Working with Coup

The rule engine

Having played Coup, the representation of the rules (https://github.com/connor-taylorbrown/coup/blob/master/src/main/resources/ruleset) should not be too confusing. As Coup has a wide variety of extensions (and the copyright holders are yet to comment on this project), I have chosen not to hardcode the rules but load them from this file at runtime. RulesetParser always uses this file as its source. This class has a simple interface - take a look at its public methods and see if you can write a main method printing the name of each rule and each card in the deck.

Actions

An action needs to be applied to the player, who might lose or receive coins, and an optional target, who might lose coins or influence. It cannot be immediately executed, however - blocking or challenging may prevent that altogether. There is an ideal design pattern for this problem - the Command Pattern. The Command Pattern allows you to express a request with a Command object (Action), specify recipients (Player), and hand it to an invoking object to execute when possible (also Player). Take a look at these two classes, then look at TransitiveAction and see if you can expand on your main method - get one player to steal off a target.

Turns

I hope you didn't jump ahead and try adding block or challenge to the fray. As you know, a turn might consist of a series of actions, all of which must be executed in reverse order - challenge prevents block, block prevents steal, hopefully you can guess the result. We can think of the turn, being invoked by a player and executed, as a composite action. According to the Composite Pattern, a series of component actions can be added, while the turn maintains the same interface as these actions (setPlayer(Player player), execute()). Have a look at Turn and get your target to block that steal.

A full game

Hopefully the design is clear and intuitive enough for you to go ahead and implement your own command line Coup. If you haven't done so already, make a new package and subclass Game and Player to begin. It might help (for inspiration, no copy) to look at the 'textui' package as well.
One final note - you might have noticed something odd about ChallengeAction compared to the other Actions. It's because a challenge relies on the target picking a card - when we're working on the server, reveal() won't wait for the client to reply. As such, ChallengeAction has to observe Player for changes, and processes the challenge only when the data is updated. If you're still confused, look at SimplePlayer's implementation of reveal().

Building a server

Communicating with a server

When a client wants to update the server or ask it for information, basic HTTP methods such as GET and POST will suffice. REST is not a protocol - it is a set of principles for designing scalable, simple and efficient Web services. It is important for us, building a Coup service, to remember that a RESTful service is client-server and stateless. This means everything the server needs to service a request is contained within the request - there are no sessions for storing client state. Stateless does *not* mean the server maintains no state, only that, from its perspective, all clients are the same. Keep these constraints in mind when thinking about the service in future steps - for now we will focus on turning a Java class into a RESTful service with Spring.

Before completing **this tutorial** - https://spring.io/guides/gs/rest-service/ - note that this project uses Maven, so don't waste time looking at Gradle or IDE instructions.

Push notifications

When another player joins the game, or the game wants to prompt a player to take their turn or choose a card, our server needs to be able to send messages to the client without first receiving a request. The alternative is expensive polling - before the introduction of Websockets with HTML 5, this made HTML an impractical tool for game development. With Websockets, a client need only subscribe to a channel - when a message appears, a callback will be invoked, perhaps prompting the player to choose their move. As usual, Spring makes this easy on the server side - an annotated method will send its result to the channel, with only minimal configuration.

Complete **this tutorial** - https://spring.io/guides/gs/messaging-stomp-websocket/

Storing state and joining games

Think of a multiplayer game like a group chat. Facebook manages a lot of group chats. All of them have the same interface - send a message, subscribe to the response channel. Yet sending a message does not notify every single Facebook user (thank God) - each chat retains its separate state and broadcasts on a separate channel. A game of Coup should be able to do the same.

The last section should give you some idea of how these channels might operate, but we still have no way of hosting multiple games. Think about it - say we call "/addPlayer" with NetworkedGame as a service, there's only one game instance available on the server. We can resolve this problem by storing several game instances in a repository.

Complete **this tutorial** - https://spring.io/guides/gs/accessing-data-jpa/

One last point

We know how to broadcast a message when a client sends a request, but @SendTo only works in this situation. It is quite simple to send unsolicited alerts, so don't let this distract you.