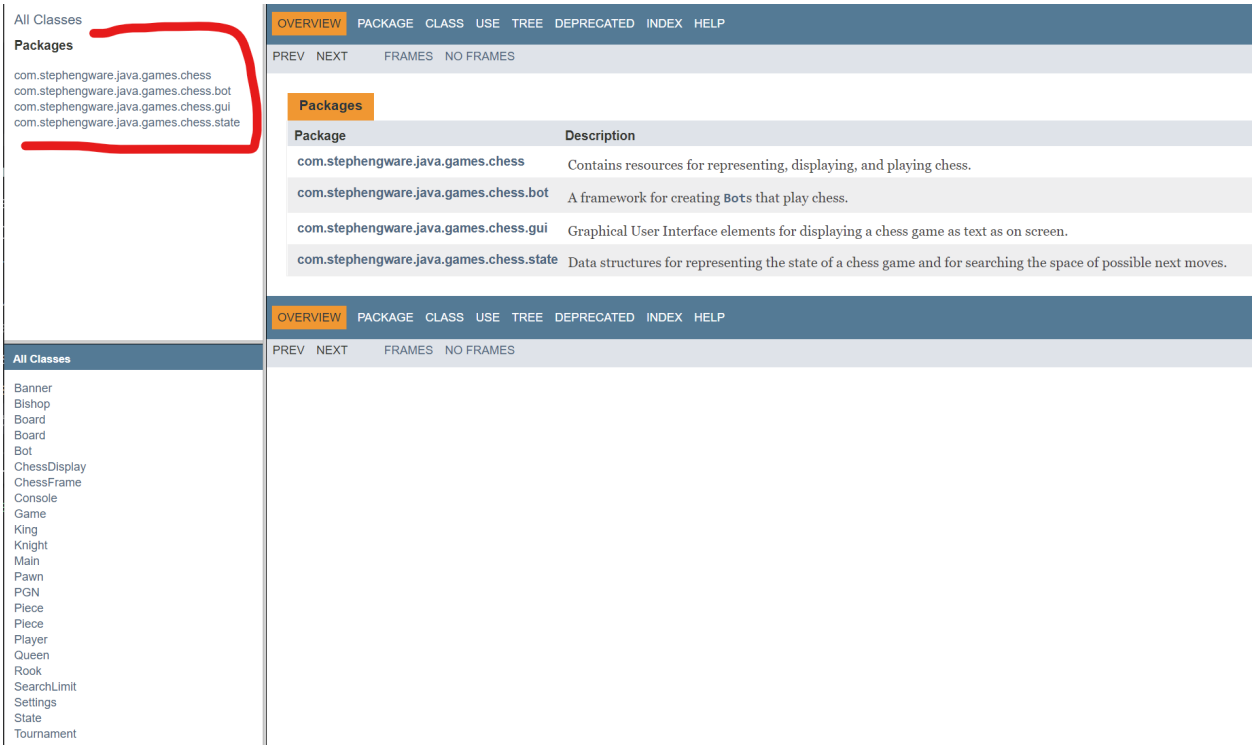
Class Tournament

Every bot which runs without throwing an exception will be entered into a class-wide chess tournament (2 games between each pair of bots). The first, second, and third place winners of this tournament will receive bonus points on this project.

Hints and Tips

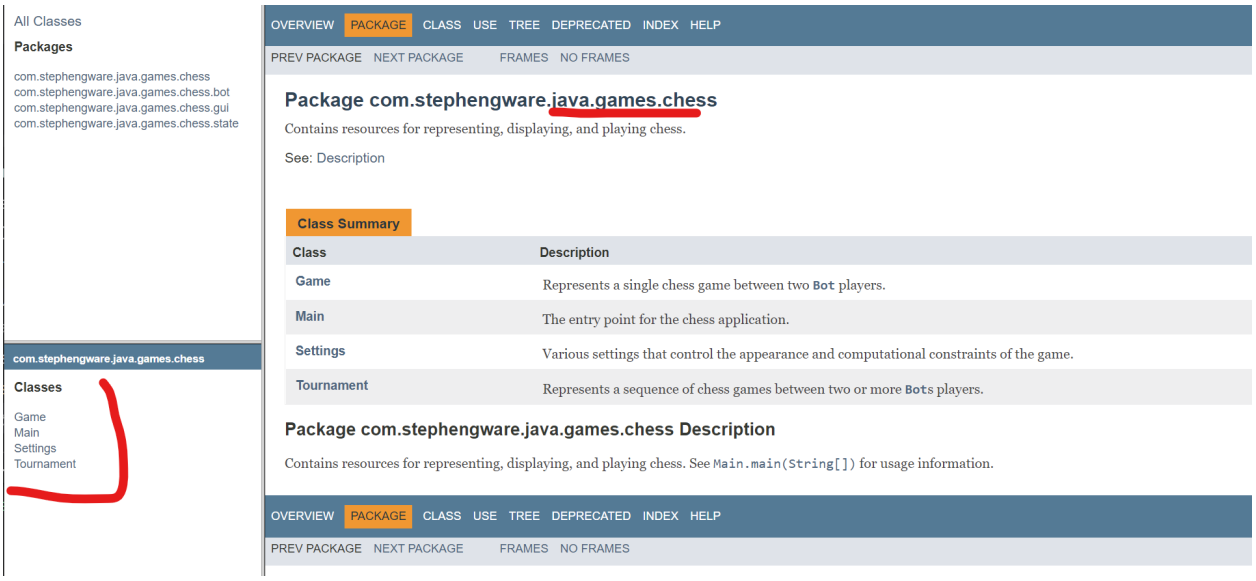
So, what's up with all this documentation? How do I read it?

Again, open the documentation by opening index.html in the doc folder you downloaded and unzipped from the Canvas page. You should see something like this:

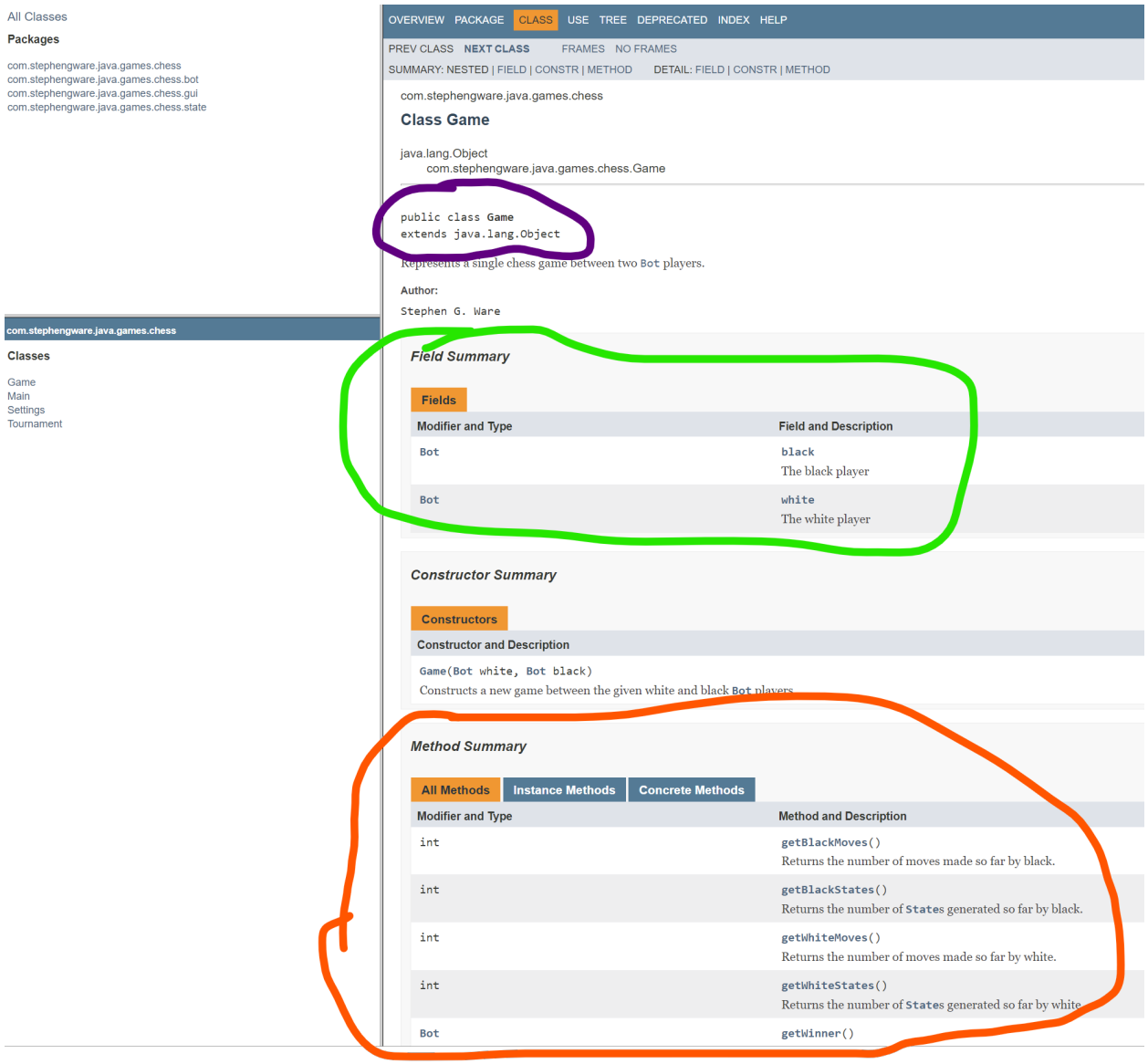


This is telling you that there are four **packages** in this framework, as shown in the upper left corner. The packages all have long names, but you can call them “chess”, “chess.bot”, “chess.gui”, and “chess.state”

As you may recall from your previous computer science courses, each Java package contains some number of **classes**. If you click on a package, you can see all of the classes it contains in the lower left. So for instance, if we click on the “chess” package, it has four classes: Game, Main, Settings, and Tournament:



You also no doubt recall from your earlier computer science courses that each Java class is a collection of Methods and Variables, and that classes can extend one another in the inheritance process. If you click on a class, you can see all of the methods and variables that the class contains, as well as the inheritance hierarchy. And remember: Java is all about types. Methods have a return type. Variables have a data type. All of that information is here in the documentation.



So for example, here is the Game class.

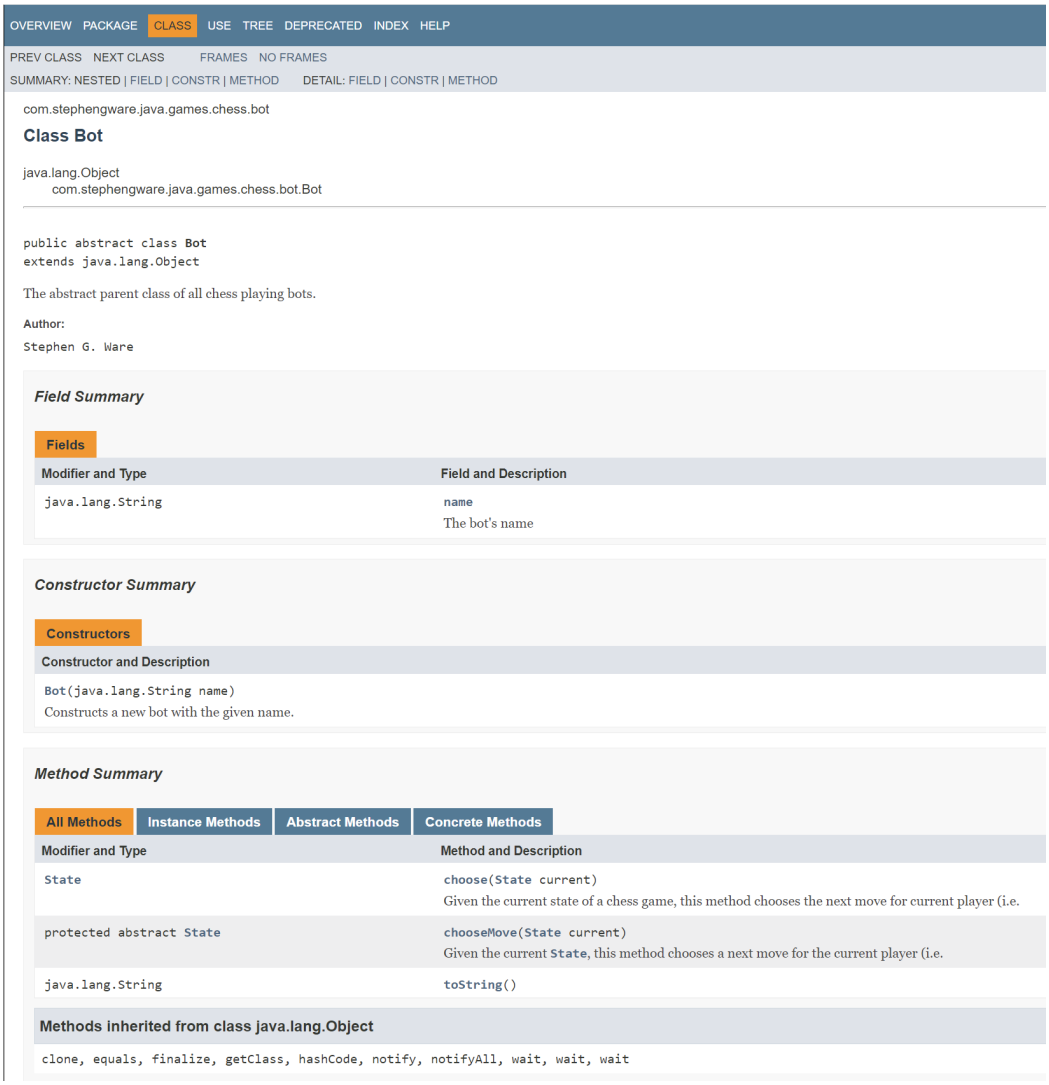
- You can see the inheritance hierarchy circled in purple (it just extends Object, no other classes in the hierarchy).
- You can see the Variables circled in green (more technically called the “Fields” of the class). You can see that the Game class has two fields – “black” and “white”, both of which are of type “Bot”. We haven’t looked at the “Bot” class yet, but you could click on the word “Bot” there and it would take you right to it!
- You can see the Methods circled in Orange. There’s quite a few. getBlackMoves(), getBlackStates(), getWinner(), etc. You can see that most return ints, though getWinner() returns a reference to a Bot object.

Do I really need to read / learn / use all of this? It seems like an awful lot…

Well, yeah, it is a lot! This is the entire game of Chess given to you (and then some!) that you don't have to program yourself! It makes sense that there's a lot of documentation.

However, that also means that a lot of the documentation included is for just getting the Chess game up-and-running, and probably won't need to be touched by you in order to actually program your own bot. Being able to recognize what is "useful" to you directly in this way is perhaps not immediately apparent. So let me help you out with that!

As discussed above, what you are actually creating for this assignment is a chess playing agent, i.e., a chess Bot. That is, you will be creating a Java class that extends the Bot class and implements the chooseMove() method. Therefore, the Bot class (the only class of the chess.bot package) is clearly of importance to you as it is what you are making.



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

com.stephengware.java.games.chess.bot

Class Bot

java.lang.Object
com.stephengware.java.games.chess.bot.Bot

```
public abstract class Bot
extends java.lang.Object
```

The abstract parent class of all chess playing bots.

Author:

Stephen G. Ware

Field Summary

Fields

Modifier and Type	Field and Description
java.lang.String	name The bot's name

Constructor Summary

Constructors

Constructor and Description
Bot(java.lang.String name) Constructs a new bot with the given name.

Method Summary

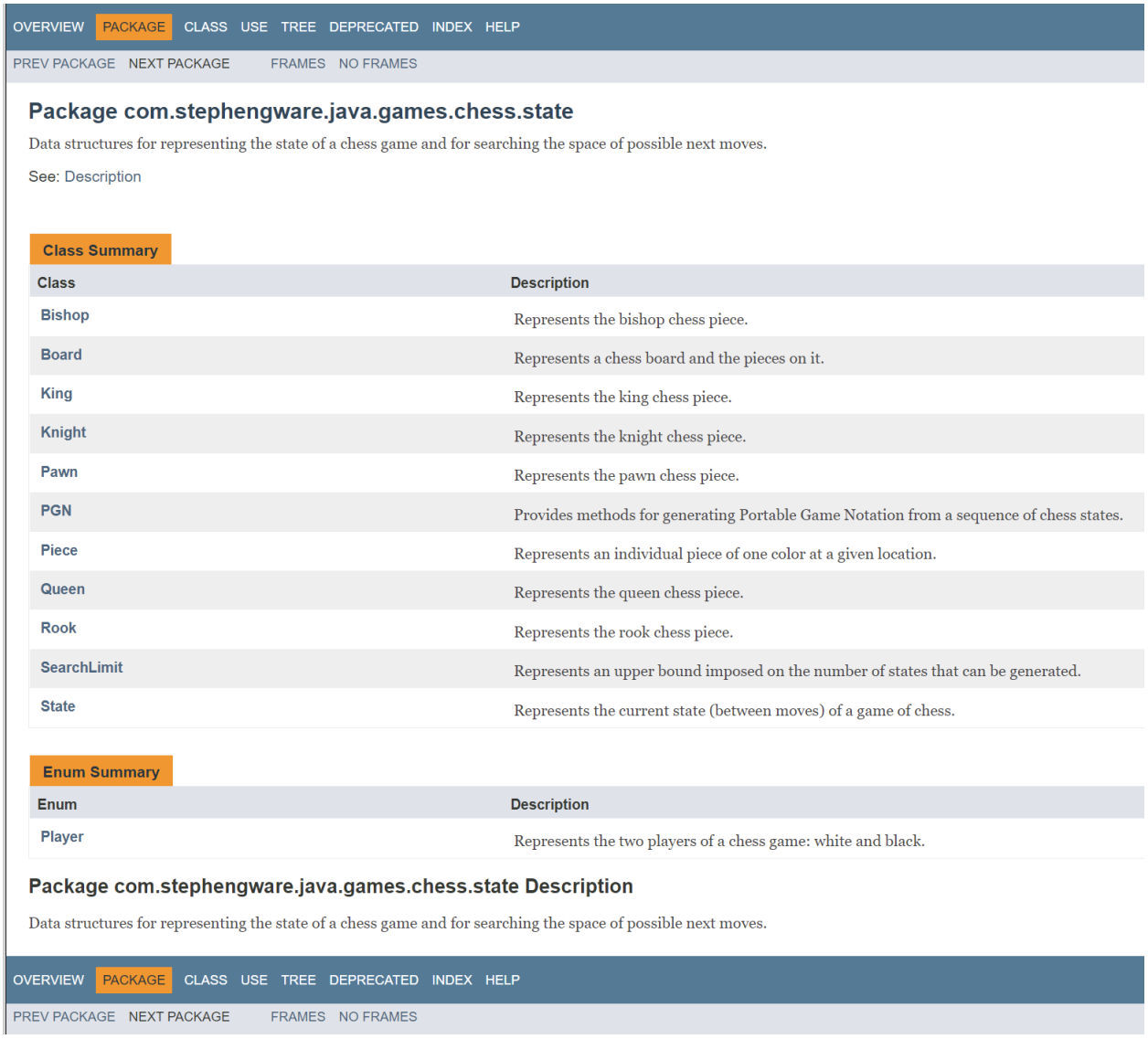
All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
State	choose(State current) Given the current state of a chess game, this method chooses the next move for current player (i.e.
protected abstract State	chooseMove(State current) Given the current State, this method chooses a next move for the current player (i.e.
java.lang.String	toString()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

However, the heart of what you are doing for this assignment is implementing the "chooseMove()" method. And in order to successfully choose a move, you'll need to compare and contrast different game states to figure out which ones will most likely lead you down the path of victory. Therefore, you'll definitely want to acquaint yourself with the chess.state package.



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package com.stephengware.java.games.chess.state

Data structures for representing the state of a chess game and for searching the space of possible next moves.

See: Description

Class Summary

Class	Description
Bishop	Represents the bishop chess piece.
Board	Represents a chess board and the pieces on it.
King	Represents the king chess piece.
Knight	Represents the knight chess piece.
Pawn	Represents the pawn chess piece.
PGN	Provides methods for generating Portable Game Notation from a sequence of chess states.
Piece	Represents an individual piece of one color at a given location.
Queen	Represents the queen chess piece.
Rook	Represents the rook chess piece.
SearchLimit	Represents an upper bound imposed on the number of states that can be generated.
State	Represents the current state (between moves) of a game of chess.

Enum Summary

Enum	Description
Player	Represents the two players of a chess game: white and black.

Package com.stephengware.java.games.chess.state Description

Data structures for representing the state of a chess game and for searching the space of possible next moves.

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

This package has many classes that you'll want to familiarize yourself with. Some notable ones:

- **Board**: Has many functions to identify how many pieces are on the board and where they are located. Also, check out how it implements Iterable for the Piece class. That means you can use an enhanced for loop on a board object to loop through every piece that is on it, like this (assuming here that 'board' is a variable of type Board): `for(Piece piece : board){ //stuff to do for any given piece }`

- **State**: Represents a possible snapshot of the current state, including a board object (i.e., the configuration of the pieces), if the game is over or not, whose turn it is to move, and a reference to the "previous" state.
- **Piece**: This is an abstract class, which means there aren't any objects that are instantiated AS Piece. However, there ARE Bishop, Knight, King, Queen, etc. objects which all extend Piece.

Focusing your attention between these two packages (chess.bot and chess.state) is, I think, a great place to start.

It seems like all of these things have references to Piece, but it is abstract! How do I tell if a Piece is, say, a Pawn versus a Queen?

Java has a few ways that you can do this! One is with the `getClass()` method – you can invoke the `getClass()` method on a '`Piece`' reference to see what its actual class is. For example, assume you have a reference of type `Piece` called `piece`, you could write:

```
if(piece.getClass() == Pawn.class){
      //turns out piece was a Pawn! Treat this piece as such!
      //e.g., maybe increase the material score by a pawn's value!
}
else if(piece.getClass() == Knight.class){
      //turns out piece was a Knight! Treat this piece as such!
      //e.g., maybe increase the material score by a knight's value!
}
```

When I run a tournament it all goes so fast, it's kind of hard to tell *where* my bot is going wrong. All I see is the final score, so I know my bot is having trouble, but how can I tell WHY my bot is having trouble?

This is a great question. There's lots of debugging strategies, but something that can be REALLY helpful is running a tournament between your bot and the "human.jar" agent. That means that you'll be able to play against your bot. You can make moves nice and slowly, and pay attention to how your bot behaves. You can "set your bot up for success",

e.g., move one of your pieces in a way that you think your bot should capture it, and then "see what happens" (i.e., you can see if it *doesn't* capture it, and hopefully use that to help you track down the underlying issue).

That, combined with traditional debugging techniques, can be really helpful to you! Don't forget: `System.out.println()` is your friend! Also, many of the provided classes have useful `toString()` overrides that you can use to 'see' the objects that you'll be working with.

In class we talked about how adversarial search gives each state a utility score. I want to associate a utility score with a state, but because I can't change the source code to state, I can't give it a consistent score. How do I do this?

As with most programming tasks, there are a lot of ways you could approach this. One possible solution is to create a helper class that links together any given state and its utility score. Here is the code for such a class which you are welcome to use if you so wish. You can see it is just a simple data structure that allows you to 'group together' a reference to a state object and a utility score.

```
private static final class Result {

	public final State state;
	public final double utility;

	public Result(State state, double utility) {
		this.state = state;
		this.utility = utility;
	}
}
```

I'm looking at the documentation, and I don't see anything that says "checkmate" and "stalemate" – how do I make my bot care about winning/losing/drawing?

This is a good chance to practice reading the documentation! But it also depends on being familiar with Chess terminology, so it can be a little tricky even with the documentation. Let's think about it!

First of all, the terminology. There's three words to know about here:

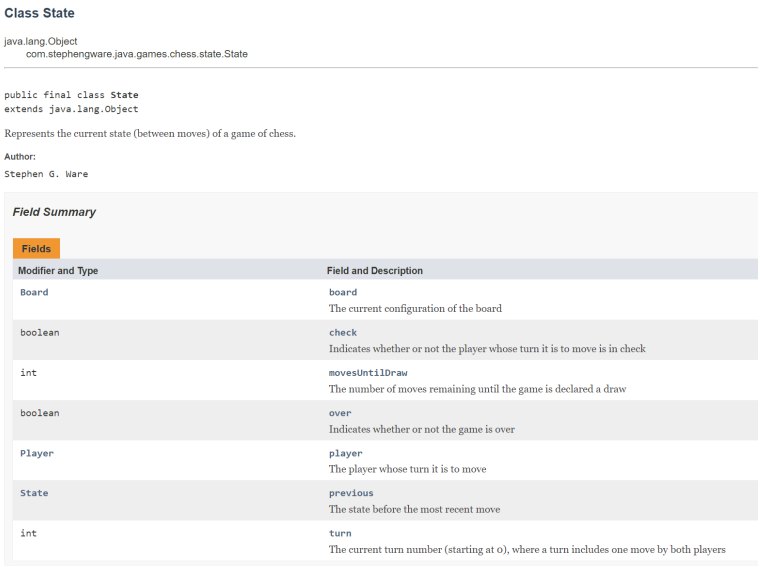
Check: This means that a King is threatened (i.e., an enemy piece can capture it), but that there exists a way for the player that is in check to 'escape' – maybe they can move

their king out of danger, or perhaps they can capture or block the piece that is threatening the king. If you are in check, you *need* to take a move that takes you out of check; you aren't allowed to make a move that would keep you in check.

Checkmate: Similar to check above, but it means that there *isn't* a way for the player in check to escape. When checkmate is reached, the game is over. The player in checkmate has lost.

Stalemate: This also means that the game is over, but that it has ended in a draw. This situation occurs when a player can make no legal moves, but their king is not in check. i.e., the king is not actively threatened, but any possible move the player would normally be able to make would place the king in danger.

So – let's say that you invest some time thinking about this and decide that "checkmate" and "stalemate" are probably properties of any given game state. And you remember that the "State" class exists and might be a good place to start looking. So you go there in the documentation, and indeed, you see the following:



Class State

java.lang.Object
com.stephengware.java.games.chess.state.State

```
public final class State
extends java.lang.Object
```

Represents the current state (between moves) of a game of chess.

Author:
Stephen G. Ware

Field Summary

Fields

Modifier and Type	Field and Description
Board	board The current configuration of the board
boolean	check Indicates whether or not the player whose turn it is to move is in check
int	movesUntilDraw The number of moves remaining until the game is declared a draw
boolean	over Indicates whether or not the game is over
Player	player The player whose turn it is to move
State	previous The state before the most recent move
int	turn The current turn number (starting at 0), where a turn includes one move by both players

So! There's the word 'check' – awesome! A Boolean variable that indicates whether or not the player whose current turn to move is in check. That's one of three down! But there's no checkmate or stalemate. Hmm!

BUT – there *is* the Boolean variable 'over' which indicates if the game is over or not.

You can combine these guys together to capture all of the situations you need:

- If 'over' and 'check' are both true – the game is over in a checkmate
- If 'over' is true and 'check' is false – the game is over in a stalemate.

- If ‘over’ is false and ‘check’ is true – the game is *not* over, but the player’s whose turn it is to move is in check, which is maybe a little scary for them.
- If ‘over’ is false and ‘check’ is false – the game is still going and the player whose turn it is to move isn’t in check.

Which can then be used however you want.

My bot just keeps on making the same move again and again! Like moving the rook back and forth forever! I’m doing all sorts of smart utility scoring for different states, so why is my bot acting in such a silly way?

Well, there could be a few reasons for this depending on what you’ve implemented so far, but a common cause of this is that there is basically a giant tie between all of your utility scores, and so your bot is always picking the first one in your list (which, if you aren’t doing any special sorting or randomization, is likely always going to involve the same piece). You can do some quick checks to see if this is what is happening for you (e.g., printing out the utility scores of all of your moves and verifying that there isn’t a giant tie going on), and a fun quick fix is that, when there a tie between multiple states with the same highest score, choosing one of those top states randomly to be the move you make.

However, that quick fix probably isn’t addressing the underlying problem: why was there a giant tie in the first place? Again, lots of possible reasons why depending on what you’ve done, but a common pitfall is going *too far down the search tree* for a single move. The search space of Chess is massive, and it’s really easy to spend all of your allotted state expansions for a turn just wondering how good it is to move a single piece (e.g., your bot has looked 20 moves ahead to figure out all of the possible outcomes that could arise from moving a single pawn, but hasn’t placed any consideration into moving any of the other pieces).

So don’t be afraid to implement (and then play around with the parameters of) a depth limited search. Even though intuitively you might think “more moves ahead = smarter moves”, that isn’t necessarily the case if it results in you using up your move budget prematurely. Setting a small depth limit (e.g., 2) does mean you’re only thinking slightly into the future, but it also means that your bot will likely consider moving every piece, and will thus be able to generate meaningful utility scores for each child state from the current state (and in so doing, reduce the likelihood of utility score ties).

Ugh. This is too much. Just tell me what to do first. And then what to do after that.

Sure! If it were me, I’d start by making a function called ‘`evaluateState()`’ that takes in a ‘`State`’ object as an argument and returns a number that measures ‘board

goodness' for any given player, with higher scores being "more good." A first pass of this might just be returning the difference in material scores (e.g., my perception of how good this state is would be *my material score* minus *your material score*).

One thing to keep in mind: when you write this, be consistent about whose "perspective" you are writing it from. There are two typical approaches here:

1.) You write it from 'white's perspective' – so if the white player is doing better the method will return a high score, and if the black player is doing better the method will return a low score. If you do it this way, you need to pay attention to what color YOU are currently playing as (and write your code so that, if you are black, you WANT a low score!)
2.) You write it from "your perspective" – so if YOU are doing better than your enemy (regardless of what color you are), then it returns a high score, and if you are losing, then it returns a low score. If you do it this way, then "high numbers are always good" for you, but you need to pay attention to who is the owner of each piece as it will change from game to game (i.e., sometimes the white pieces are yours, and sometimes the black pieces are yours).

You can then make the evaluation function fancier by also taking into account game ending states (see a previous tip from above regarding check, checkmate, and stalemate).

You can make your evaluation function *even more fancy* by taking still more things into account. There are a ton of Chess resources for expert strategy on the internet you can search for. Here's a nice one that modifies a piece's material score by it's current position (e.g., if it is in a 'good' position it gets a small bonus, if it is a 'bad' position it gets a small penalty).

Regardless, your evaluation function will only get you so far if you only look at the moves you can immediately make. If you want your bot to perform well, you'll need to look ahead to future moves. The technique we describe in class; minimax search with alpha beta pruning, is a good way to achieve this for this assignment.

This is traditionally a challenging project for students, but is also generally regarded as the hardest part of the whole class! If you can make it past this, you're gonna be OK! Get started early, ask lots of questions, come to office hours, and, you know… try to have fun with it!

It is a lot of work, but like… still kinda cool!