Project 1: Blackjack Report

Abstract

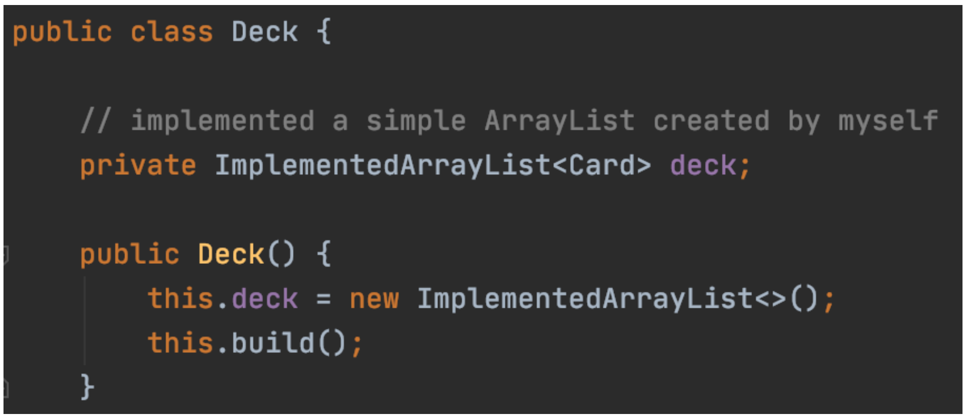
This week we were given the first project written in Java, called Blackjack, with intensive use of Object-Oriented Programming. I modeled the Blackjack game by creating various classes that represent objects used in the game including `Card`, `Deck`, `Hand`, and `Blackjack`, mainly using `ArrayList` (a list-like data structure that allows dynamic memory allocation) and `HashMap` (a dictionary-like data structure that maps a key to a value, used for storing game result). I eventually coded a `Simulation` class that repeats the game for more than thousands times, and it is interesting to find out that the dealer usually has a bigger winning chances than the player.

Task 1 - Card.java

I had to model each object in a real life blackjack game, and the card is the first one I start with. The `Card` class has one attribute, its `value`. It is used commonly in other classes such as `Deck` and `Hand`.

Task 2 - Deck.java

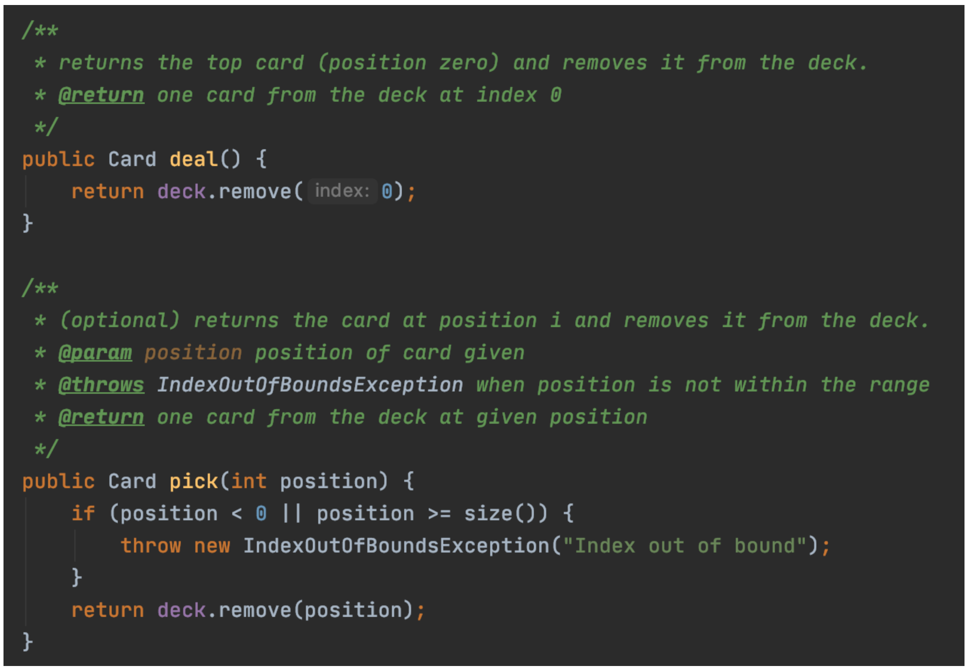
The deck class is trickier, because it needs to store multiple `Card` instances in a data structure whose length can vary during program runtime. This is due to that fact that the deck will often deal cards to `player` and `dealer`, i.e. the size of the deck will decrease as the game goes on. Thus, I first used Java `ArrayList` since it has the dynamic memory allocation feature, and later after I implemented my own ArrayList, I used my own as the attribute of the `Deck` class.



```java
public class Deck {

    // implemented a simple ArrayList created by myself
    private ImplementedArrayList<Card> deck;

    public Deck() {
        this.deck = new ImplementedArrayList<>();
        this.build();
    }
```

The Deck class also has several interesting methods as follow.

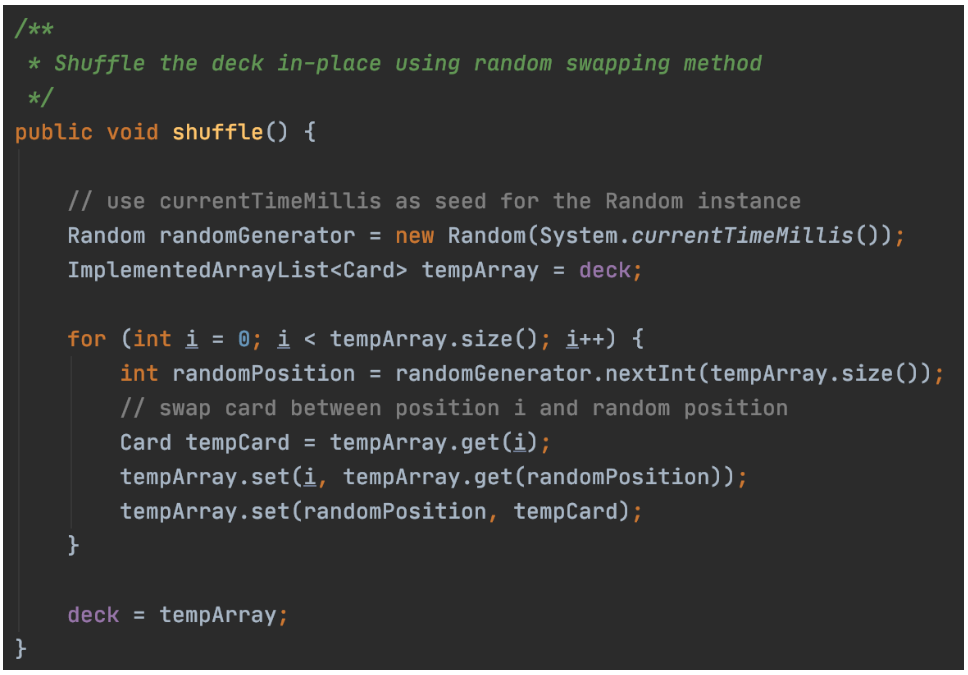
1. `build()`: This method builds an new deck of card, consisting 4 cards of each value from 1 to 9, and 16 cards of 10.
2. `deal()` and `pick()`: These two methods will return one card from the deck, the `deal()` method removes the top card (index is 0) from the deck and return it, while the `pick()` method removes any card whose index is within the deck range.



```java
/**
 * returns the top card (position zero) and removes it from the deck.
 * @return one card from the deck at index 0
 */
public Card deal() {
    return deck.remove( index: 0);
}

/**
 * (optional) returns the card at position i and removes it from the deck.
 * @param position position of card given
 * @throws IndexOutOfBoundsException when position is not within the range
 * @return one card from the deck at given position
 */
public Card pick(int position) {
    if (position < 0 || position >= size()) {
        throw new IndexOutOfBoundsException("Index out of bound");
    }
    return deck.remove(position);
}
```

3. `shuffle()`: This is the method I implemented for shuffling the deck of cards. I didn't use the one taught in lab. The idea of this shuffle algorithm

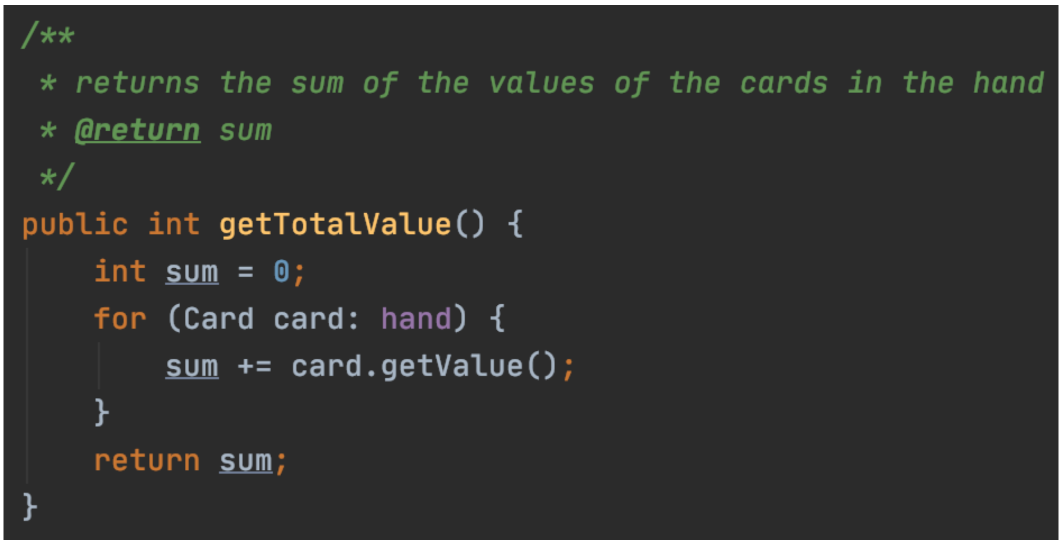
is to iterate through all the items in the array and swap the item with another item of random index (generated by a `Random` instance). It makes sure every item got swapped with a random item, and thus shuffle the entire array.



```java
/**
 * Shuffle the deck in-place using random swapping method
 */
public void shuffle() {

    // use currentTimeMillis as seed for the Random instance
    Random randomGenerator = new Random(System.currentTimeMillis());
    ImplementedArrayList<Card> tempArray = deck;

    for (int i = 0; i < tempArray.size(); i++) {
        int randomPosition = randomGenerator.nextInt(tempArray.size());
        // swap card between position i and random position
        Card tempCard = tempArray.get(i);
        tempArray.set(i, tempArray.get(randomPosition));
        tempArray.set(randomPosition, tempCard);
    }

    deck = tempArray;
}
```

Task 3 - Hand.java

This class was to model player's and dealer's hand. The hand needs to contain a set of cards, and I in this case used my ImplementedArrayList again to store the Cards that every hand possesses. The hand object has several methods, one of which is critical here called `getTotalValue()`. This method returns the total value of all the cards in the hand, and I used it commonly because I frequently need to compare the total value of one hand with that of another.

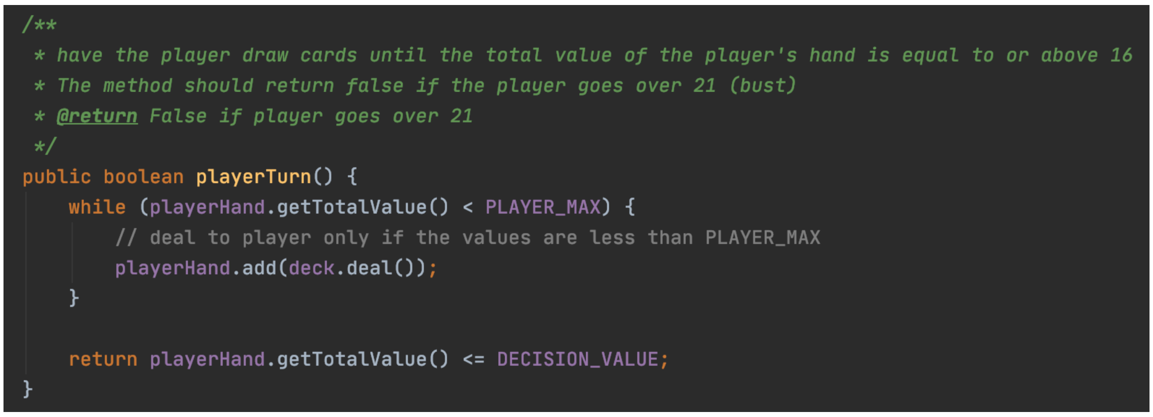


```java
/**
 * returns the sum of the values of the cards in the hand
 * @return sum
 */
public int getTotalValue() {
    int sum = 0;
    for (Card card: hand) {
        sum += card.getValue();
    }
    return sum;
}
```

Task 4 - Blackjack.java

This class was the most important one, as it models the entire game, integrating all the classes written above. It had several import methods among others, as well:

1. `deal()`: This method is used at the start of the game. It deals two cards to player and the dealer.
2. playerTurn() and `dealerTurn()`: These two methods return `false` when it's the end of their turn. I had a little problem understanding how they end their turn and what else these methods would return. But after several hand-written simulations, I figured it out.



```java
/**
 * have the player draw cards until the total value of the player's hand is equal to or above 16
 * The method should return false if the player goes over 21 (bust)
 * @return False if player goes over 21
 */
public boolean playerTurn() {
    while (playerHand.getTotalValue() < PLAYER_MAX) {
        // deal to player only if the values are less than PLAYER_MAX
        playerHand.add(deck.deal());
    }

    return playerHand.getTotalValue() <= DECISION_VALUE;
}
```

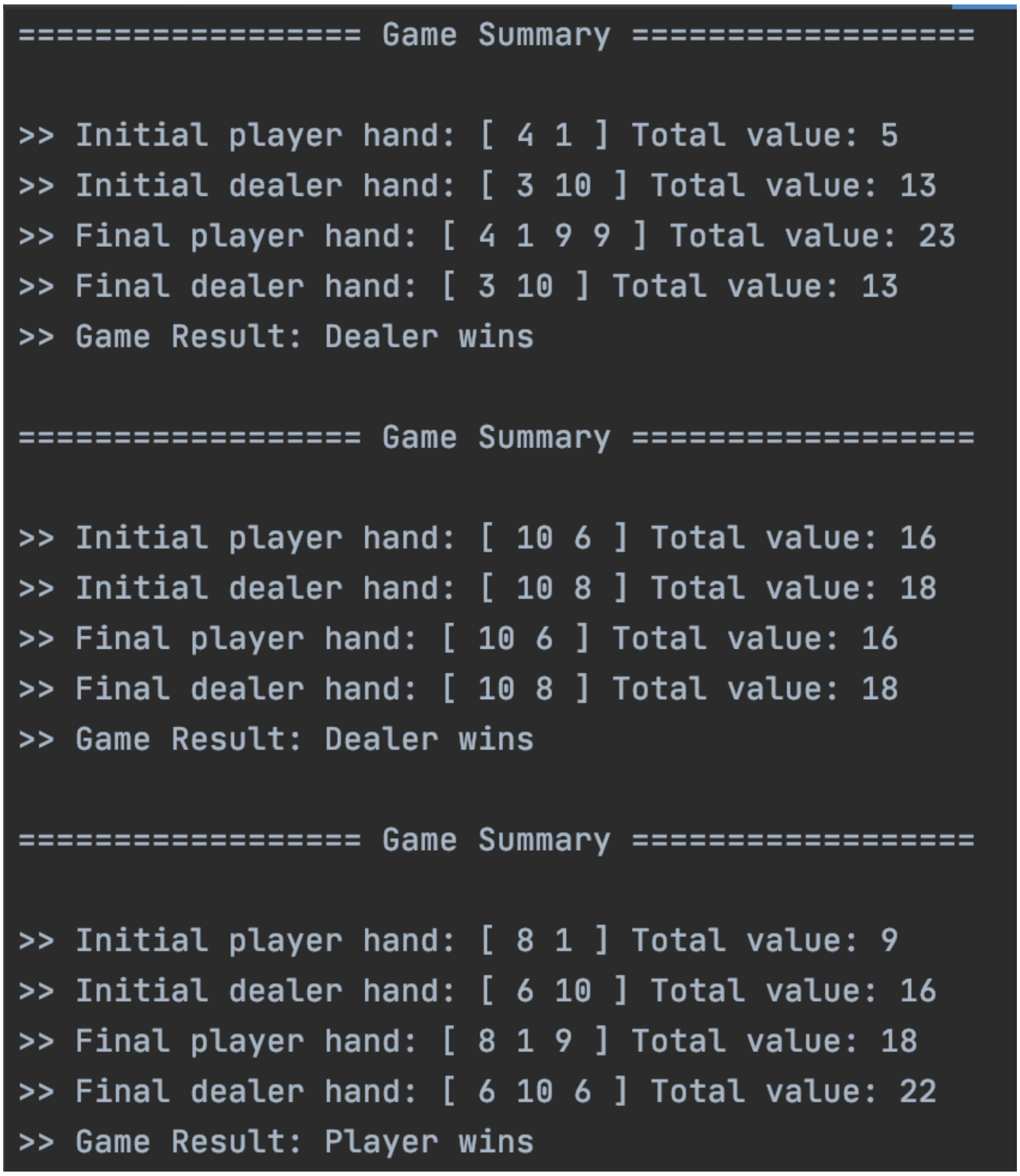
3. reset(): This method resets everything the player and dealer has on their hand. It will also reset the Deck when the size of the remaining deck is lower than the reShuffleCutoff value.

```java
/**
 * reset the game
 *
 * Both the player Hand and dealer Hand should start with no cards. If the number of
 * cards in the deck is less than the reshuffle cutoff, then the method should create a fresh (complete),
 * shuffled deck. Otherwise, it should not modify the deck, just clear the player and dealer hands
 */
public void reset() {
    playerHand.reset();
    dealerHand.reset();

    // refill and shuffle the deck if size is lower than CUT_OFF value
    if (deck.size() < CUT_OFF) {
        deck = new Deck();
        deck.shuffle();
    }
}
```

4. `game()`: This models the entire game, and it returns either -1,0,1 to represent dealer wins, pushed, or player wins. I had little problem figuring out the logic of comparison at first, but I overcame it by writing down a few simulations at last. The code below is the comparison logic.

```java
if (playerTurn()) {
    if (dealerTurn()) {
        if (playerHand.getTotalValue() > dealerHand.getTotalValue()) {
            status = "Player wins";
            result = 1;
        } else if (playerHand.getTotalValue() == dealerHand.getTotalValue()) {
            status = "It's a push";
            result = 0;
        } else {
            status = "Dealer wins";
            result = -1;
        }

    } else {
        status = "Player wins";
        result = 1;
    }
} else {
    status = "Dealer wins";
    result = -1;
}
```

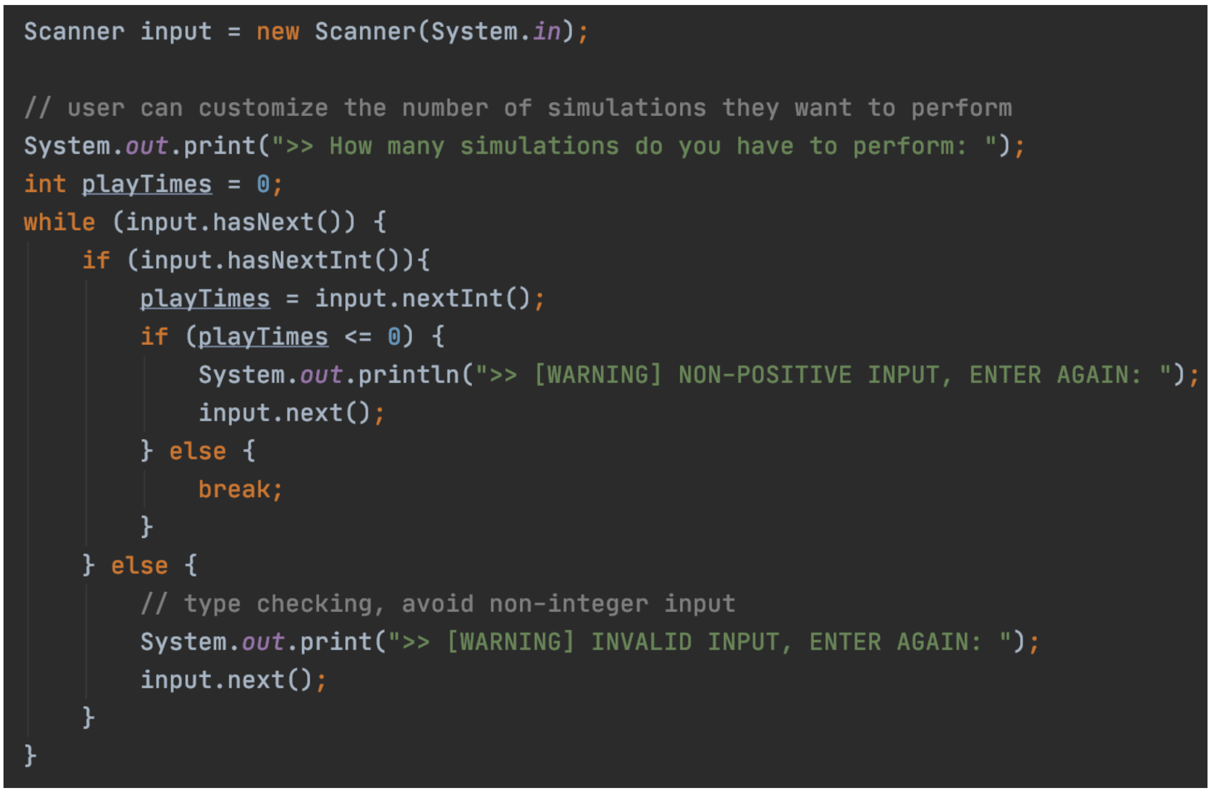
5. `main()`: I also used Blackjack to perform three games, and print out the game summary.



```
================== Game Summary ==================

>> Initial player hand: [ 4 1 ] Total value: 5
>> Initial dealer hand: [ 3 10 ] Total value: 13
>> Final player hand: [ 4 1 9 9 ] Total value: 23
>> Final dealer hand: [ 3 10 ] Total value: 13
>> Game Result: Dealer wins

================== Game Summary ==================

>> Initial player hand: [ 10 6 ] Total value: 16
>> Initial dealer hand: [ 10 8 ] Total value: 18
>> Final player hand: [ 10 6 ] Total value: 16
>> Final dealer hand: [ 10 8 ] Total value: 18
>> Game Result: Dealer wins

================== Game Summary ==================

>> Initial player hand: [ 8 1 ] Total value: 9
>> Initial dealer hand: [ 6 10 ] Total value: 16
>> Final player hand: [ 8 1 9 ] Total value: 18
>> Final dealer hand: [ 6 10 6 ] Total value: 22
>> Game Result: Player wins
```

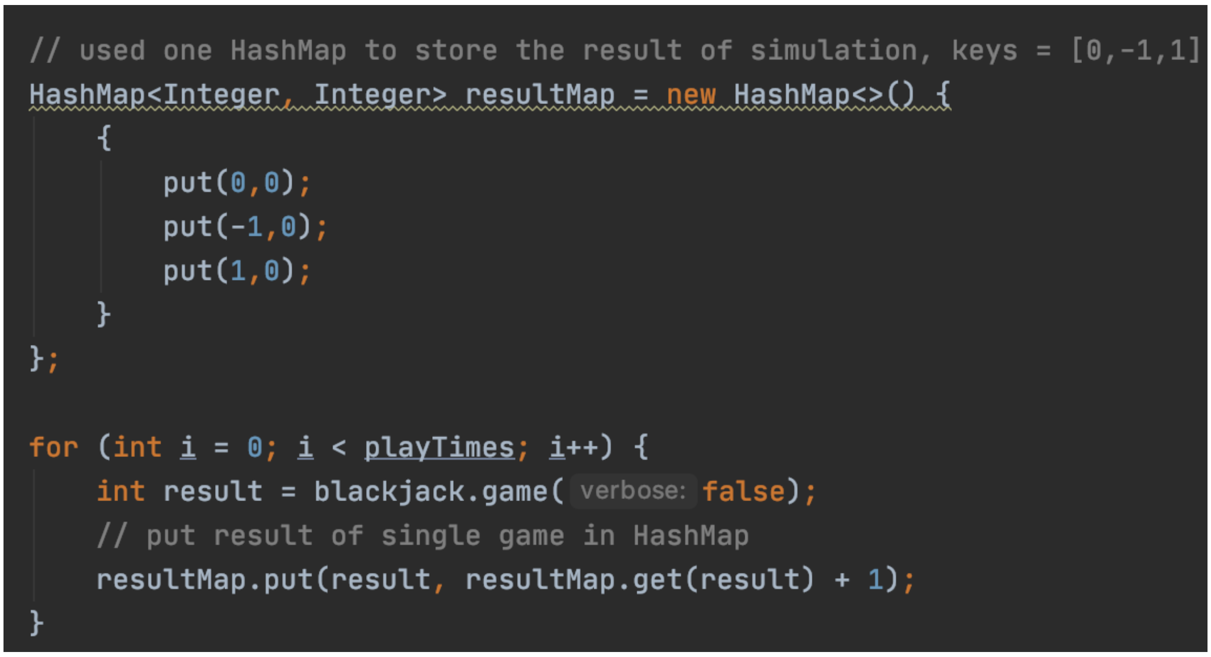
Task 5 - Simulation.java

When running this class, the user will be prompted to enter the number of simulations he/she wants to perform. I used a Scanner to achieve user input from the terminal. To avoid non-integer input and non-positive integer input, I also wrote an algorithm to do input checking. See below.



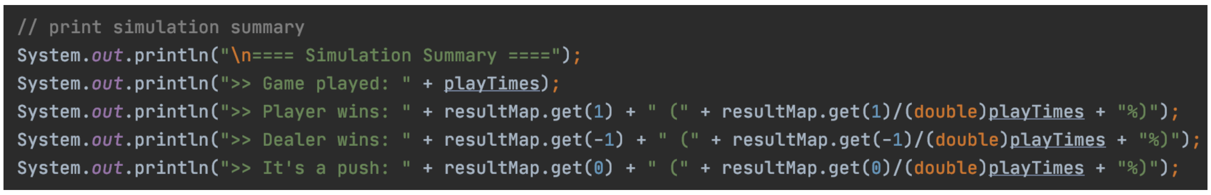
```java
Scanner input = new Scanner(System.in);

// user can customize the number of simulations they want to perform
System.out.print(">> How many simulations do you have to perform: ");
int playTimes = 0;
while (input.hasNext()) {
    if (input.hasNextInt()){
        playTimes = input.nextInt();
        if (playTimes <= 0) {
            System.out.println(">> [WARNING] NON-POSITIVE INPUT, ENTER AGAIN: ");
            input.next();
        } else {
            break;
        }
    } else {
        // type checking, avoid non-integer input
        System.out.print(">> [WARNING] INVALID INPUT, ENTER AGAIN: ");
        input.next();
    }
}
```

To store the result of one single game, I used a dictionary-like structure called `HashMap`. It updates the number of three cases when a new game result is returned.

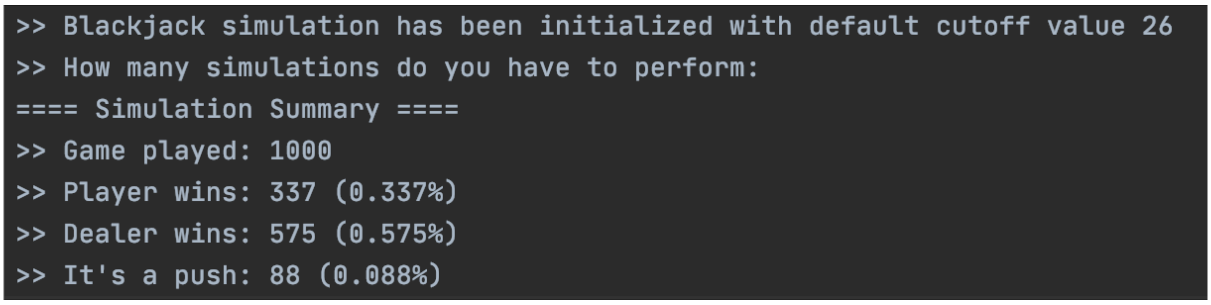


```java
// used one HashMap to store the result of simulation, keys = [0,-1,1]
HashMap<Integer, Integer> resultMap = new HashMap<>() {
    {
        put(0,0);
        put(-1,0);
        put(1,0);
    }
};

for (int i = 0; i < playTimes; i++) {
    int result = blackjack.game( verbose: false);
    // put result of single game in HashMap
    resultMap.put(result, resultMap.get(result) + 1);
}
```

The simulation will print the result at the console, containing number of dealer wins, player wins, and pushes, as well as the percentage.

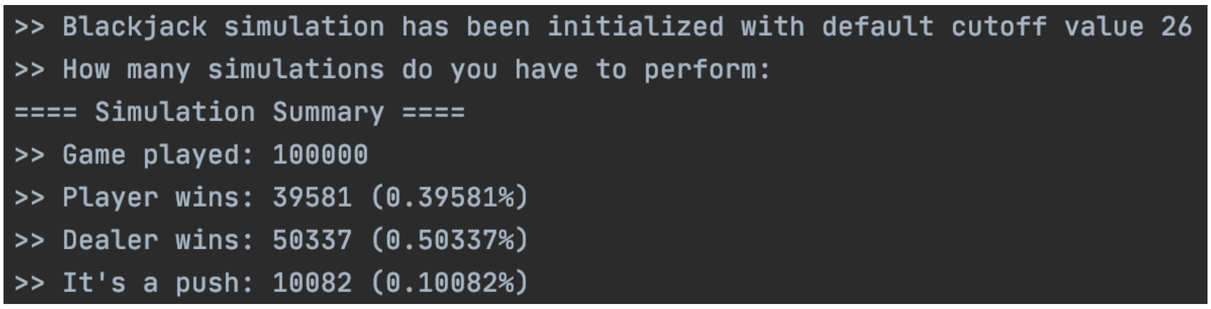


```java
// print simulation summary
System.out.println("\n==== Simulation Summary ====");
System.out.println(">> Game played: " + playTimes);
System.out.println(">> Player wins: " + resultMap.get(1) + " (" + resultMap.get(1)/(double)playTimes + "%)");
System.out.println(">> Dealer wins: " + resultMap.get(-1) + " (" + resultMap.get(-1)/(double)playTimes + "%)");
System.out.println(">> It's a push: " + resultMap.get(0) + " (" + resultMap.get(0)/(double)playTimes + "%)");
```



```
>> Blackjack simulation has been initialized with default cutoff value 26
>> How many simulations do you have to perform:
==== Simulation Summary ====
>> Game played: 1000
>> Player wins: 337 (0.337%)
>> Dealer wins: 575 (0.575%)
>> It's a push: 88 (0.088%)
```

When user enters 100000 simulations, here is the result:



```
>> Blackjack simulation has been initialized with default cutoff value 26
>> How many simulations do you have to perform:
==== Simulation Summary ====
>> Game played: 100000
>> Player wins: 39581 (0.39581%)
>> Dealer wins: 50337 (0.50337%)
>> It's a push: 10082 (0.10082%)
```

Results

As we can see from above, the Dealer has much a higher percentage of winning in both simulation. As you can see in the extension below, it seems that the dealer wins more in many cases, varied by different decision values and game playtimes.

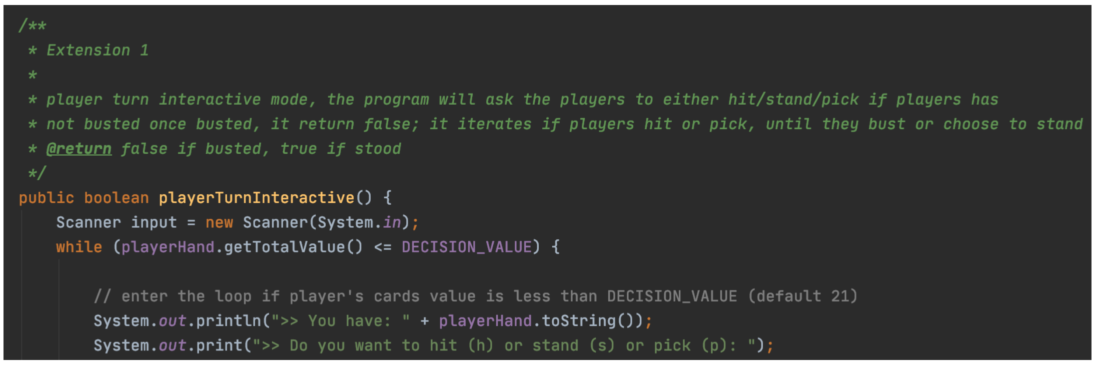
Extensions

Extension 1 - Interactive Mode

I designed the interactive mode that allows the user to be the player in the Blackjack game. I only need to add two methods in the Blackjack class, the `playerTurnInteractive()` and `gameInteractive()`, adapted from `playerTurn()` and `game()`.

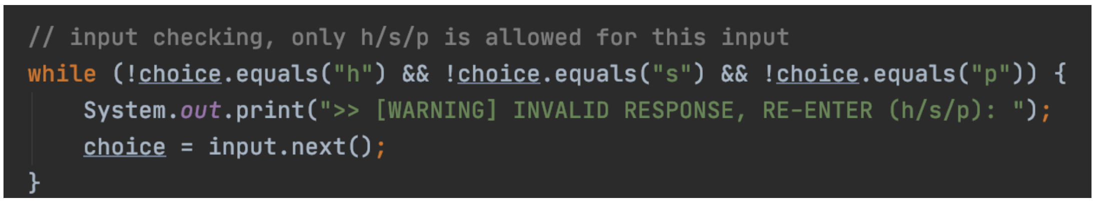
1. `playerTurnInteractive()`:

1. In this method, they will first be given the cards content and their total value on the terminal. Next they will be asked to choose hit, stand, or pick a card from the deck.



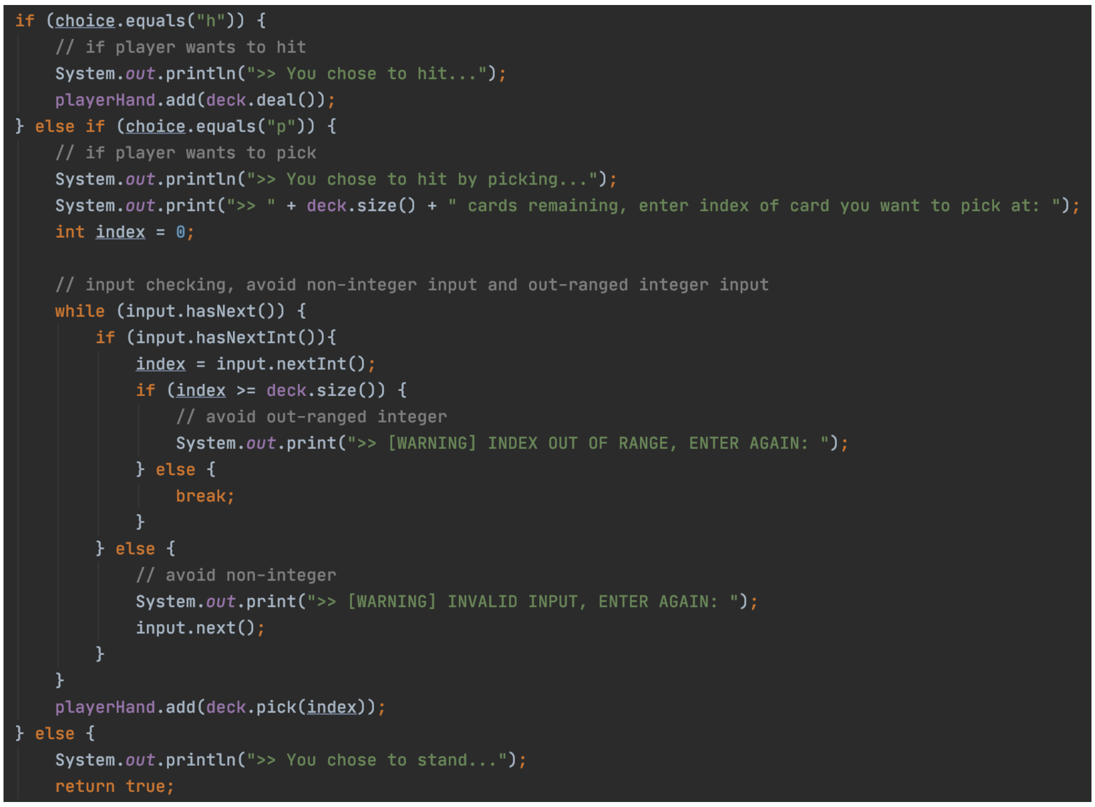
```java
/**
 * Extension 1
 *
 * player turn interactive mode, the program will ask the players to either hit/stand/pick if players has
 * not busted once busted, it return false; it iterates if players hit or pick, until they bust or choose to stand
 * @return false if busted, true if stood
 */
public boolean playerTurnInteractive() {
    Scanner input = new Scanner(System.in);
    while (playerHand.getTotalValue() <= DECISION_VALUE) {

        // enter the loop if player's cards value is less than DECISION_VALUE (default 21)
        System.out.println(">> You have: " + playerHand.toString());
        System.out.print(">> Do you want to hit (h) or stand (s) or pick (p): ");
```

2. Then, the code will check whether the input is allowed. Only `h/s/p` are allowed, as they represent hit, stand, and pick. If the user enter something else, they will receive a warning message and are asked to re-enter.



```java
// input checking, only h/s/p is allowed for this input
while (!choice.equals("h") && !choice.equals("s") && !choice.equals("p")) {
    System.out.print(">> [WARNING] INVALID RESPONSE, RE-ENTER (h/s/p): ");
    choice = input.next();
}
```

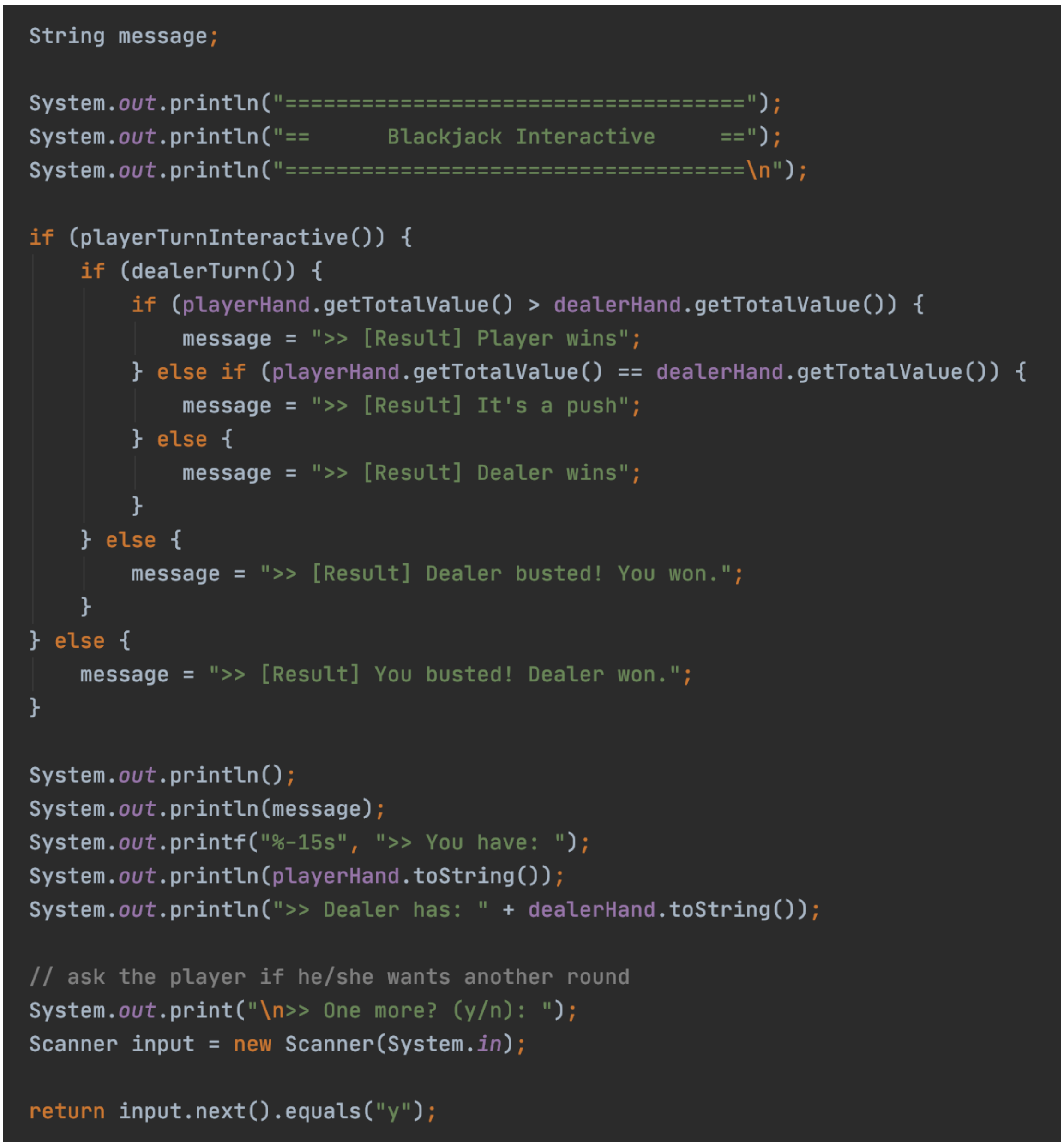
3. The the code will go into conditionals according to the user input. Here, particularly when user inputs "p" that stands for "pick", the program will first show the size of remaining cards, then ask user to enter a index at which the user wants to pick from the deck. It will also perform range and type checking here, until the program reads a valid input. The program will return true, if the user chooses to stand, i.e. the user didn't bust and it's the dealer's turn.



```java
if (choice.equals("h")) {
    // if player wants to hit
    System.out.println(">> You chose to hit...");
    playerHand.add(deck.deal());
} else if (choice.equals("p")) {
    // if player wants to pick
    System.out.println(">> You chose to hit by picking...");
    System.out.print(">> " + deck.size() + " cards remaining, enter index of card you want to pick at: ");
    int index = 0;

    // input checking, avoid non-integer input and out-ranged integer input
    while (input.hasNext()) {
        if (input.hasNextInt()){
            index = input.nextInt();
            if (index >= deck.size()) {
                // avoid out-ranged integer
                System.out.print(">> [WARNING] INDEX OUT OF RANGE, ENTER AGAIN: ");
            } else {
                break;
            }
        } else {
            // avoid non-integer
            System.out.print(">> [WARNING] INVALID INPUT, ENTER AGAIN: ");
            input.next();
        }
    }
    playerHand.add(deck.pick(index));
} else {
    System.out.println(">> You chose to stand...");
    return true;
```

2. gameInteractive():

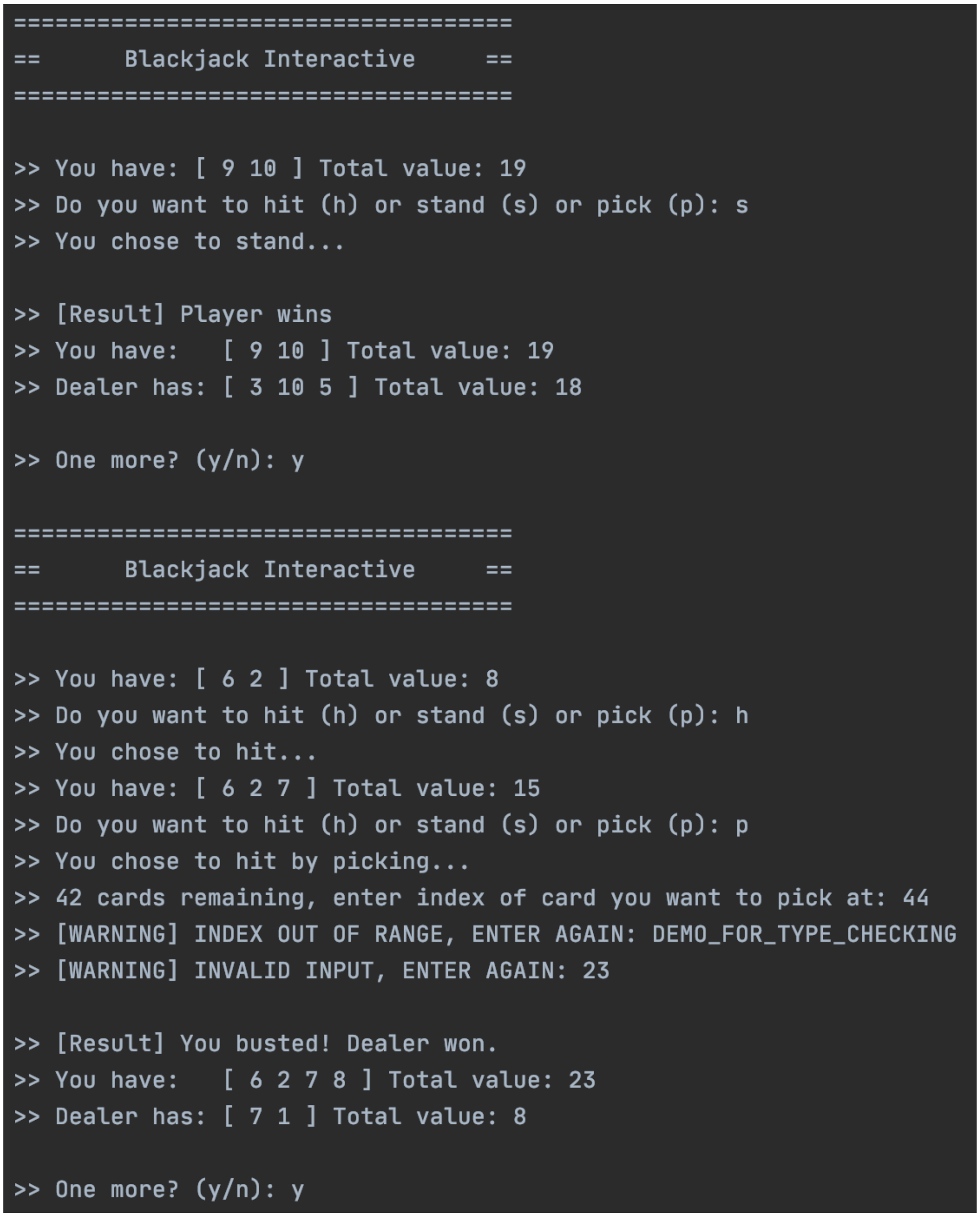
   While the other parts of the gameInteractive() method is similar to game(), there's one difference, which is the "One More Game" feature. It asks the user whether he wants to play for one more.



```java
String message;

System.out.println("====================================");
System.out.println("==      Blackjack Interactive     ==");
System.out.println("====================================\n");

if (playerTurnInteractive()) {
    if (dealerTurn()) {
        if (playerHand.getTotalValue() > dealerHand.getTotalValue()) {
            message = ">> [Result] Player wins";
        } else if (playerHand.getTotalValue() == dealerHand.getTotalValue()) {
            message = ">> [Result] It's a push";
        } else {
            message = ">> [Result] Dealer wins";
        }
    } else {
        message = ">> [Result] Dealer busted! You won.";
    }
} else {
    message = ">> [Result] You busted! Dealer won.";
}

System.out.println();
System.out.println(message);
System.out.printf("%-15s", ">> You have: ");
System.out.println(playerHand.toString());
System.out.println(">> Dealer has: " + dealerHand.toString());

// ask the player if he/she wants another round
System.out.print("\n>> One more? (y/n): ");
Scanner input = new Scanner(System.in);

return input.next().equals("y");
```

3. Demonstration:

   You can find below in InteractiveResult.txt:



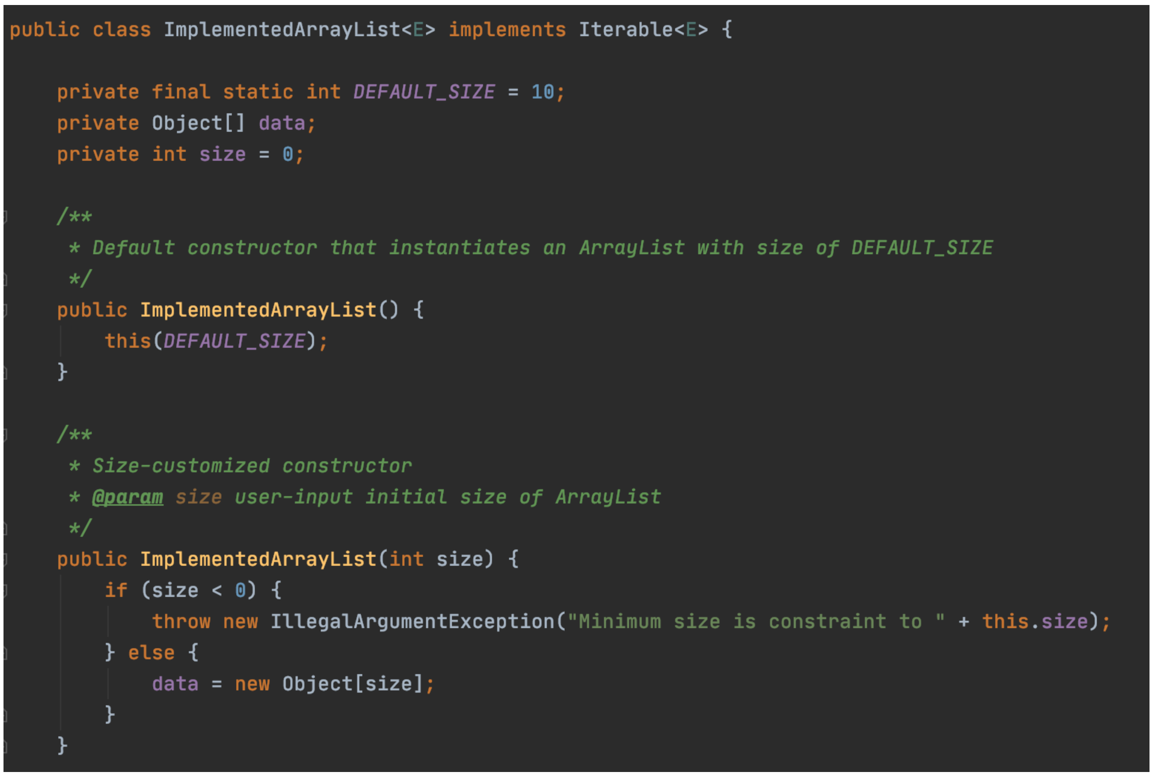
```
====================================
==      Blackjack Interactive     ==
====================================

>> You have: [ 9 10 ] Total value: 19
>> Do you want to hit (h) or stand (s) or pick (p): s
>> You chose to stand...

>> [Result] Player wins
>> You have:   [ 9 10 ] Total value: 19
>> Dealer has: [ 3 10 5 ] Total value: 18

>> One more? (y/n): y

====================================
==      Blackjack Interactive     ==
====================================

>> You have: [ 6 2 ] Total value: 8
>> Do you want to hit (h) or stand (s) or pick (p): h
>> You chose to hit...
>> You have: [ 6 2 7 ] Total value: 15
>> Do you want to hit (h) or stand (s) or pick (p): p
>> You chose to hit by picking...
>> 42 cards remaining, enter index of card you want to pick at: 44
>> [WARNING] INDEX OUT OF RANGE, ENTER AGAIN: DEMO_FOR_TYPE_CHECKING
>> [WARNING] INVALID INPUT, ENTER AGAIN: 23

>> [Result] You busted! Dealer won.
>> You have:   [ 6 2 7 8 ] Total value: 23
>> Dealer has: [ 7 1 ] Total value: 8

>> One more? (y/n): y
```

Extension 7 - ImplementedArrayList

In this project, I reviewed my understanding of ArrayList and got much more familiar with the Dynamic Memory Allocation feature, so I decided to implement this class on my own.

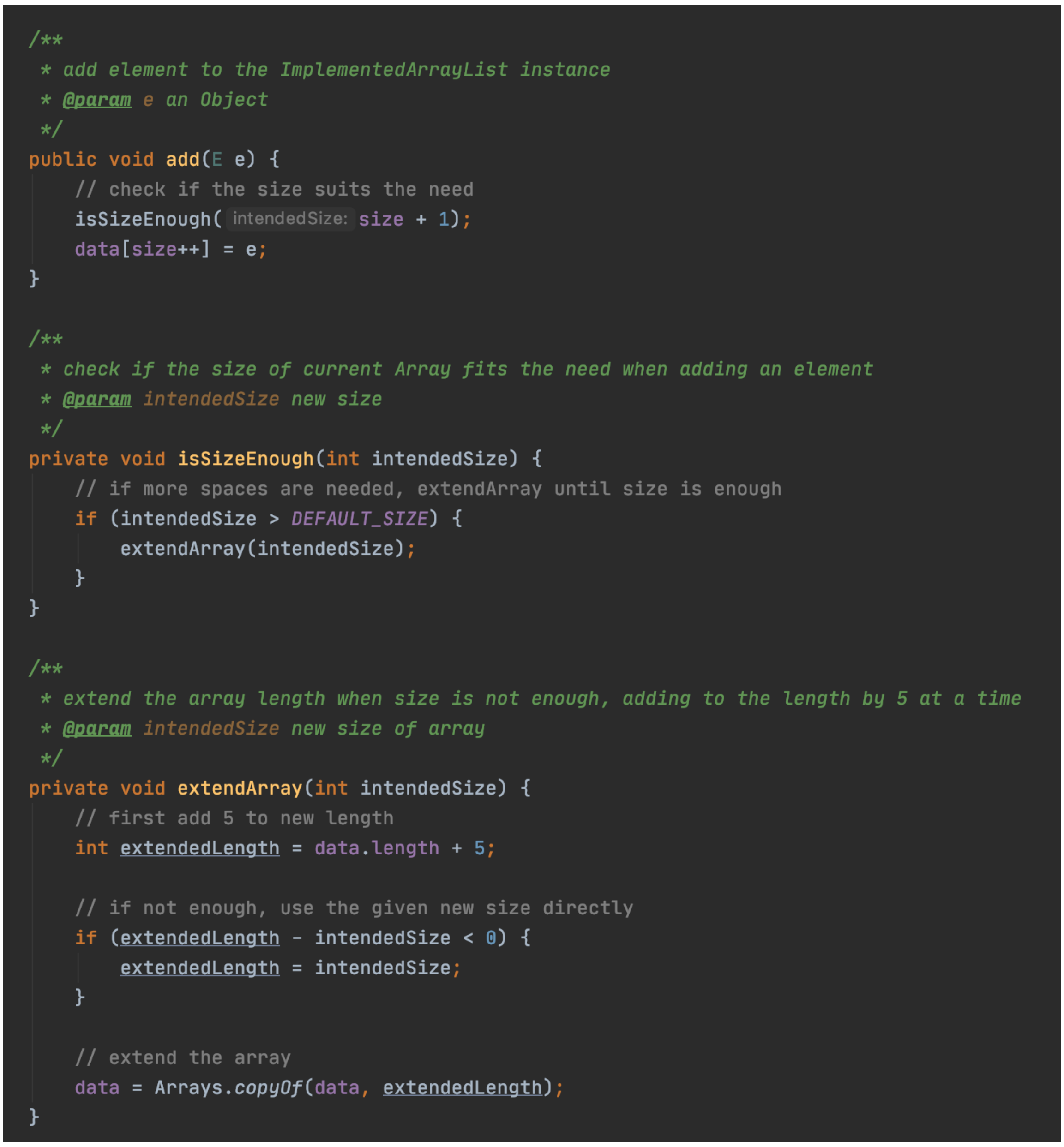
1. An ArrayList uses Java default Array to store elements. So I used data, a Object array in the attribute, and size, the size of array. The constructor

method has two overloaded. One method is to create a ArrayList of default size 10, and one is to create one array of customized size.



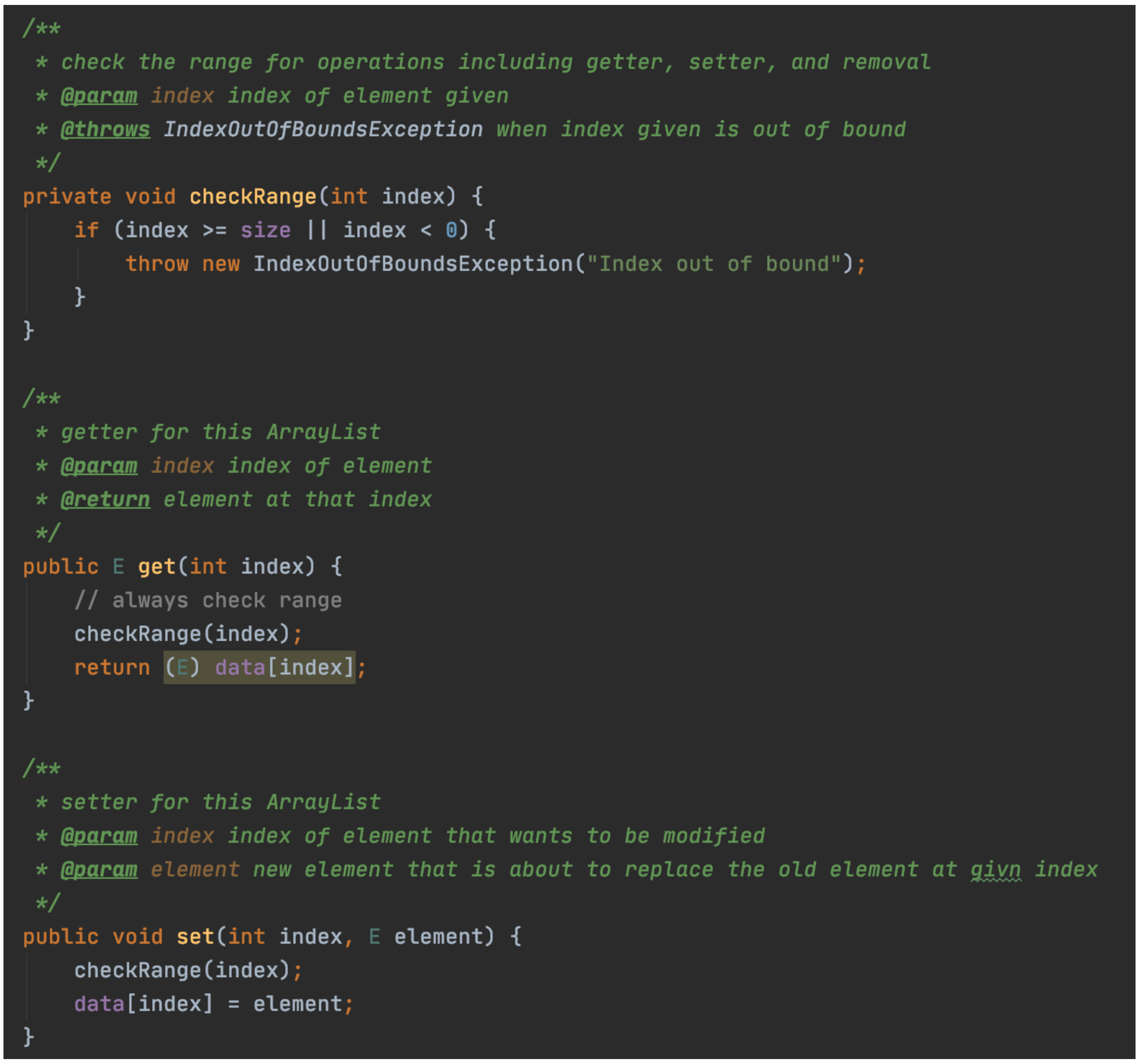
```java
public class ImplementedArrayList<E> implements Iterable<E> {

    private final static int DEFAULT_SIZE = 10;
    private Object[] data;
    private int size = 0;

    /**
     * Default constructor that instantiates an ArrayList with size of DEFAULT_SIZE
     */
    public ImplementedArrayList() {
        this(DEFAULT_SIZE);
    }

    /**
     * Size-customized constructor
     * @param size user-input initial size of ArrayList
     */
    public ImplementedArrayList(int size) {
        if (size < 0) {
            throw new IllegalArgumentException("Minimum size is constraint to " + this.size);
        } else {
            data = new Object[size];
        }
    }
```

2. The most crucial feature of `ArrayList` is dynamic memory allocation, and that happens when the user adds an element while the size of array doesn't allow one more element to add in. So in my implementation, I first check whether the `size` is enough for adding one more object, if not, I will use the `extendArray()` to extend the size of array. I do this by adding 5 more spaces each time, and I copy this extended array back to the attribute of this class, `data` . After checking the size and extending the array, I can then add the element to the last element, and also add the size of the array that waits for the next element to be added in.



```java
/**
 * add element to the ImplementedArrayList instance
 * @param e an Object
 */
public void add(E e) {
    // check if the size suits the need
    isSizeEnough( intendedSize: size + 1);
    data[size++] = e;
}

/**
 * check if the size of current Array fits the need when adding an element
 * @param intendedSize new size
 */
private void isSizeEnough(int intendedSize) {
    // if more spaces are needed, extendArray until size is enough
    if (intendedSize > DEFAULT_SIZE) {
        extendArray(intendedSize);
    }
}

/**
 * extend the array length when size is not enough, adding to the length by 5 at a time
 * @param intendedSize new size of array
 */
private void extendArray(int intendedSize) {
    // first add 5 to new length
    int extendedLength = data.length + 5;

    // if not enough, use the given new size directly
    if (extendedLength - intendedSize < 0) {
        extendedLength = intendedSize;
    }

    // extend the array
    data = Arrays.copyOf(data, extendedLength);
}
```

3. The `getter` and `setter` method of my `ArrayList` will all need to check whether the index input is within the range, so I wrote the `checkRange()` method here. If it's not within range, it will throw a new Exception indicating "Index out of Bound".

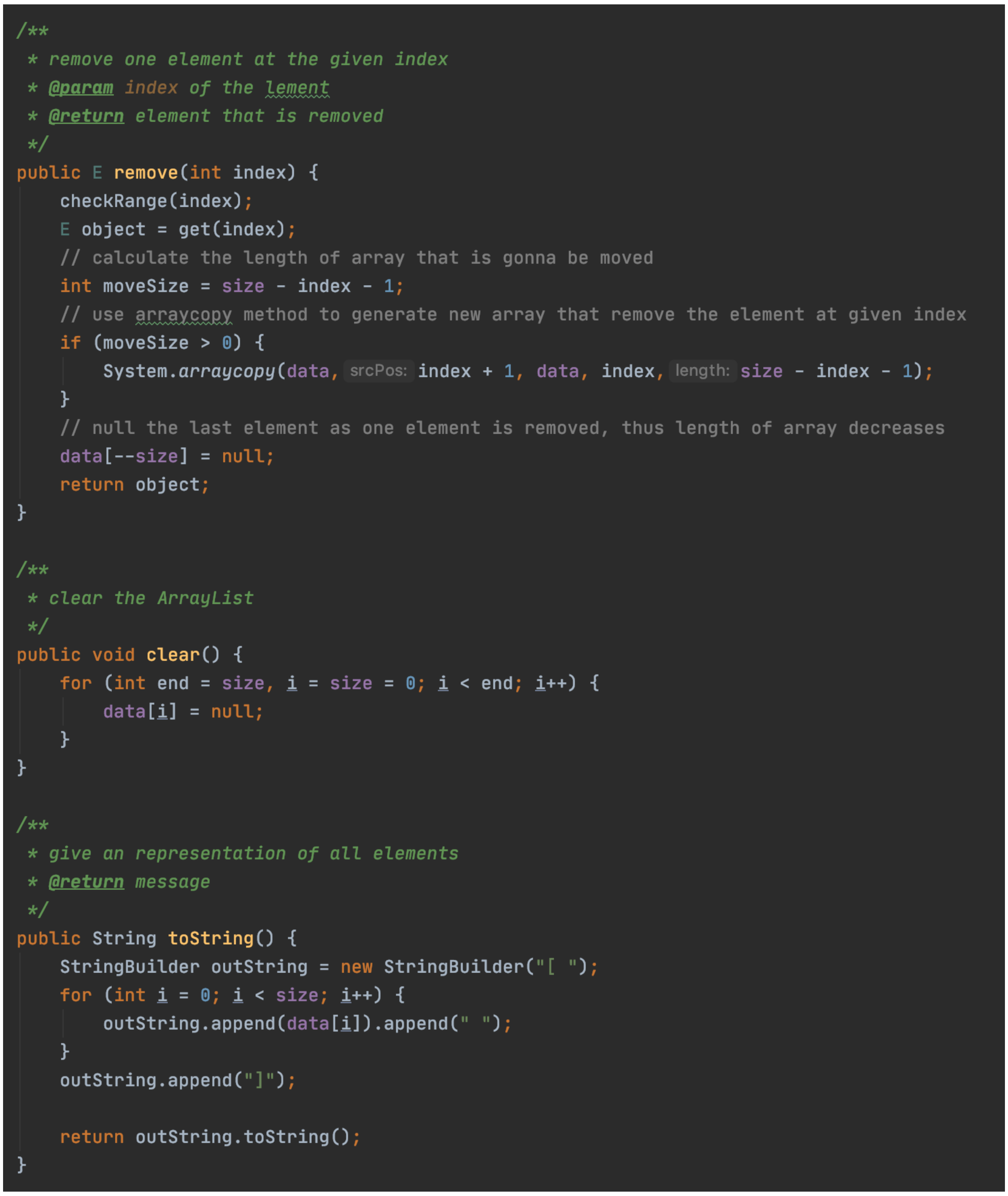


```java
/**
 * check the range for operations including getter, setter, and removal
 * @param index index of element given
 * @throws IndexOutOfBoundsException when index given is out of bound
 */
private void checkRange(int index) {
    if (index >= size || index < 0) {
        throw new IndexOutOfBoundsException("Index out of bound");
    }
}

/**
 * getter for this ArrayList
 * @param index index of element
 * @return element at that index
 */
public E get(int index) {
    // always check range
    checkRange(index);
    return (E) data[index];
}

/**
 * setter for this ArrayList
 * @param index index of element that wants to be modified
 * @param element new element that is about to replace the old element at givn index
 */
public void set(int index, E element) {
    checkRange(index);
    data[index] = element;
}
```

4. The `remove()` method here is also tricky. It first checks the range, whether the index input is within range, and it get the element at that index if `checkRange` is good. It then copy separate the array into two parts at the given input. It takes out the first half that is at the left of the removed element, and concatenates with the second half at its right, so the array now doesn't have the removed element. Since the size has changed, the size of the array needs to be decreased, and the last element will also be set to `null`.

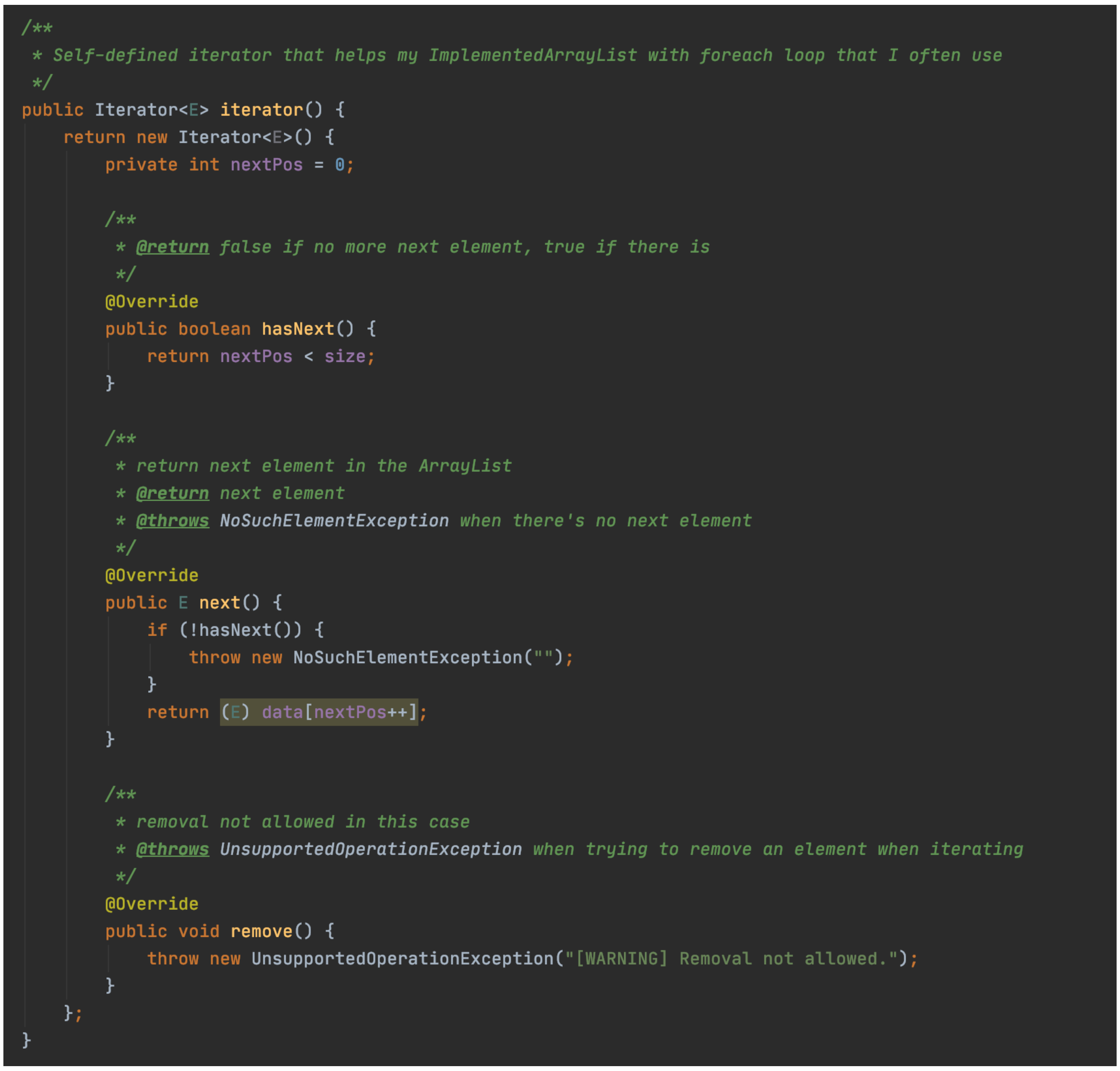
   The `clear()` method iterates through every single element and set it to `null`.

   The `toString()` method comes from how the default ArrayList prints itself. i.e. `[ 1 2 3 4 ]`.



```java
/**
 * remove one element at the given index
 * @param index of the lement
 * @return element that is removed
 */
public E remove(int index) {
    checkRange(index);
    E object = get(index);
    // calculate the length of array that is gonna be moved
    int moveSize = size - index - 1;
    // use arraycopy method to generate new array that remove the element at given index
    if (moveSize > 0) {
        System.arraycopy(data, srcPos: index + 1, data, index, length: size - index - 1);
    }
    // null the last element as one element is removed, thus length of array decreases
    data[--size] = null;
    return object;
}

/**
 * clear the ArrayList
 */
public void clear() {
    for (int end = size, i = size = 0; i < end; i++) {
        data[i] = null;
    }
}

/**
 * give an representation of all elements
 * @return message
 */
public String toString() {
    StringBuilder outString = new StringBuilder("[ ");
    for (int i = 0; i < size; i++) {
        outString.append(data[i]).append(" ");
    }
    outString.append("]");

    return outString.toString();
}
```

5. Iterator for this ArrayList. The `iterator()` here for this class will be used for `foreach` loop.



```java
/**
 * Self-defined iterator that helps my ImplementedArrayList with foreach loop that I often use
 */
public Iterator<E> iterator() {
    return new Iterator<E>() {
        private int nextPos = 0;

        /**
         * @return false if no more next element, true if there is
         */
        @Override
        public boolean hasNext() {
            return nextPos < size;
        }

        /**
         * return next element in the ArrayList
         * @return next element
         * @throws NoSuchElementException when there's no next element
         */
        @Override
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException("");
            }
            return (E) data[nextPos++];
        }

        /**
         * removal not allowed in this case
         * @throws UnsupportedOperationException when trying to remove an element when iterating
         */
        @Override
        public void remove() {
            throw new UnsupportedOperationException("[WARNING] Removal not allowed.");
        }
    };
}
```

Learning Outcomes

1. Implementation of ArrayList
2. Use of HashMap
3. Interactive game mode design
4. Learning the Iterator

Acknowledgement/Reference

1. Java Documents - ArrayList, Iterator.
2. I didn't ask for help from others.