

Table of Contents

Overview.....	2
Installation.....	2
Example.....	2
First tournament.....	2
Building your first AI.....	3
Configuring a tournament.....	3
Changing the AI.....	4
Debugging.....	6
Quick Reference.....	8
Javadocs.....	8
What we're working with.....	8
AI Interface.....	8
GameState.....	8
Card.....	9
Move.....	9
MoveChecker.....	9
Naive AI.....	9
How to run a tournament.....	10
How to build the AI.....	10
Configuring a tournament.....	10
Logs/Results of the tournament.....	10
Stats.....	12
Overall stats.....	12
AI specific stats.....	12
Game specific stats.....	12
Readable Log.....	13
How to use logs.....	13
Cleaning Logs.....	13
Last Card Rules.....	14
How To Win.....	15
Hand value.....	15
Calling Last Card.....	15
FAQ.....	16
I'm feeling lost, where do I start?.....	16
The commands aren't working, what's happening?.....	16
The logs say really huge numbers (eg 200+ illegal moves). What?.....	16
The formatting of the table is screwed up!.....	16
Do I really have to implement all of the last card rules?.....	16
I get a null pointer error, but the tournament continues. What?.....	16

Overview

The objective of this coding challenge is to create an AI that plays a game of last card: a card game in which the objective is to get rid of all the cards in your hand. Your AI will take part in a tournament that will play over 1000 games to determine the winner! The winner is determined based on taking the least number of “penalty points” per game, which is based on the cards left in your hand at the end of the game. You also take penalty points for calling last card too often. Just to be clear, you don’t actually have to implement the rules of last card, we provide a [LastCardMoveChecker class](#) that you can use. Your goal is just to implement a strategy to win.

Installation

1. [Download the project](#)
2. Import the project into Eclipse (File > Import > Maven > Existing Maven projects)
(You can use an alternative IDE such as IntelliJ if you want)

Example

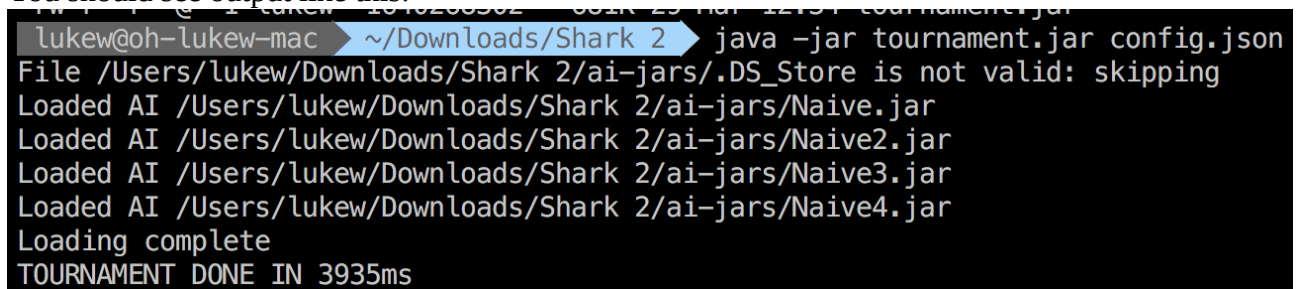
This section gives a quick example of how you would go about creating the AI as well as how to test it and improve it. Feel free to skip this if you want to get straight into coding (but there’s a lot of useful stuff in here!). Also, if you don’t already know the basics of playing last card, it might pay to have a quick look at the [Last Card Rules section](#). Also make sure to check out the [javadocs](#) for useful info about the classes we use.

First tournament

To start with, let’s see what happens when we run a tournament. Make sure you’re in the directory with tournament.jar, and run the following command:

```
java -jar tournament.jar config.json
```

You should see output like this:



```
lukew@oh-lukew-mac ~/Downloads/Shark 2$ java -jar tournament.jar config.json
File /Users/lukew/Downloads/Shark 2/ai-jars/.DS_Store is not valid: skipping
Loaded AI /Users/lukew/Downloads/Shark 2/ai-jars/Naive.jar
Loaded AI /Users/lukew/Downloads/Shark 2/ai-jars/Naive2.jar
Loaded AI /Users/lukew/Downloads/Shark 2/ai-jars/Naive3.jar
Loaded AI /Users/lukew/Downloads/Shark 2/ai-jars/Naive4.jar
Loading complete
TOURNAMENT DONE IN 3935ms
```

As we can see, the tournament has completed, with 10 games in total played. Now we want to see the results of the tournament, so open up the logs directory in your preferred text editor (I’ll be using sublime text, but atom or similar works – it’s best to have something that you can quickly switch between different files in). Open up overall-stats.log. You should see a table looking similar to this:

	AI	pass moves	hand value pts	points taken	cards played	punis
	Naive.jar	3.90	5.90	5.90	10.40	
	Naive2.jar	4.80	7.00	7.00	10.80	
	Naive3.jar	3.50	9.50	9.50	11.30	
	Naive4.jar	3.80	4.60	4.60	11.70	

From this, we can see the AIs that took part in the tournament and some interesting stats about how they did. These stats are all based on the average across all games the AI played. In the above example, we can see that the ai all performed in a similar manner with regards to pass moves, cards played etc. We can also see that Naive4.jar would win the tournament, as it has the lowest “points taken” per game.

Building your first AI

To build the project, first ensure you are in the shark-impl directory (there should be a pom.xml file here). Then run the following command from the terminal:

```
mvn package
```

Now if you look in the target directory, you should see a .jar file named “please-change-this-to-a-unique-name”. Let’s rename that to whatever you want (your name might be a good idea when submitting as it needs to be unique). **NOTE: You have to rename/move the generated jar file every time you run the mvn package command**

Configuring a tournament

Let’s take a look at the config.json file (this will be in the main directory, with tournament.jar):

```
{
  "ai_dirs": ["ai-jars"],
  "ai": [],
  "num_games": 10
}
```

Here we see a few options that you can use to configure the tournament. To make sure the AI we created takes part in the tournament, it needs to either be in the list of ais, or in one of the directories in ai_dirs. For now, only the naive AIs are in the tournament. Let’s use the first way for now: add your ai to the list like this:

```
{
  "ai_dirs": ["ai-jars"],
  "ai": [ "shark-impl/target/upi1234.jar" ],
  "num_games": 10
}
```

As well as choosing which AIs compete in a tournament, you can specify the number of games by changing the `num_games` field. Let's bump it up to 50, as 10 games doesn't really give an accurate picture of how well our AI is doing.

```
{
  "ai_dirs": ["ai-jars"],
  "ai": [ "shark-impl/target/upi1234.jar" ],
  "num_games": 50
}
```

Now let's run the tournament again:

```
java -jar tournament.jar config.json
```

Once it's done running (it should take about 20-30s), open up the overall logs again. It should look pretty similar to this:

	AI	pass moves	hand value pts	points taken	punishment taken	illegal moves	cards played	games won	times punished	last card pts
	Naive.jar	26.20	5.54	5.54	5.37	-	10.85	0.12	1.17	0.00
	Naive2.jar	32.34	2.87	2.87	3.82	0.03	11.82	0.47	0.92	0.00
	Naive3.jar	31.05	3.60	3.60	3.55	0.10	13.52	0.17	1.05	0.00
	Naive4.jar	42.00	4.02	4.02	4.15	0.02	13.44	0.32	1.12	0.00
	upi1234.jar	53.45	41.55	41.55	7.60	0.03	-	-	1.80	0.00

From the results, we can see that our AI didn't do very well! The main stat that tells us this is the "points taken" stat, which shows the number of penalty points taken each game. Here, we can see that our AI lost quite heavily to the other AIs. Interestingly, we can also see that it had way more pass moves than the other AIs. We can use this information to improve our AI!

Changing the AI

Let's open up the project in eclipse, and look at the file `YourStrategy.java` in the `com.orchestral.shark` package in `src/main/java`.

You should see this:

```
public class YourStrategy implements CardGameStrategy {
    @Override
    public Move playCardFromHand(final GameState gamestate) {
        return Move.PASS_MOVE;
    }
}
```

The function you'll be implementing to create your AI is the `playCardFromHand` function. Currently, it plays a default pass move no matter what, which doesn't seem like a great strategy. Let's make it do something more interesting!

The only argument to the function is an instance of the `GameState` class, so take a quick look at the outline of that class in the [Quick Reference section](#) to get an idea of what information you can use.

Ok, so we want our AI to actually play a card rather than pass every turn, so one relevant piece of info from `GameState` is the cards in our hand (as we can't play a card if it's not in our hand!). So how do we tell our AI to play a card?

The function we're implementing returns an instance of the `Move` class, so take a quick look at that in the [Quick Reference section](#).

The easiest way to return the move we want to is to use the `Move Builder`. Here's an example of how to do so. This example simply plays the first card in our hand.

```
@Override
    public Move playCardFromHand(final GameState gamestate) {
        final Move.Builder builder = new Builder();
        return builder.addCardPlayed(gamestate.getMyHand().get(0))
            .withLastCard(false)
            .build();
    }
```

Now that we have a slightly more interesting AI, let's try it out!
Build the ai using the same command as last time:

```
mvn package
```

Now, to add it to the tournament, let's try the other way! Move the generated `.jar` file to a new folder called "my-ais". Then, update the config file: it should now look like this (after removing our first AI).

```
{
  "ai_dirs": ["ai-jars", "my-ais"],
  "ai": [ ],
  "num_games": 50
}
```

Debugging

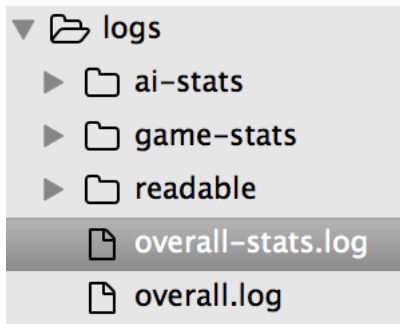
We're all configured and ready to go, so let's run the tournament again:

```
java -jar tournament.jar config.json
```

Once it's done, open up `overall-stats` again.

	AI	pass moves	hand value pts	points taken	cards played	illegal moves	punishment taken	games won	times punished	last card pts
Naive.jar		51.95	3.97	3.97	11.51	0.08	3.49	0.23	0.95	0.00
Naive2.jar		35.32	5.39	5.39	11.47	-	5.32	0.24	1.16	0.00
Naive3.jar		33.58	4.13	4.13	11.95	-	4.97	0.35	1.33	0.00
Naive4.jar		53.21	4.60	4.60	11.81	-	5.50	0.21	1.14	0.00
upi1234.jar		-	47.20	47.20	4.17	62.00	7.95	-	62.00	0.00

Well, we're doing less pass moves. But now, we can see that we're making some illegal moves! Let's look deeper to figure out why. The logs folder should have a structure like this:



We can look at logs/ai-stats/upi1234.jar.log to look at the stats for our ai for each game, giving us a better indication of where our AI didn't do too well.

gameID	hand value pts	points taken	cards played	illegal moves	punishment taken	times punished	last card pts
0	62.00	62.00	5.00	16.00	17.00	16.00	0.00
2	37.00	37.00	6.00	13.00	2.00	13.00	0.00
4	32.00	32.00	-	6.00	5.00	6.00	0.00
6	70.00	70.00	6.00	26.00	5.00	26.00	0.00
7	19.00	19.00	3.00	4.00	0.00	4.00	0.00
8	64.00	64.00	8.00	243.00	13.00	243.00	0.00

Quick note: to see why game 8 had 243 illegal moves, see the [FAQ re: long games](#)

From this, we can see that in game 6, we made a lot of illegal moves (your results will be slightly different, so find a similar game in your results). If we look at the logs for this specific game in logs/game/0006.log, we can confirm that we played a lot of illegal moves this game. However, this doesn't give us a ton of info. Why did we make illegal moves? To find this out, we can instead look at logs/readable/0006.log.

```
Naive4.jar passes
Naive4.jar picks up 10 cards as punishment
upi1234.jar plays card(s) [The EIGHT of DIAMONDS]
Illegal Move!
Either suit or value did not match
Naive.jar plays card(s) [The TEN of HEARTS]
Naive4.jar plays card(s) [The JACK of HEARTS]
Naive2.jar plays card(s) [The JACK of DIAMONDS]
Naive4.jar plays card(s) [The KING of DIAMONDS]
upi1234.jar plays card(s) [The EIGHT of DIAMONDS]
Naive.jar plays card(s) [The NINE of DIAMONDS] and calls last card(s)
Naive2.jar plays card(s) [The TWO of DIAMONDS]
Naive4.jar plays card(s) [The TWO of SPADES]
upi1234.jar plays card(s) [The SEVEN of DIAMONDS]
Illegal Move!
Can't just play the same suit on a 2/5 to avoid punishment
upi1234.jar picks up 4 cards as punishment
Naive.jar passes
Naive2.jar passes
Naive4.jar plays card(s) [The NINE of SPADES]
upi1234.jar plays card(s) [The SEVEN of DIAMONDS]
Illegal Move!
Either suit or value did not match
```

The readable logs help reconstruct the events of the game: who played which cards, what illegal moves were made and much more. Here, we can see there are quite a lot of reasons we were making illegal moves. Probably because last card has rules that our AI isn't following yet. The next steps are up to you! A good idea would be to look over the quick reference guide to see what info you can use. For example, there might be something you can use to ensure your AI plays less illegal moves. Hopefully this example has given you an idea of how to debug, improve and iterate on your AI. Make sure to check through the quick reference section below, and if you have any troubles feel free to ask questions!

Quick Reference

Javadocs

You should be able to hover over the classes/methods in Eclipse to get more info about them. Alternatively, view the javadocs here: <http://shark.odl.io/docs/>

What we're working with

AI Interface

To create your AI, you need to implement the CardGameStrategy interface. Skeleton code is already set up for you, so all you need to do is implement the playCardFromHand function in the yourCardGameStrategy class provided.

The playCardFromHand function gives you a GameState instance containing information about the current state of the game such as the last card played and the number of cards in everyone else's hand. You need to use this information to make a decision about what Move to play. For more information on the GameState and Move classes, see below.

GameState

A GameState contains information about a game of last card at a certain point in time. It contains the following fields:

Field	Meaning
playerHandSizes	A list of integers representing the number of cards left in each player's hand
deckSize	The number of cards currently in the deck
playedCards	The list of cards played in order (cards may be removed from this list when a player picks up and there are no cards left in the deck)
lastPlayedCard	The last played card – you generally have to match the suit or value of this card
myHand	A list of Cards representing your hand – you have to play one of these cards
chosenSuit	If a player plays an Ace, they get to decide what the current suit is. This represents the suit chosen by the previous player (and is only relevant when an Ace has been played)
punishmentCards	An integer representing the active punishment: if you pass or make an illegal move, you will have to pick up this many cards

Card

A Card is a simple representation of a card. It contains two parts: a Suit and a Value. Both Suit and Value are enumerated types, with values from Spades to Hearts and Ace to King respectively.

Move

A move represents the action you want your AI to take for it's turn during the game. You create these moves via a move builder class. For example, a move in which you play the Ace of Spades, choose Hearts as the suit and call last card might look like this:

```
builder.addCardPlayed(new Card(Suit.SPADES, Value.ACE))
    .withSuit(Suit.HEARTS)
    .withLastCard(true)
    .build();
```

A move where you play the first and third cards in your hand, but don't call last card might look like this:

```
builder.addCardPlayed(gamestate.getMyHand().get(0))
    .addCardPlayed(gamestate.getMyHand().get(2))
    .withLastCard(false)
    .build();
```

MoveChecker

The MoveChecker is a utility interface that lets you check whether what you are doing is a legal move. There is an implementation of this: LastCardMoveChecker which you should find extremely useful. To use the MoveChecker, you need to create an instance of it, and then call that instance's isLegalMove function, passing in a gameState and a Move. This function will return either true or false representing whether or not that move is legal for that gameState. Note that it does not cover every possible circumstance: if you are trying to end without calling last card, for example, it will not take this into account. For more information on the specific rules in play, check out the Last Card Rules section.

Naive AI

```
@Override
public Move playCardFromHand(final GameState gameState) {
    // The naive strategy simply plays the first playable card in its hand
    for (int i = 0; i < gameState.getMyHand().size(); i++) {
        final Move move = new Move.Builder()
            .addCardPlayed(gameState.getMyHand().get(i))

            // call last card if we currently have 2 cards left
            // (ie we will have 1 card left next turn)
            .withLastCard(gameState.getMyHand().size() == 2)
            .build();

        if (moveChecker.isLegalMove(gameState, move)) {
            return move;
        }
    }
    return Move.PASS_MOVE;
}
```

How to run a tournament

From the terminal, make sure that you're in the directory with tournament.jar.

Run this command:

```
java -jar tournament.jar config.json
```

This command runs a tournament with the configuration as defined in config.json. The results of the tournament are shown in the logs generated by this process.

How to build the AI

From the terminal, make sure that you're in the directory shark-impl.

Run this command:

```
mvn package
```

This command builds the maven project and creates your AI in the form of a .jar file. The file is created in the target directory, with the name "please-change-this-to-a-unique-name".

Configuring a tournament

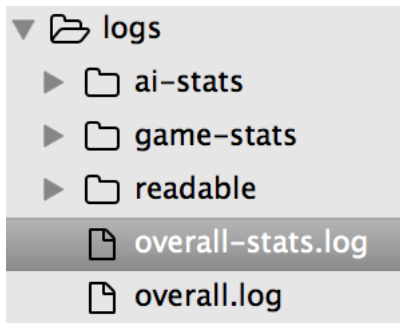
The configuration of the tournament is set in the config.json file. Here, you can set which AIs to complete in the tournament and the total number of games to play. To include an AI file, include the relative path to the file in the "ais" field. Alternatively, to include an entire directory of AIs, include the relative path to the directory in the "ai_dirs" field. Set the total number of games via the "num_games" field.

```
{  
  "ai_dirs": ["old-ais", "my-ais", "naive-ais"],  
  "ai": [ "shark-impl/target/my-ai.jar", "someRandomAI.jar"],  
  "num_games": 1000  
}
```

In the above example, we include 2 specific AIs, 3 directories of AIs and play 1000 games.

Logs/Results of the tournament

The results of the tournament are shown in the log files generated by the tournament. These logs can be found in the logs directory. The logs directory should look as follows:



Stats

Statistic name	Meaning
Pass moves	The number of passing moves the AI made
Cards played	The number of cards played by the AI
Illegal moves	The number of illegal moves the AI made
Times Punished	The number of times the AI was forced to pick up due to a 2 or 5 being played
Punishment taken	The number of cards the AI was forced to pick up due to a 2 or 5 being played
Games won	The number of games the AI won
Points taken	The number of penalty points against the AI. The winner is determined by who has the lowest value of this.
Hand value points	The number of points taken by the AI due to cards remaining in their hand at the end of games
Last card points	The number of penalty points taken by the AI due to calling last card too many times

Overall stats

AI	pass moves	hand value pts	points taken	cards played	illegal moves	punishment taken	games won	times punished	last card pts
Naive.jar	51.95	3.97	3.97	11.51	0.00	3.49	0.23	0.95	0.00
Naive2.jar	35.32	5.39	5.39	11.47	-	5.32	0.24	1.16	0.00
Naive3.jar	33.58	4.13	4.13	11.95	-	4.97	0.35	1.33	0.00
Naive4.jar	53.21	4.60	4.60	11.81	-	5.50	0.21	1.14	0.00
up1234.jar	-	47.20	47.20	4.17	62.00	7.95	-	62.00	0.00

Overall stats are shown in the overall-stats.log file. This shows the stats for each AI such as illegal moves made, passing moves etc. As an average amount per game that the AI played. This should be the first file you look at to get an overall understanding of how your AI compares to the others in the tournament.

AI specific stats

Statistics grouped by the AI are shown in the ai/<AI_NAME>.log files. These files show the stats on a per game basis for the relevant AI. This helps you identify specific games and situations where your AI has undesirable behaviour. For example, you might notice that your AI played 13 illegal moves in game 10. This can help point you to which game you want to check to see exactly where and how your AI can be improved.

Game specific stats

Statistics grouped by game are shown in the game/<GAME_ID>.log files. These files show the results of each game: who won, how many illegal moves were made by each AI in this specific game etc.

Readable Log

```
Naive.jar plays card(s) [The NINE of DIAMONDS] and calls last card(s)
Naive2.jar plays card(s) [The TWO of DIAMONDS]
Naive4.jar plays card(s) [The TWO of SPADES]
upi1234.jar plays card(s) [The SEVEN of DIAMONDS]
Illegal Move!
Can't just play the same suit on a 2/5 to avoid punishment
```

The readable logs for each game are shown in the readable/<GAME_ID>.log files. These files give a readable summary of the events that happened in the game: who played what cards or made which illegal moves or had to pick up when. This is incredibly useful for identifying the specific situations in which your AI doesn't work as well as you want it to.

How to use logs

Probably the best way to use the logs is to first look at the overall stats to get an idea of how your AI works. You can identify a metric that you want to improve, such as playing less illegal moves here. Then, open up the AI specific stats for your AI. Find an example game where the AI didn't do as well as it could have. Go through the readable log for this game and understand the situation in which your AI could have done better. This can provide valuable information as to how to improve your AI.

Cleaning Logs

If you ever want to clean up your logs (for example, running a 10 game tournament after a 100 game tournament and you don't want the old files for games 10-99), simply delete the logs directory before running the new tournament. A useful command for this is

```
rm -r logs (on mac/linux)
rd /s /q (on windows)
```

Last Card Rules

Each game consists of four players, each starting with a hand of seven cards. One card is played face up as the most recently played card, and then the game begins. Each player takes turns playing some cards from their hand with the following conditions:

- To play a card, it must have the same suit or value as the most recently played card.
- You may play zero to four cards per turn, they must all be the same value, the first card you play must match the suit or value of the most recently played card. For example, you can play three 7s if the last played card was a 7, or two 10s (starting with the 10 of hearts) on the 8 of hearts. However, you cannot play a 7 and a 10 of hearts together on the same turn, as they don't have the same value.
- If you cannot play any cards, you can pass and you will be made to draw one card.
- An ace can be played on anything, and lets you choose the currently active suit.
- If a two or a five is played, you may not play and you must draw that many cards, unless you can play another two (if a two was played) or five (if a five was played). This causes the next player to draw an additional two/five cards as the "punishment" stacks until someone is forced to pick up. An ace cannot cancel this effect.
- Playing a ten will cause the next player to miss their turn.
- Playing a jack will reverse the direction of play.
- The effects of cards stack. For example, if you play two jacks, the direction of play reverses twice, effectively cancelling itself out. If you play three tens, the next three players' turns are skipped.
- You win the game by playing all the card in your hand. When you end your turn with no cards in your hand (assuming it's a legal move and you called last card the previous turn), you win and the game ends.
- You must call last card if you think you will win next turn. If you finish without calling last card, you must draw two more cards, putting you at a disadvantage.
- You cannot end on an ace.
- If one thousand turns pass with no winner, then no-one wins and the game ends.

How To Win

The winner of a tournament is the AI with the lowest penalty points taken per game. This is shown as “points taken” in the stats logs created by the tournament. Throughout the tournament, AIs play many games of last card against each other, and accrue penalty points in two ways:

Hand value

At the end of each game, the value of the cards left in each player’s hand is given to them as penalty points. The points are the sum of the values of each card in the player’s hand, with cards having the following values:

Ace	–	5
Two, Five	–	2
Anything else	–	1

Calling Last Card

To prevent AI from calling last card every single turn, penalty points are also awarded to AI each time they call last card. These points are calculated as follows:

$$\text{Last Card Penalty Points} = \text{Number of times called last card} / 10$$

However, these points will only increase each time you go above 10 calls per game. For example, if in one game you call last card 7 times, you will not be penalised. If you call last card 30 times, you will be given 3 penalty points.

FAQ

I'm feeling lost, where do I start?

A good place to start would be to look at the [Naive AI Strategy](#). See what it does, and think of some simple ways you could improve on it.

The commands aren't working, what's happening?

First make sure you're in the correct directory to execute the command. For example, to build your AI, ensure you are in the shark-impl directory with pom.xml

Second, ensure the command itself is correct. If that doesn't work, and you're 100% sure you're doing it right, give it a few seconds and try again, or grab one of the helpers.

The logs say really huge numbers (eg 200+ illegal moves). What?

There are some situations that can happen with certain AI that lead to an impasse. For example, consider 4 AIs that pass. In this case the game will only end after 1000 moves have been made! It doesn't even have to be this obvious: if you have one AI that is the only one that can make a legal move (eg: it has all the spades, and the last played card is an Ace with Spades as the chosen suit) but it doesn't realise this and tries to play an illegal move, it will lead to the same infinite loop.

The formatting of the table is screwed up!

You can kind of improve this by reducing the font size in your document viewer.

Do I really have to implement all of the last card rules?

No. Make sure to check the [quick reference guide](#) and the [javadocs](#), the LastCardMoveChecker class can help you with this.

I get a null pointer error, but the tournament continues. What?

There is a known minor issue which shouldn't affect the tournament. It usually happens when a tournament is run really fast after the previous one ends. It can sometimes mean that one or two readable logs aren't created correctly, but nothing major. Running another tournament after waiting about 10 seconds generally works.

```
Exception in thread "Thread-6" java.lang.NullPointerException
    at com.orchestral.shark.output.ReadableLogger.consume(ReadableLogger.java:52)
    at com.orchestral.shark.output.JsonEventPublisher.lambda$0(JsonEventPublisher.java:39)
    at java.util.ArrayList.forEach(ArrayList.java:1249)
    at com.orchestral.shark.output.JsonEventPublisher.publish(JsonEventPublisher.java:39)
    at com.orchestral.shark.tournament.manager.TournamentManager$1.consume(TournamentManager.java:50)
    at com.orchestral.shark.output.JsonEventPublisher.lambda$0(JsonEventPublisher.java:39)
    at java.util.ArrayList.forEach(ArrayList.java:1249)
    at com.orchestral.shark.output.JsonEventPublisher.publish(JsonEventPublisher.java:39)
    at com.orchestral.shark.tournament.manager.GameManager.setup(GameManager.java:125)
```