The given code does not really make use of key features object oriented programming (OOP). In this report, some methods to modularise the code will be examined, as well as restructuring the code to take advantage of OOP features.

Firstly, encapsulation is a feature that could be used for this game. To begin, the code should be split between three classes: myconnectfour.java, player.java, and board.java. The main class will be myconnectfour.java. Each of these classes will have variables that are private, meaning they are accessible only to that class. This is good coding practice due to the fact that they cannot be unintentionally accessed or changed by another class during the execution of the program. If they do need to be changed, methods called getters and setters will be used. These methods are public methods, so we can, from our main class call something such as board.setBoard() if it needs to be changed, for example when placing a counter.

The use of these classes also helps with modularising the code. In the given java file, there is a large while loop and a lot of code is repeated for each player. By having a player class with methods for these, we can create instances of player, for example playerR and playerY and simply call the relevant method from the player class. For example, we could call playerR.checkHorizontal() and later on call playerY.checkHorizontal(). This helps readability, and is far easier to code, as no redundant lines of code are required.

Due to the simplicity of this game, inheritance and polymorphism are not required. The only time one class is used to make two different objects is for the two players, however, as the functionality of these players is so similar, it is not necessary for us to use inheritance (for example class bot extends player) or polymorphism. Instead, as the bot can simply be a random number generator, it is sufficient to just create a method in the main class to generate a random number, and have the bot operate like: playerY.placeCounter(random()). If a more complex bot were required, it would make sense to use inheritance, as it would share the methods and variables from the player class, but also require many more methods, or just one very complex method to define the logic it uses to place the counter.

Finally, abstract classes will not be necessary to re-write this program. This is because we need to create objects for board and the players to be used in the main class. As such, by definition, none of the classes we have defined can be abstract.