Sure, I'd be happy to explain my design and code.

My design consists of three main classes: Player, TeamImpl, and Main (the Driver class).

The Player class represents a player on a soccer team, with attributes such as their first and last name, date of birth, position, and jersey number.

The TeamImpl class represents a soccer team, with a list of players and methods for adding players, assigning jersey numbers, and setting the starting lineup.

The Main class is the driver class that initializes a team with a list of players, adds a new player, and outputs the starting lineup.

Main class show that the team is created successfully, a new player is added successfully, and the starting lineup is outputted correctly.

The Main class initializes a team with a list of players, adds a new player, and outputs the starting lineup. It uses try-catch blocks to handle exceptions thrown by the TeamImpl class.

The Main class initializes the team by creating a list of players and passing it to the TeamImpl constructor. It checks if the size of the list is between 10 and 20. If it is not, an IllegalArgumentException is thrown.

In terms of error handling, I added some basic checks to the TeamImpl class such as enforcing a minimum age requirement for new players. This provides some safeguards against common errors that can occur when managing a soccer team. However, more advanced error handling techniques such as exception handling could be added to improve the robustness of the code, such as checking same date of birth and same first/last name.

As for my code, I wrote unit tests for each class to ensure that they are working as expected.

The unit tests for the Player class test the constructor and getter methods.

The unit tests for the TeamImpl class test the addPlayer method, the assignJerseyNumbers method, and the setStartingLineup method.

Overall, my design and code are aimed at creating a simple, easy-to-use system for managing a soccer team. I hope that this explanation helps to clarify my thought process and design decisions.

///

Q: why enum?

One significant design decision I made was to use an enum for the Position attribute in the Player class. This makes it easier to enforce type safety and avoids the use string that declares directly in several places in a code without assigning it to a variable.

Q:

in above design? why not use array? why list? tradeoff?

- I chose to use a List instead of an array for storing the players in the TeamImpl class

- Flexibility: A List allows for dynamic resizing, whereas an array has a fixed size. This means that we can add and remove players from the team without having to worry about resizing the array. In addition, we can easily iterate over the list using a for-each loop or an iterator.

- Convenience: A List provides convenient methods for adding, removing, and accessing elements, such as add(), remove(), get(), and size(). This makes it easier to manipulate the list of players in the team.

- Readability: When using an array, we have to manually track the size of the array using a separate variable. This can make the code more difficult to read and understand. In contrast, a List encapsulates this information, making the code more readable and easier to understand.

Of course, there are tradeoffs to using a List as well. One of the main tradeoffs is that a List requires more memory overhead than an array, due to the additional pointers and metadata needed to manage the list. In addition, accessing elements in a List can be slightly slower than accessing elements in an array, due to the additional indirection required. However, these tradeoffs are generally small and are outweighed by the benefits of using a List in most cases.

Q: in above design, if you have one extra week, what would you do differently? improve?

If I had one extra week to work on this design, there are a few things I would consider improving:

- Adding more functionality to the TeamImpl class: Currently, the TeamImpl class has methods for adding players, assigning jersey numbers, and setting the starting lineup. However, there are many other things that could be added to make the class more robust, such as methods for substituting players during a game, calculating team statistics, or generating a schedule for upcoming games.

- Improving error handling: While I did add some error handling to the TeamImpl class, there are certainly more edge cases that could be handled. For example, what if a player has same lastname and firstname? what if they have same lastname and firstname and date of birth?

- Adding persistence: Currently, the system only exists in memory, which means that if the program is shut down, all data will be lost. By adding some form of persistence (such as a database or file system), we can ensure that data is retained even if the program is restarted.

- Improving unit test coverage: While I did write unit tests for each class, there are likely more test cases that could be added to ensure that the system is functioning correctly. By increasing test coverage, we can reduce the likelihood of bugs and improve the overall quality of the code.

Overall, while the current design is functional, there is certainly room for improvement. With an extra week, I would focus on adding more functionality, improving error handling, adding persistence, and increasing test coverage.