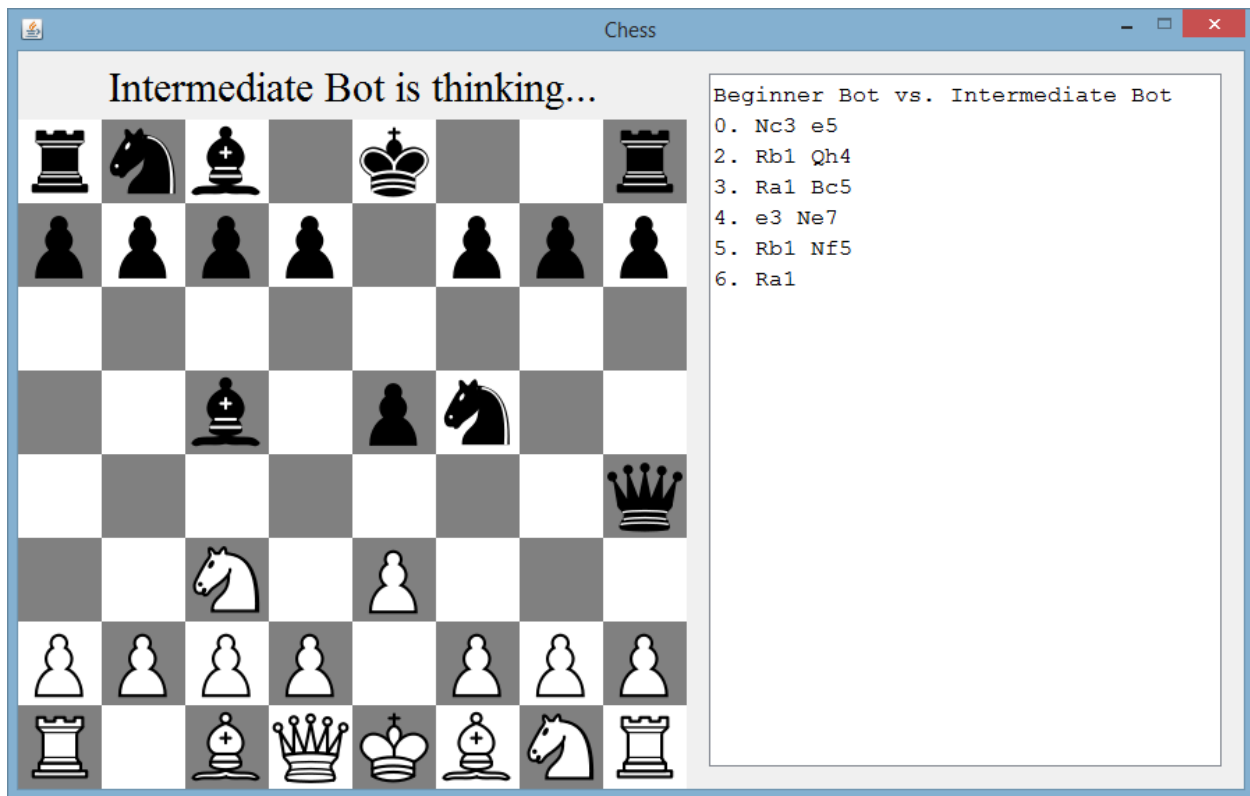


# Project 1: Chess Bot

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](#)
- [Assignment](#)
- [Grading](#)
- [About the Bots](#)
- [Class Tournament](#)
- [Hints and Tips](#)

## 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. Slightly more advanced terms, such as *rank* and *file* to describe the [chessboard](#) and the [material score](#) to evaluate how any given player is doing is also valuable.

The chess framework you will be using for this project (again, generously provided by Dr. Stephen Ware), is written in [Java 9](#). To work on this project, you will need to download the latest version of the Java Virtual Machine (anything later than Java 9 should do the trick), 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.

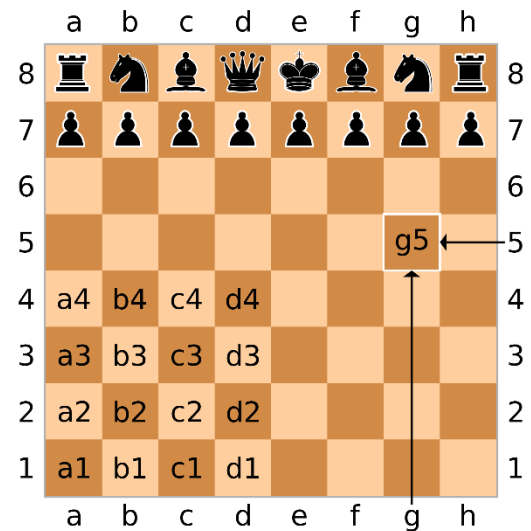
Once Java is installed, download the following elements from the Moodle page:

- 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 and future projects, your success will hinge on not being afraid to look through the documentation. To access it, simply unzip doc.zip, and then double click on "index.html" – it should open up the documentation in a browser for you to peruse. 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 represents a different chess-playing program, or "bot" (except for human.jar, which is a mechanism that let's YOU make the moves instead of a bot, if you want to play against any of these bots yourself).



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"

**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`. 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, 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 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).

So 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.

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](#). I definitely recommend taking a few minutes brushing up on PGN, but 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 handily best 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 moodle), 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 adding each 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.

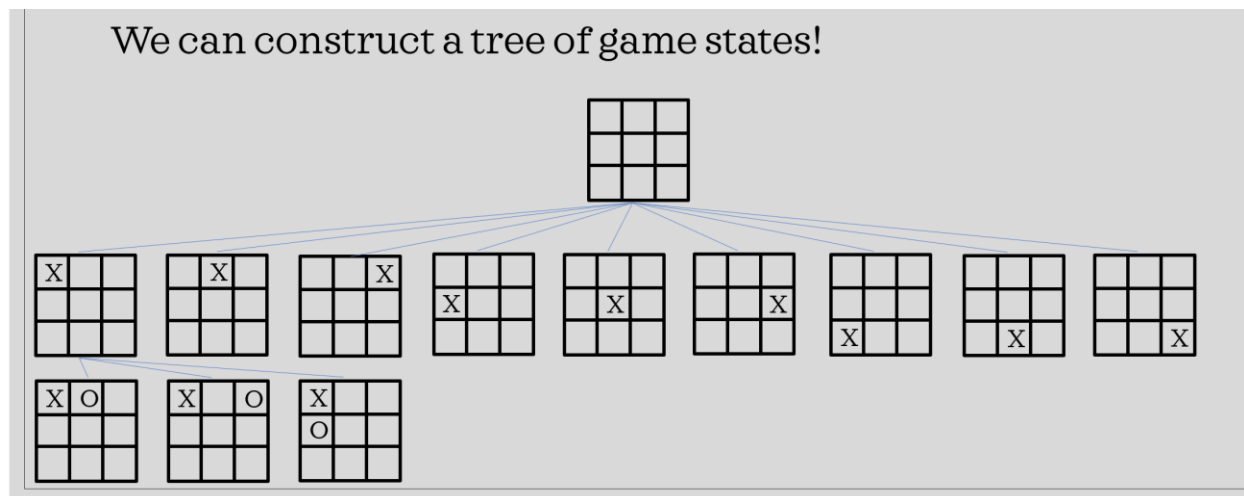
It is also recommended that you rename the Eclipse project to reflect your name:

- 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. Because my university e-mail address is [bsamuel@cs.uno.edu](mailto:bsamuel@cs.uno.edu), my ID would be bsamuel.

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 might have moved).

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, 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.

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 Moodle 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.

Since my ID is `bsamuel`, my bot's name will be `bsamuel.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 Moodle.**

## **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 bsamuel.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 Moodle by the deadline.
- Submit your jar file, named for your student ID (e.g., bsamuel.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 before hand!
- 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 chooses 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.
- **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 considered 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’s kinda 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 Moodle page. You should see something like this:

The screenshot displays a Java documentation interface. On the left, a sidebar lists 'All Classes' and 'Packages'. The 'Packages' section is highlighted with a red box and contains the following entries:

- com.stephengware.java.games.chess
- com.stephengware.java.games.chess.bot
- com.stephengware.java.games.chess.gui
- com.stephengware.java.games.chess.state

The main content area shows a table of packages with the following columns: Package and Description.

Package	Description
com.stephengware.java.games.chess	Contains resources for representing, displaying, and playing chess.
com.stephengware.java.games.chess.bot	A framework for creating Bots that play chess.
com.stephengware.java.games.chess.gui	Graphical User Interface elements for displaying a chess game as text as on screen.
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.

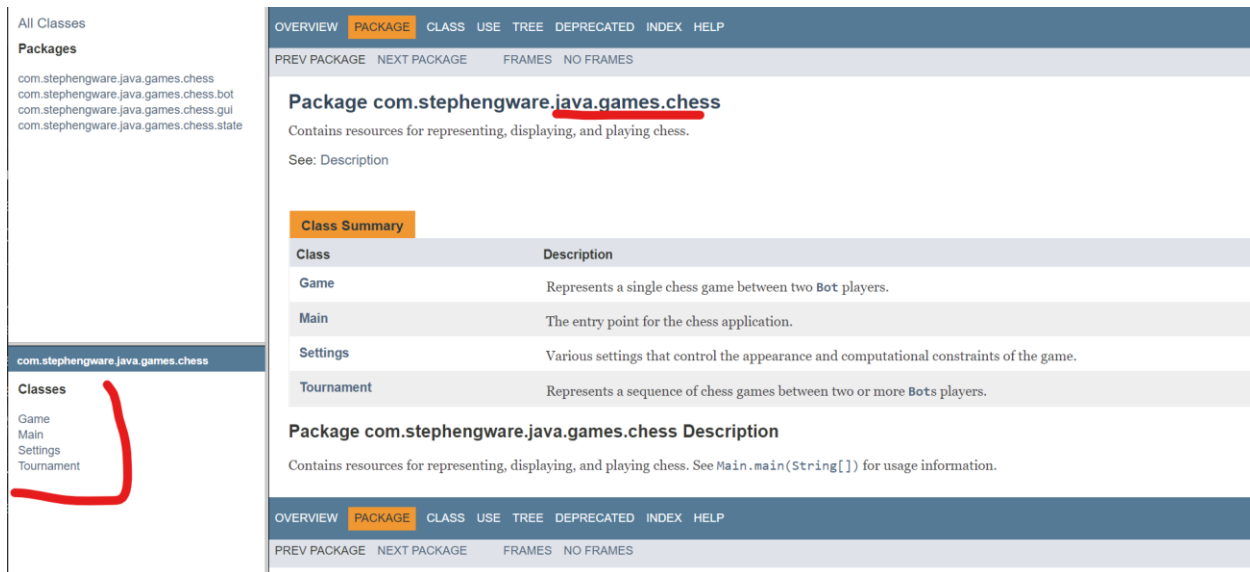
Below the package table, there is another navigation bar with tabs: OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, HELP. The 'OVERVIEW' tab is selected. Below this, there is a list of classes under the 'All Classes' section:

- Banner
- Bishop
- Board
- Board
- Bot
- ChessDisplay
- ChessFrame
- Console
- Game
- King
- Knight
- Main
- Pawn
- PGN
- Piece
- Piece
- Player
- Queen
- Rook
- SearchLimit
- Settings
- State
- Tournament

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 classes, each 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:



The screenshot shows the Java documentation for the package `com.stephengware.java.games.chess`. The left sidebar has a section titled "Classes" with a red bracket highlighting the list of classes: `Game`, `Main`, `Settings`, and `Tournament`. The main content area shows the package details, including a "Class Summary" table and a "Description".

**Package `com.stephengware.java.games.chess`**  
Contains resources for representing, displaying, and playing chess.  
See: [Description](#)

**Class Summary**

Class	Description
<code>Game</code>	Represents a single chess game between two <code>Bot</code> players.
<code>Main</code>	The entry point for the chess application.
<code>Settings</code>	Various settings that control the appearance and computational constraints of the game.
<code>Tournament</code>	Represents a sequence of chess games between two or more <code>Bots</code> players.

**Package `com.stephengware.java.games.chess` Description**  
Contains resources for representing, displaying, and playing chess. See `Main.main(String[])` for usage information.

You also no doubt recall from your earlier classes that each 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.

The screenshot displays the Java IDE interface for the `Game` class. The left sidebar shows the package structure: `com.stephensware.java.games.chess`, `com.stephensware.java.games.chess.bot`, `com.stephensware.java.games.chess.gui`, and `com.stephensware.java.games.chess.state`. The main window shows the `Game` class documentation. The inheritance hierarchy is circled in purple, showing `public class Game` extending `java.lang.Object`. The Fields section is circled in green, showing two fields: `Bot` (black) and `Bot` (white). The Method Summary section is circled in orange, showing methods: `getBlackMoves()`, `getBlackStates()`, `getWhiteMoves()`, `getWhiteStates()`, and `getWinner()`.

**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.stephensware.java.games.chess

### Class Game

java.lang.Object  
com.stephensware.java.games.chess.Game

public class Game  
extends java.lang.Object

Represents a single chess game between two Bot players.

Author:  
Stephen G. Ware

#### Field Summary

**Fields**

Modifier and Type	Field and Description
Bot	black The black player
Bot	white The white player

#### Constructor Summary

**Constructors**

Constructor and Description
Game(Bot white, Bot black) Constructs a new game between the given white and black Bot players

#### Method Summary

**All Methods** **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
int	getBlackMoves() Returns the number of moves made so far by black.
int	getBlackStates() Returns the number of States generated so far by black.
int	getWhiteMoves() Returns the number of moves made so far by white.
int	getWhiteStates() Returns the number of States generated so far by white
Bot	getWinner()

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 its 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 | METHODDETAIL: 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. which all extend Piece.

Focusing your attention between these two packages 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!
}
else if(piece.getClass() == Knight.class){
    //turns out piece was a Knight!
}
```

**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?**

There are a lot of ways to do this, but 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;  
    }  
}
```