```
/**
 * @file Rules.js
 * @description This file contains rules of BigTwo game.
 * @author Jiaxin Tang
 * @version Latest edition on April 11, 2021
 */

const suitsPath = ["Diamonds", "Clubs", "Hearts", "Spades"]
const valuesPath = ["", "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"]
const suits = ["D", "C", "H", "S"]
const SuiteVal = [1, 2, 3, 4]
const type = ["", "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]

/**
 * @function newDeck
 * @description A function that generates a deck of 52 cards, and rearranges the order of cards in the deck
 * @returns {card[]} deck - with cards in random order
 */
export function newDeck() {
    let deck = []

    for (let i = 1; i < 14; i++) {
        for (let j = 0; j < 4; j++) {
            let value = (i === 1) ? 14 : (i === 2) ? 15 : i
            let imagePath = "NAP-01_"+ suitsPath[j] + "_"+ valuesPath[i] + ".png"
            let card = {
                type: type[i],
                suit: suits[j],
                suiteVal : SuiteVal[j],
                value: value,
                imagePath: imagePath
            }
            deck.push(card)
        }
    }

    return shuffle(deck)
}

/**
 * @function shuffle
 * @description A function that rearranges the order of cards in the given deck
 * @param {card[]} deck - a list of cards
 * @returns {card[]} deck - with cards in random order
 */

function shuffle(deck) {
    var temp, i, j;
    for (i = deck.length - 1; i > 0; i--) {
        j = Math.floor(Math.random() * (i + 1));
        temp = deck[i];
        deck[i] = deck[j];
        deck[j] = temp;
    }
    return deck;
}
```

```
57
58 /**
59  * @function isValidStartingPlay
60  * @description A function that checks if the current play is valid starting play
61  * @param {card[]} cards - the cards that current player has
62  * @returns {boolean} = true if cards contain Diamond 3
63  */
64 export function isValidStartingPlay(cards) {
65     let containsThreeOfDiamonds
66
67     cards.forEach((card) => {
68         if (card.suit === "D" && card.value === 3) containsThreeOfDiamonds = true
69     })
70
71     if (containsThreeOfDiamonds) {
72         return isValidPlay(cards)
73     } else {
74         return false
75     }
76 }
77
78 /**
79  * @function isValidSPlay
80  * @description A function that checks if the current play is valid play
81  * @param {card[]} cards - the cards that current player selects
82  * @returns {boolean} - true if is valid play
83  */
84
85 export function isValidPlay(cards) {
86     if (cards == null) return false
87     sortCardsValue(cards)
88
89     return isValidSingle(cards) || isValidPair(cards) ||
isValidFiveCardPlay(cards)
90 }
91
92
93 /**
94  * @function isValidSingle
95  * @description A function that checks if the current play is valid single play
96  * @param {card[]} cards - the cards that current player selects
97  * @returns {boolean} - true if cards contain a single card
98  */
99 export function isValidSingle(cards) {
100     return cards.length === 1
101 }
102
103 /**
104  * @function isValidPair
105  * @description A function that checks if the current play is valid pair
106  * @param {card[]} cards - the cards that current player selects
107  * @returns {boolean} - true if cards is a valid pair
108  */
109 export function isValidPair(cards) {
110     return cards.length === 2 && cards[0].type === cards[1].type
111 }
112
113 /**
114  * @function isValidFiveCardPlay
```

```
 * @description A function that checks if the current play is valid five card play
 * @param {card[]} cards - the cards that current player selects
 * @returns {boolean} - true if cards is a valid combination of five cards
 */
export function isValidFiveCardPlay(cards) {
    if (cards.length !== 5) return false

    return isValidStraight(cards) || isValidFlush(cards) || isValidFullHouse(cards) || isValidFourOfaKind(cards)
}

/**
 * @function isValidStraight
 * @description A function that checks if the current play is valid straight
 * @param {card[]} cards - the cards that current player selects
 * @returns {boolean} - true if cards is a valid straight
 */
function isValidStraight(cards) {
    if(cards.length !== 5)
        return false
    //12345
    sortCardsValue(cards)
    if(cards[0].value === 14){
        if(cards[1].value === 15 && cards[2].value === 3 &&
            cards[3].value === 4 && cards[4].value === 5 )
                return true
        else
            return false
    }
    //23456
    if(cards[0].value === 15){
        if(cards[1].value === 3 && cards[2].value === 4 &&
            cards[3].value === 5 && cards[4].value === 6 )
                return true
        else
            return false
    }
    var flag = true
    for(var i = 0; i < 4; i++){
        if((cards[i].value + 1) !== cards[i+1].value){
            flag = false
            return flag
        }
    }
    return flag
}

/**
 * @function isValidFlush
 * @description A function that checks if the current play is valid flush
 * @param {card[]} cards - the cards that current player selects
 * @returns {boolean} - true if cards is a valid flush
 */
function isValidFlush(cards) {
    if(cards.length !== 5)
        return false
    var flag = true
    for(var i = 1; i < 5; i++){
```

```
172         if(cards[i].suiteVal !== cards[0].suiteVal){
173             flag = false
174             return flag
175         }
176     }
177     return flag
178 }
179 
180 /**
181  * @function isValidFullHouse
182  * @description A function that checks if the current play is valid fullhouse
183  * @param {card[]} cards - the cards that current player selects
184  * @returns {boolean} - true if cards is a valid fullhouse
185  */
186 function isValidFullHouse(cards) {
187     if(cards.length !== 5)
188         return false
189     sortCardsValue(cards)
190     if(cards[0].value === cards[1].value && cards[0].value === cards[2].value &&
191         cards[3].value === cards[4].value)
192         return true
193     if(cards[0].value === cards[1].value && cards[2].value === cards[3].value &&
194         cards[2].value === cards[4].value)
195         return true
196     return false}
197 
198 /**
199  * @function isValidFourOfaKind
200  * @description A function that checks if the current play is valid four of a
    kind
201  * @param {card[]} cards - the cards that current player selects
202  * @returns {boolean} - true if cards is a valid four of a kind
203  */
204 function isValidFourOfaKind(cards) {
205      if(cards.length !== 5)
206         return false
207     sortCardsValue(cards)
208     if(cards[0].value === cards[1].value && cards[0].value === cards[2].value &&
209         cards[0].value === cards[3].value)
210         return true
211     if(cards[4].value === cards[1].value && cards[4].value === cards[2].value &&
212         cards[4].value === cards[3].value)
213         return true
214     return false
215 }
216 
217 /**
218  * @function isStrongerPlay
219  * @description A function that checks if the current play is stronger than last
    play
220  * @param {card[]} last - the cards that the last player plays
221  * @param {card[]} select - the cards that current player selects
222  * @returns {boolean} - true if the select play is stronger than last play
223  */
224 export function isStrongerPlay(last, select) {
225     var n = select.length
226     if(n !== last.length)
227         return false
228     switch(n){
```

```
229         case 1: return isStrongerSingle(last, select);
230         case 2: return isStrongerPair(last, select);
231         case 5: return isStrongerFive(last, select);
232         default:
233             return false
234     }
235 }
236 
237 /**
238  * @function isStrongerSingle
239  * @description A function that checks if the current single is stronger than
    last single
240  * @param {card[]} last - the cards that the last player plays
241  * @param {card[]} select - the cards that current player selects
242  * @returns {boolean} - true if the select play is stronger than last play
243  */
244 export function isStrongerSingle(last, select){
245     if(select[0] && last[0]){
246         if(select[0].value > last[0].value)
247             return true
248         if(select[0].value === last[0].value && select[0].suiteVal >
    last[0].suiteVal)
249             return true
250     }else if(select[0] && !last[0]){
251         if(select[0].value > last.value)
252             return true
253         if(select[0].value === last.value && select[0].suiteVal > last.suiteVal)
254             return true
255     }else if(!select[0] && !last[0]){
256         if(select.value > last.value)
257             return true
258         if(select.value === last.value && select.suiteVal > last.suiteVal)
259             return true
260     }else if(!select[0] && last[0]){
261         if(select.value > last[0].value)
262             return true
263         if(select.value === last[0].value && select.suiteVal > last[0].suiteVal)
264             return true
265     }
266     return false
267 }
268 
269 /**
270  * @function isStrongerPair
271  * @description A function that checks if the current pair is stronger than last
    pair
272  * @param {card[]} last - the cards that the last player plays
273  * @param {card[]} select - the cards that current player selects
274  * @returns {boolean} - true if the select play is stronger than last play
275  */
276 export function isStrongerPair(last, select){
277     if(!isValidPair(select))
278         return false
279     if(select[0].value > last[0].value)
280         return true
281     sortCardsSuit(select)
282     sortCardsSuit(last)
283     if(select[0].value === last[0].value && select[1].suiteVal >
    last[1].suiteVal)
```

```
284         return true
285     return false
286 }
287 
288 /**
289  * @function isStrongerFive
290  * @description A function that checks if the current five card play is stronger
than last five card play
291  * @param {card[]} last - the cards that the last player plays
292  * @param {card[]} select - the cards that current player selects
293  * @returns {boolean} - true if the select play is stronger than last play
294  */
295 export function isStrongerFive(last, select){
296     if(isValidFourOfaKind(select) && isValidFullHouse(last))
297         return true
298     if(isValidFourOfaKind(select) && isValidFlush(last))
299         return true
300     if(isValidFourOfaKind(select) && isValidStraight(last))
301         return true
302     if(isValidFullHouse(select) && isValidFlush(last))
303         return true
304     if(isValidFullHouse(select) && isValidStraight(last))
305         return true
306     if(isValidFlush(select) && isValidStraight(last))
307         return true
308     if(isValidStraight(select) && isValidStraight(last)){
309         sortCardsValue(select)
310         sortCardsValue(last)
311         if(select[4].value > last[4].value)
312             return true
313         else
314             return false
315     }
316     if(isValidFlush(select) && isValidFlush(last)){
317         sortCardsValue(select)
318         sortCardsValue(last)
319         if(select[0].suiteVal > last[0].suiteVal)
320             return true
321         if(select[0].suiteVal === last[0].suiteVal && select[4].value >
last[4].value)
322             return true
323         return false
324     }
325     if(isValidFullHouse(select) && isValidFullHouse(last)){
326         sortCardsValue(select)
327         sortCardsValue(last)
328         if(select[3].value > last[3].value)
329             return true
330         return false
331     }
332     if(isValidFourOfaKind(select) && isValidFourOfaKind(last)){
333         sortCardsValue(select)
334         sortCardsValue(last)
335         if(select[3].value > last[3].value)
336             return true
337         return false
338     }
339 }
340 
```

```
/**
 * @function setUserCards
 * @description A function that places 13 cards in a deck into a list to be assigned to a player.
 * @param {card[]} deck - a list of 52 cards in a random order
 * @returns {card[]} userCards - contains 13 cards for a player
 */
export function setUserCards(deck) {
    let userCards = []
    for (let i = 0; i < 13; i++) {
        userCards.push(deck.pop())
    }
    return userCards
}

/**
 * @function setFirstTurn
 * @description A function that decides which player plays the first turn.
 * @param {card[]} playerCards - a list of cards that player has
 * @param {card[]} opponentLeftCards - a list of cards that left AI has
 * @param {card[]} opponentTopCards - a list of cards that top AI has
 * @param {card[]} opponentRightCards - a list of cards that right AI has
 * @returns {string} turn - represeting the initial player
 */
export function setFirstTurn(playerCards, opponentLeftCards, opponentTopCards, opponentRightCards) {
    let turn
    playerCards.forEach((card) => {
        if (card.suit === "D" && card.value === 3) turn = "player"
    })

    opponentLeftCards.forEach((card) => {
        if (card.suit === "D" && card.value === 3) turn = "left"
    })

    opponentTopCards.forEach((card) => {
        if (card.suit === "D" && card.value === 3) turn = "top"
    })

    opponentRightCards.forEach((card) => {
        if (card.suit === "D" && card.value === 3) turn = "right"
    })
    return turn
}

/**
 * @function getSuitValue
 * @description A function that gets the integer value of the corresponding suit.
 * @param {string} suit
 * @returns {int} - integer value related to suit
 */
export function getSuitValue(suit) {
    return (suit === "D") ? 1 : (suit === "C") ? 2 : (suit === "H") ? 3 : 4
}

/**
 * @function sortCardsValue
 * @description A function that sorts the given cards in the number rank order.
 * @param {card[]} cards
```

```
 * @returns {card[]} cards - ordered in the number rank
 */
export function sortCardsValue(cards) {
    if (cards == null) return

    cards.sort((a, b) => {
        return a.value - b.value
    })
}

/**
 * @function sortCardsSuit
 * @description A function that sorts the given cards in the suit rank order.
 * @param {card[]} cards
 * @returns {card[]} cards - ordered in the suit rank
 */
export function sortCardsSuit(cards) {
    if (cards == null) return

    cards.sort((a, b) => {
        return a.suiteVal - b.suiteVal
    })
}
```