AP ACTIVITY **4.2.8**

Using an Abstract Board Class (AP)

INTRODUCTION

Think of all the different types of boards a game can use. Now, imagine that you have all of those boards in a cabinet in your house, and someone asks you to hand them the game board. You would probably hesitate and ask some follow up questions, such as, “which board?” Or perhaps, “what kind of board?” There is not really a single game board that works for all games; you need a specific game board for a specific game.

In an object-oriented world, you might think that all game boards have things in common, like the game boards for checkers and chess. Common features of related items (game boards for similar games) is the idea behind **abstract classes**. The Elevens game belongs to a set of related solitaire games that use the same board with different rules. In this activity, you will learn about some games related to Elevens. Then, you will see how **inheritance** can be used to reuse the code that is common to all of these games without rewriting it.

abstract class
A class that cannot be instantiated and defines instance fields and methods for subclasses; differs from an interface mainly in that it can contain method implementations.

inherit or inheritance
When a subclass created automatically gains all the methods and fields of its superclass or parent class.

Materials

- Computer with BlueJ IDE

Procedure

Thirteens and Tens are two of the games related to Elevens.

Thirteens

- Thirteens uses a 10-card board. Ace, 2, … , 10, jack, queen correspond to the point values of 1, 2, …, 10, 11, 12.
- Pairs of cards whose point values add up to 13 are selected and removed.
- Kings are selected and removed singly.
- Chances of winning are claimed to be about 1 out of 2.

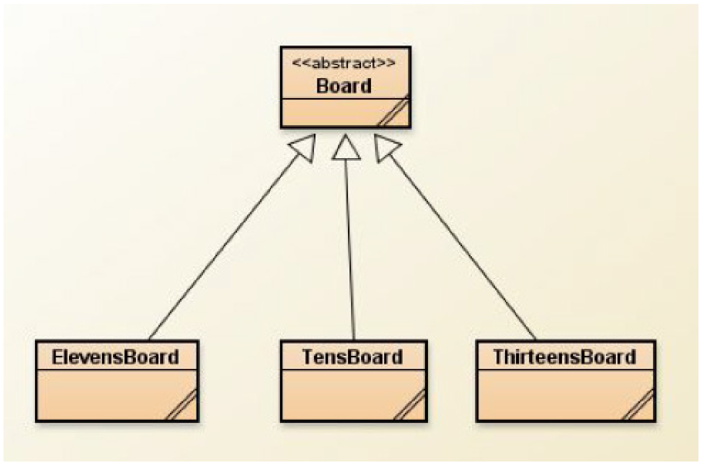
Tens

- Tens, uses a 13-card board.
- Pairs of cards whose point values add to 10 are selected and removed, as are quartets of kings, queens, jacks, and tens, all of the same rank (for example, K♠, K♥, K♦, and K♣).
- Chances of winning are claimed to be about 1 in 8 games.

It is evident that these games share common state and behaviors to Elevens. All three games require:

- State (instance variables) — a deck of cards and the cards "on the" board.
- Behavior (methods) — to deal the cards, to remove and replace selected cards, to check for a win, to check if selected cards satisfy the rules of the game, to see if there are more legal selections available, and so on.

With the state and behavior in common, it would seem that inheritance could allow us to write code once and reuse it instead of having to copy it for each game.



The common state and behavior is what the abstract Board class contains. Take all the state and behavior that these boards have in common and put them into a new `Board` class. Then, the `ElevensBoard`, `TensBoard`, and `ThirteensBoard` inherit from the `Board` class. This makes sense because each of them is just a different kind of board. An `ElevensBoard` "IS-A" `Board`, a `ThirteensBoard` "IS-A" `Board`, and a

`TensBoard` “IS-A” `Board`. Examine the diagram that shows the inheritance relationships of these classes.

How does this works for dividing up our original `ElevensBoard` code from AP Activity 4.2.7?

- All three games need a deck and the cards on the board, so all of those instance variables can go into `Board`.
- Some methods, like `deal`, will work the same for every game, so they should be in `Board` too.
- Methods like `containsJQK` are Elevens-specific and should be in `ElevensBoard`.

But what should be done with the `isLegal` and `anotherPlayIsPossible` methods?

Every Elevens-related game will have both of these methods, but they need to work differently for each game. That’s exactly why Java has abstract methods.

- Because each of these games needs `isLegal` and `anotherPlayIsPossible` methods, include those methods in `Board`.
- However, the implementation of these methods depends on the specific game, so you make them `abstract` in `Board` and don’t include their implementations there.
- Also, because `Board` now contains abstract methods, it must be specified as `abstract`.
- Finally, override each of these abstract methods in the subclasses to implement their specific behavior for that game.

But if `isLegal` and `anotherPlayIsPossible` are implemented in each game-specific board class, why do you need to have the abstract methods in `Board`?

Consider a class that uses a board, such as the UI program you used in AP Activity 4.2.6. Such a class is called a client of the Board class. The client takes care of specific game details, which means the UI program does not actually need to know what kind of a game it is displaying. It only needs to know that the game board “IS-A” Board, and it only knows about the methods in the `Board` class. The UI program is able to call `isLegal` and `anotherPlayIsPossible` only because they are included in `Board`.

How is the UI program able to execute the correct `isLegal` and `anotherPlayIsPossible` methods?

When the UI program starts, it is provided an object of a class that inherits from `Board`. If you want to play Elevens, you provide an `ElevensBoard` object. If you want to play Tens, you provide a `TensBoard` object. When the UI program uses that object to call `isLegal` or `anotherPlayIsPossible`, it automatically uses the method implementation included in that particular object. This is known as **polymorphism**.

polymorphism
The ability of an object to take on multiple data types, all related by the super/sub class relationship.

1. Discuss the similarities and differences between Elevens, Thirteens, and Tens.

2. As discussed previously, all of the instance variables are declared in the `Board` class, but it is the `ElevensBoard` class that knows the board size, and the ranks, suits, and point values of the cards in the deck. How do the `Board` instance variables get initialized with the ElevensBoard values? What is the exact mechanism?

3. Now, examine the files `Board.java` and `ElevensBoard.java` in the ElevensActivity8 provided by your teacher. Identify the abstract methods in `Board.java`. See how these methods are implemented in `ElevensBoard`. Do they cover all the differences between Elevens, Thirteens, and Tens as discussed in question 1? Why or why not?

CONCLUSION

1. How are the Concentration `Board` class and the Elevens `Board` class similar? Describe similarities in terms of state and behavior.