UEPB	**UNIVERSIDADE ESTADUAL DA PARAÍBA**	
	DISCIPLINA:	Laboratório de Programação II
	PROFESSOR:	Klaudio Henrique Mascarenhas Medeiros
	ALUNOS:	Matheus Victor Feitosa Pereira e Vitor Emanuel dos Santos Vidal

DOCUMENTAÇÃO

PROJETO AVALIATIVO DA UNIDADE I

TEMA GERAL: JOGOS

TEMA ESPECÍFICO: UNO

Resumo: Este documento tem por objetivo explicar detalhadamente o código feito na linguagem de programação Java para o projeto avaliativo da primeira unidade da cadeira de Laboratório de Programação II. O Jogo escolhido neste caso foi o UNO, um jogo de cartas onde cada jogador joga uma carta por vez, e deve-se jogar uma carta de cor ou valor semelhante a carta jogada pelo jogador anterior, quando tiver apenas uma carta sobrando na sua mão, o jogador deve dizer UNO do contrário ele deve puxar mais duas cartas, ganha o jogador que ficar sem cartas na mão.

Neste documento explicaremos passo a passo o código para a criação deste jogo, explicando o motivo de termos tomado cada decisão no código para que o jogo fique melhor representado. Algumas funções podem não ser explicadas neste documento pois acreditamos que os comentários no código são suficientes para isto.

Código Completo disponível em: https://github.com/MattVictor/UNO

SUMARIO

BIBLIOTECAS UTILIZADAS

- **java.util.Scanner** – Para coletar entradas dos usuários.
- **java.util.ArrayList** – Para representação do deck principal de cartas e da mão de cada jogador.
- **java.util.Collections** – Para o método que embaralha as cartas no início do jogo.
- **java.io.IOException** – Para identificação de possíveis erros em blocos try e catch.
- **java.util.Random** – Para a implementação de métodos baseados em chance na classe Com.

ARQUIVOS E SUAS DESIGNAÇÕES

Todos arquivos estão dentro da pasta Uno, dentro dela, temos um Main na parte de fora, código onde o jogo deve ser executado, e quatro pastas, Card, Player, Game, Outro, em ordem:

- **Card** – Pacote referente as cartas, Como Card e ActionCards (bem como suas subclasses) e o Deck onde vão ser armazenadas todas as cartas para dar início ao jogo.
- **Player** – Pacote referente aos players, contém a classe abstrata Player e suas subclasses RealPlayer (Usada para instanciar jogadores reais) e Com (usada para instanciar o Computador, uma sequencia de ações pré-programadas para que algum jogador real possa jogar o jogo sozinho).
- **Game** – Pacote do jogo, contém toda a lógica para fazer o jogo funcionar, bem como suas verificações.
- **Outro** – Uma classe auxiliar para as demais classes, contém métodos referentes ao próprio Java, como uma maneira segura de conseguir as entradas, clear para limpar o console, gerador de strings para prints, etc.

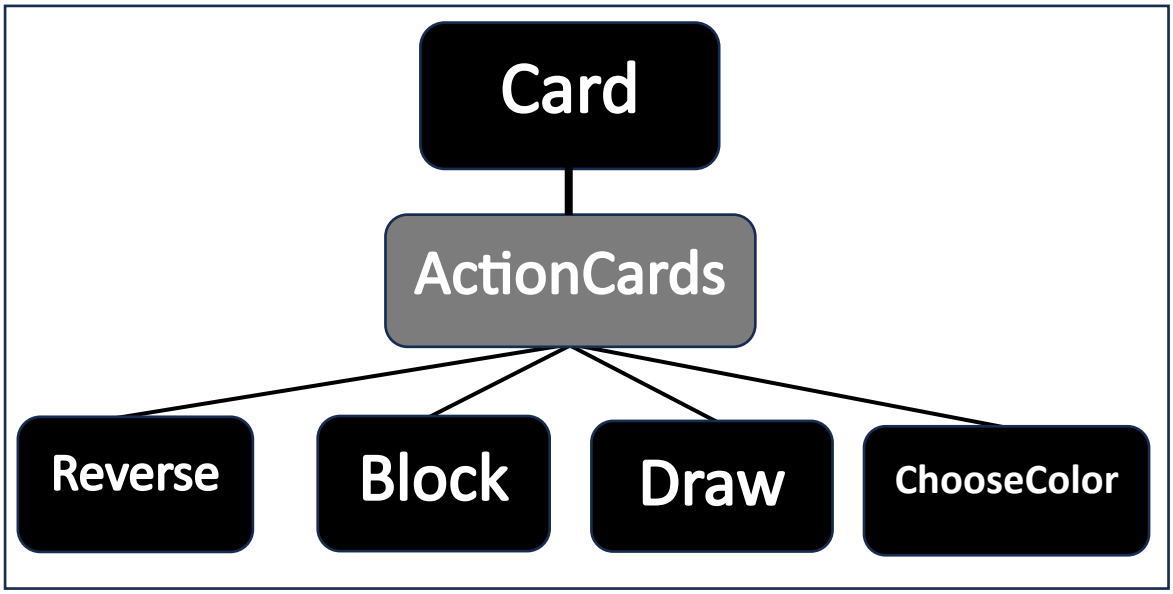
OBS: Caso algum dos métodos mencionados ao longe deste documento possuir throws na sua declaração, isso se deve pelo método clear da classe “Outros”.

ÁRVORES DE HERANÇA

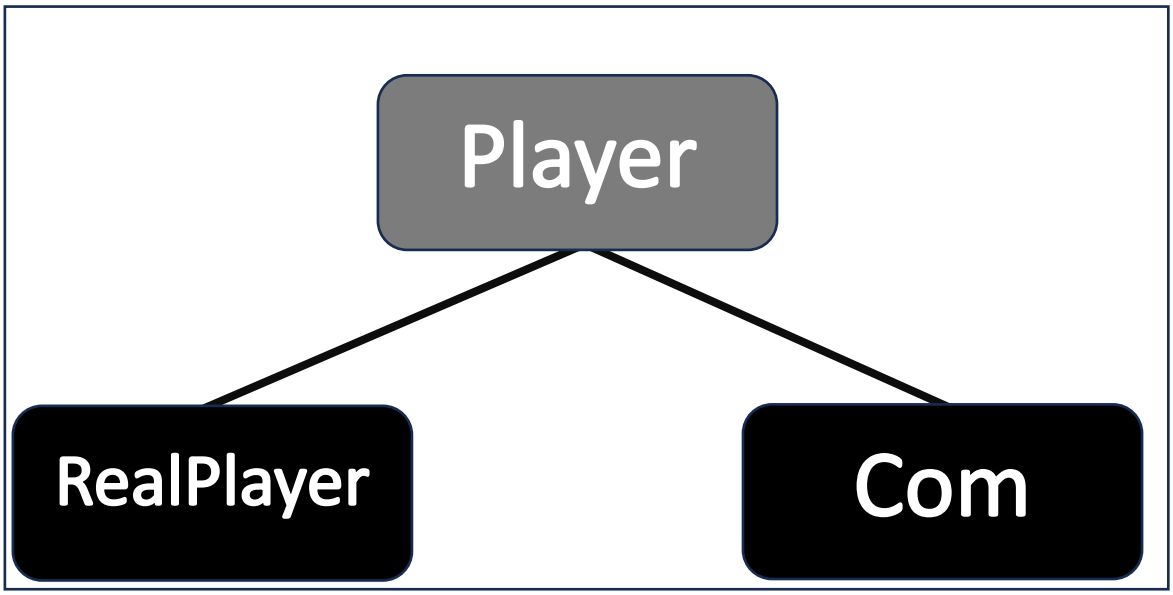
- Classe Abstrata

- Classe Normal

- **CARDS:**



- **PLAYERS:**



ESTRUTURA DO JOGO

1 - CARTAS

O principal elemento do Uno são suas cartas, para representar elas, criamos a classe Card (Carta) que representa uma carta do jogo, dentro desta classe criamos dois Enum (Uma classe especial do Java que pode transformar as variáveis que definimos dentro dela como constantes que podem ser usadas como "valores"), um para as cores (representando as cores das cartas) e uma para os valores (representando os valores das cartas, ou pelo menos a impressão dos seus valores – mais sobre isso será explicado em breve.

```
public enum Color {
    RED, BLUE, GREEN, YELLOW, WILD;
    private static final Color[] colors = Color.values();
    public static Color getColor(int i){
        return Color.colors[i];
}
    `
```

```
public enum Value {
    ZERO(0), ONE(1), TWO(2), THREE(3), FOUR(4), FIVE(5),
    SIX(6), SEVEN(7), EIGHT(8), NINE(9),
    DRAW_TWO(20), BLOCK(20), REVERSE(20),
    CHOOSE_COLOR(50), DRAW_FOUR(50);

    private static final Value[] values = Value.values();
    public static Value getValue(int i){
        return Value.values[i];
    }
}
```

Nos Values, existem números atrelados a eles, estes serão usados caso o jogo acabe sendo fechado e o vencedor será determinado pela contagem dos pontos, quem tiver o menor número de pontos vence.

Dentro da classe definimos dois atributos, o atributo de cor e o atributo de valor, (ambos definidos como "Final" pois são valores inalteráveis):

```
private final Color cardColor;

private final Value cardValue;
```

Como também seus Getters and Setters.

Nesta Classe utilizamos um Override do método toString, para que este retorne a cor e o valor do objeto separados por um underline:

```
    public String toString(){

        return this.cardColor + "_" + this.cardValue +  "_";
    }
```

1.1 – ACTIONCARDS

Derivada desta classe temos a classe abstrata ActionCards, que estende a classe Card e representa as classes com valores “especiais” e que executam determinada ação no jogo:

```
public abstract class ActionCards extends Card{

    public ActionCards(Color color, Value value) {
        super(color, value);
    }

    public abstract void ExecuteAction();
}
```

A necessidade desta classe ser abstrata é atribuir as ações as suas determinadas cartas. Neste caso apenas as cartas especiais farão uso desta classe, pois as cartas normais não possuem nenhuma ação, estas serão instanciadas a partir da Super Classe “Card”. As cartas especiais possuirão uma classe distinta para representar cada uma delas, ainda mantendo o “valor” delas para melhor verificação de validade (se pode jogar a carta) sendo assim o atributo “cardValue” tem o objetivo de representar mais uma “Impressão” como dito anteriormente.

Todas as subclasses de ActionCards precisarão ter um atributo “ExecuteAction()” que faz exatamente o que está escrito, executa uma ação referente aquela carta específica.

1.2 – BLOCK

A carta de Bloqueio no Uno impede o próximo jogador de jogar por uma rodada, passando a vez para o jogador subsequente.

```java
public class Block extends ActionCards{

    public Block(Color color, Value value) {
        super(color, value);
    }

    @Override
    public void ExecuteAction() {
        Game.verifyPlayerCounter();
    }
}
```

Para fazer isso, incrementamos o contador dos jogadores com a direção atual do jogo através do método verifyPlayerCounter(), que incrementa o contador e faz todas as verificações para que o contador não ultrapasse o limites definidos do array de jogadores.

1.3 – REVERSE

A carta Reverse ou reversão permite com que o jogador inverta o sentido do jogo, fazendo com que a rotação dos jogadores agora seja no sentido anti-horário.

```java
public class Reverse extends ActionCards{

    public Reverse(Color color, Value value) {
        super(color, value);
    }

    @Override
    public void ExecuteAction() {
        Game.setDirection();
    }
}
```

Para isso, usamos o método setDirection(); da Classe Game, que multiplica o atributo direção (que determina o sentido da rotação) por -1, fazendo com que a rotação seja invertida.

1.4 – DRAW

As cartas Draw permitem com que o jogador faça o próximo jogador puxar duas ou quatro cartas, no jogo original estas cartas são completamente diferentes, uma deles pode ser de qualquer uma das quatro cores presentes no jogo, e a outra apenas pode ser representada pela cor Wild (Coringa). Neste caso, ela pode ser presentada como uma única classe

```
public class Draw extends ActionCards{

    private int number;

    public Draw(Color color, Value value, int number) {
        super(color, value);
        this.number = number;
    }

    @Override
    public void ExecuteAction() {
        int j;
        for(j = 0; j < this.number; j++){
            Game.getPlayer(Game.NextPlayer()).drawCard();
        }
    }
}
```

Para representar quantas cartas serão puxadas pelo próximo jogador, temos dentro dessa classe o atributo number que é um inteiro que pode ser do valor 2 ou 4 (nada específico ou que impeça ele de assumir outros valores, mas dentro do gerador de cartas, ele pode apenas assumir esses dois valores).

Sua ação faz com que o próximo jogador puxe quantas forem determinadas no seu atributo, a informação de quem é o próximo jogador é adquirida através da função NextPlayer(); que faz todas a verificações para informar quem é o próximo jogador sem que os limites do array sejam excedidos.

1.5 – CHOOSE COLOR

A carta coringa “Choose Color” permite ao jogador escolher qual cor o próximo jogador vai precisar jogar, esta carta é um pouco mais complicada que as demais pois essa impõe uma condição muito específica no jogo que pode acabar impossibilitando o jogo de continuar.

```java
public class ChooseColor extends ActionCards{

    public ChooseColor(Color color, Value value) {
        super(color, value);
    }

    @Override
    public void ExecuteAction() {
        if(Game.getCurrentPlayer() instanceof Com){
            Random rand = new Random();
            Color color;
            if(Deck.getDeckSize() != 0){
                color =
Deck.getCard(rand.nextInt(Deck.getDeckSize())).getColor();
                Game.setCardColor(new Card(color, null));
            }
            else{
                color =
Game.getPlayer(0).getCard(rand.nextInt(Game.getPlayer(0).checkSize())).getColor(
);
                Game.setCardColor(new Card(color, null));
            }
        }else{
            String[] colors =
{"\u001B[31m","\u001B[34m","\u001B[32m","\u001B[33m"};
            int escolha;

            while(true) {
                for(int i = 0; i < 4; i++) {
                    System.out.printf("\u001B[37m%d - %s%s\n", (i+1), colors[i],
Color.getColor(i));
                    }
                System.out.println("\u001B[37mEscolha sua cor:");
                escolha = Outro.getNumEntry();
                escolha--;
                if(escolha <= 3 && escolha >= 0) {
                    Game.setCardColor(new Card(Color.getColor(escolha), null));
                    break;
                }
            }

            Game.CloseGame();
        }
    }
}
```

Primeiro ela verifica se quem está jogando é um Com (de Computador) ou um jogador real, pois o Com não pode escolher a cor, um algoritmo pré-definido deve fazer isso por ele, se for um com, primeiro ele verifica se existem cartas no deck principal (ou se preferir, na pilha de cartas que podem ser puxadas) se existirem, ele vai selecionar aleatoriamente uma carta deste , e definir a nova cor como a cor dessa respectiva carta, se não existirem ele verifica nas cartas do jogador qual carta ele ainda possui e pega uma carta aleatória para definir como cor.

Se for um jogador real jogando, então ele vai selecionar entre as opções de cores que serão impressas para ele no terminal.

Como dito anteriormente, essa é a única carta que pode acabar (por vontade ou não do jogador) fechando o jogo, no final do método existe uma verificação para saber se o jogo foi fechado ou não, através do método CloseGame(); este só é chamado quando um jogador real escolhe a carta pois só ele é capaz de fechar o jogo (numa situação de um jogo real, assim como no dominó, o jogador pode acabar fechando o jogo, impossibilitando outros jogadores, e ele mesmo de jogarem, e o ganhador é definido pela contagem de pontos), pois para que o jogo não acabe ficando sem graça, o Com acaba sempre selecionando uma carta que ainda existe, pois ele verifica todas as cartas do jogo para achar alguma que ainda se encaixe.

2 – DECK

Para criação do deck de cartas do Uno, usamos um sequência de loops for para criar cada carta, num jogo padrão de Uno (Com 108 cartas) temos:

19 cartas de cada cor enumeradas de 0 a 9 (apenas uma delas sendo 0 e as demais com duas instancias de cada).

- 8 cartas de reversão (duas de cada cor).
- 8 cartas de bloqueio (duas de cada cor).
- 8 cartas de puxar dois (duas de cada cor).
- 4 cartas de puxar quatro (todas Wild)
- 4 cartas coringa de escolher a cor.

Seus atributos:

```
    private static ArrayList<Card> deck = new ArrayList<Card>();

    private static int cardsDeck;
```

o ArrayList representa o Deck, este precisa que o seu tamanho e seus elementos sejam dinâmicos pois cartas serão retiradas (e adicionadas na sua criação) pelos jogadores.

cardsDeck representa apenas a quantidade de cartas no ArrayList, e vai adicionado as cartas baseado no index.

3 – PLAYERS

Mais uma classe abstrata, definida desta forma pois possuímos dois tipos de jogadores no jogo, os jogadores reais e o Com. o Com é uma sequencia de ações pré-programadas para que seja uma opção caso o jogador não possua ninguém para jogar e queira jogar sozinho.

```
    protected final String name;

    protected ArrayList <Card> hand;
```

A classe possui dois atributos, nome (identificação do jogador) e hand (ou mão) que representa as cartas na mão do jogador.

```
    public void printPlayer(){

        for(int i = 0; i < hand.size(); i++){

            System.out.println("\u001B[37m"+ (i+1) + " - " +
Outro.printCard(hand.get(i)));
        }
    }

    public void drawCard(){
        hand.add(Deck.getCard(0));
        Deck.removeCard(0);
    }

    public int checkSize(){
        return hand.size();
    }

    public Card getCard(int i){
        return hand.get(i);
    }

    public void putCard(int i){
        hand.remove(i);
    }
```

Explicando método a método:

- **printPlayer()** – Método usado para printar as cartas do jogador, o método printCard() que é chamado a partir da classe “Outro” tem como único intuito

adicionar cores para serem imprimidas no terminal, deixando o jogo mais intuitivo.

- **drawCard()** – Este método serve para quando o jogador quiser puxar uma carta (ou precisar, ele pega uma carta do Deck e remove a mesma dele (para que não fique repetida ou em loop.
- **checkSize()** – Checa o tamanho do deck do jogador para verificações de caso jogador ganhar (tiver com zero cartas na mão) ou precisar dizer Uno (estiver com apenas uma carta na mão).
- **getCard(int i)** – Pega a carta da mão do jogador baseado no index.
- **putCard(int i)** – Remove a carta da mão do jogador (usado para quando o jogar colocar sua carta na mesa).

Os dois métodos abstratos para essa classe são:

```
public abstract int ChooseCard();

public abstract boolean sayUno();
```

Como já dito anteriormente, existem dois tipos de jogadores, então precisamos que eles executem ações de forma distinta, as ações que o player pode fazer são: escolher uma carta (no caso do computador isso tem que ser feito de forma automática), e "dizer" Uno (quando for preciso). Mais sobre eles nos próximos tópicos.

3.1 – REALPLAYERS

Classe usada para representar os jogadores reais.

Nela temos o Override de dois métodos, ChooseCard(), este permite com que o jogador faça a escolha da sua carta. E SayUno() que tenta traduzir essa ação (já que não podemos falar com o computador) para um jogo que funciona no terminal. Para fazer essa tradução, precisamos colocar um tempo limite em que o jogador pode realizar esta ação, neste caso 3 segundos, pegamos o tempo antes do jogador digitar a palavra, e o tempo depois, se este tempo for menor que 3 segundos, retorna true (significando que o jogador "disse" Uno) caso contrário retorna false (simbolizando que o jogador não disse Uno) fazendo com que o jogador precise puxar duas cartas.

O método toUpperCase é usado para não importe a forma que o jogador digite, estando as letras maiúsculas ou minúsculas, ele apenas precisa digitar Uno.

```
public class RealPlayer extends Player{


    public RealPlayer(String name) {
        super(name);
    }

    @Override
    public int ChooseCard() {
        int num = Outro.getNumEntry();
        return num;
    }

    @Override
    public boolean sayUno() {
        String uno = "UNO";
        String spell;
        Scanner sc = new Scanner(System.in);
        long before = System.currentTimeMillis();
        System.out.println("Digite UNO: ");
        spell = sc.next();
        long after = System.currentTimeMillis();
        if(uno.compareTo(spell.toUpperCase()) == 0 && (after -
before) < 3000){
            return true;
        }
        return false;
    }
}
```

3.2 – COM

Agora temos o Com (Computador), este não tem ações que necessitam de Scanner ou métodos parecidos que precisem de entradas, todas as funções desta classe tem que ser executadas de forma automática.

ChooseCard() para este, apenas lê todas as cartas que estão na sua mão, alguma delas for válida, ele imediatamente retorna esta carta, caso não tenha puxa uma carta.

SayUno() este método preferimos colocar baseado em chance, já que a não ser que um fator aleatório seja introduzido, se colocássemos o computador para digitar ele sempre retornaria true, então colocamos uma chance de 80% para ele "dizer" Uno, com um Random sorteando algum número de 0 a 4.

```
public class Com extends Player{

    public Com(String name) {

        super(name);
    }

    @Override
    public int ChooseCard() {
        for(int i = 0; i < hand.size(); i++){
            if(Game.verifyValidCard(hand.get(i))){
                return i;
            }
        }
        return -2;
    }

    @Override
    public boolean sayUno() {
        Random rand = new Random();
        int chance = rand.nextInt(5);
        if(chance < 4){
            return true;
        }
        return false;
    }
}
```

4 – JOGO

O Uno é um jogo muito dinâmico e várias ações dentro do jogo alteram o andamento do jogo como um todo. Como vários dos elementos do jogo podem ser alterados dependendo da carta que o jogador jogar definimos elas como atributos da classe Game, e como só podemos jogar um jogo por vez (e também para melhor interação das cartas com o jogo principal) todos os atributos e métodos desta classe são definidos como estáticos.

```
    private static int playerCounter;
    private static int choosedCard, direction = 1;
    private static int numPlayers;
    private static Scanner sc = new Scanner(System.in);
    private static Card currentCard;
    private static Player[] players;
    private static boolean gameClosed;
```

Função de cada atributo:

- **playerCounter** – Inteiro – Representa um contador que define quem é o player atual, e vai variando de player em player dependendo de quem for jogar no momento.
- **choosedCard** – Inteiro – Atributo que representa a carta escolhida pelo jogador na hora da sua jogada (Por conta do jogo rodar no prompt de comando, esta é a melhor forma de fazer com que ele selecione alguma carta).
- **direction** – Inteiro – Diz a respeito do sentido da rotação entre os jogadores do jogo, se a rotação é horária (neste caso o será somado + 1 ao playerCounter) ou Anti-Horária (será reduzido 1 do playerCounter).
- **numPLayers** – Inteiro – Quantidade de jogadores jogando no momento.
- **sc** – Scanner – Variável que recebe alguma entrada pelo usuário através da entrada padrão do sistema.
- **currentCard** – Card – Representa a carta que está na mesa no momento, jogada pelo jogador anterior.
- **players** – Player[] – Usado para armazenar os jogadores que estão jogando o jogo no momento.
- **gameClosed** – boolean – Diz se o jogo ainda está jogável (se ainda é possível jogar alguma carta para continuar o jogo).

Para checar o código completo acesse https://github.com/MattVictor/UNO no arquivo Game.java.

4.1 – COMEÇANDO O JOGO

Para dar início ao jogo coletamos a quantidade de jogadores (fornecida pelo usuário e chamamos o método createGame():

```
public static void createGame(){

    direction = 1;
    playerCounter = 0;
    gameClosed = false;
    Deck.reset();
    currentCard = Deck.getCard(0);
    Deck.removeCard(0);

    if(numPlayers == 1){
        numPlayers++;
        players = new Player[numPlayers];
        System.out.printf("Nome do jogador: ");
        players[0] = new RealPlayer(sc.next());
        players[1] = new Com("COM");
    } else{
        players = new Player[numPlayers];
        for(int i = 0; i < numPlayers; i++){
            System.out.printf("Nome do jogador %d: ", (i+1));
            players[i] = new RealPlayer(sc.next());
        }
    }

    playerCounter = 0;
}
```

Este método inicia a direção do jogo como positiva, reseta o contador para 0, seta o jogo como aberto (colocando gameClosed como false) e reseta o Deck.

Além disso, pega a primeira carta para ser colocada na mesa (que deve ser aleatória então pegamos a primeira carta do Deck e depois eliminamos ela do Deck).

Por fim, cria o array de jogadores com o número determinado e cria os jogadores com suas respectivas identificações, se o número de jogadores for 1, então deve-se criar o objeto “Com” para jogar contra o jogador real e perguntar o nome do jogador.

Caso o número de jogadores for maior que um e menor que 4, pega a identificação de cada um e coloca-os dentro do array de players.

Começando o jogo, temos a seguinte condição:

```
while(true)
```

que define o loop do jogo, este fica em loop até que uma condição de parada seja executada.

Em seguida a primeira condição que separa os jogadores reais, do Com:

```
            if(players[playerCounter] instanceof RealPlayer){
                Outro.printStatus(players, playerCounter);
                System.out.printf("Carta atual: %s\n\n",
Outro.printCard(currentCard));
                System.out.printf("\u001B[37mCartas de %s:\n",
players[playerCounter].getName());

                players[playerCounter].printPlayer();

                System.out.println();
                System.out.println("\u001B[37mEscolha sua carta
(número negativo para puxar uma carta, 0 para passar)");
                choosedCard = players[playerCounter].ChooseCard();
                choosedCard--;
            }else{
                choosedCard = players[playerCounter].ChooseCard();
            }
```

Aqui separamos algumas coisas que só devem ser executadas caso o jogador seja real, como mostrar o status do jogo (Quantas cartas os seus oponentes tem), mostrar suas cartas e fazer com que ele escolha a carta que quer jogar, fazendo um tratamento para que a carta escolhida esteja condizente com o index do array de cartas da sua mão.

Caso o jogador não seja um jogador real (neste caso, um Com), então a função de escolher carta vai ser executada de acordo com o Player, aqui temos o primeiro caso de polimorfismo, onde chamamos o método ChooseCard() de um array de players, neste caso chamando o método da subclasse a qual ele está referenciando no momento.

4.2 – CONDIÇÕES

Algumas condições tem que ser verificadas para saber qual ação deve ser tomada e o que devemos mostrar para o jogador

4.2.1 – PASSAR

```
if(choosedCard == -1){

      verifyPlayerCounter();
      Outro.changingPlayers();
   }
```

Caso o jogador digite 0 (tratamento realizado no tópico anterior) ele executa a função de passar, e passa a vez para o próximo jogador.

4.2.2 – PUXAR CARTA

```
else if(choosedCard < -1 && Deck.getDeckSize() != 0){

       Outro.LimpaConsole();

       players[playerCounter].drawCard();

       if(!verifyValidCard(players[playerCounter].getCard(players[pla
yerCounter].checkSize()-1))){
         verifyPlayerCounter();
         Outro.changingPlayers();
       }
```

Para esta condição ser acionada o jogador precisa digitar algum número negativo, e o Deck precisa ter alguma carta, se essas duas condições forem satisfeitas, o jogador puxa uma carta, se a carta que ele puxou for da mesma cor ou valor que a carta que está atualmente na mesa, então ele terá a opção de jogar esta carta, se não for, então ele passa a vez para o próximo jogador.

4.2.3 – NÃO EXISTEM MAIS CARTAS PARA SEREM PUXADAS

```
else if(choosedCard < -1 && Deck.getDeckSize() == 0){

 verifyPlayerCounter();
 if(players[playerCounter] instanceof RealPlayer){
     Outro.LimpaConsole();
     System.out.println("Não Há mais cartas a serem puxadas!");
     Outro.sleep(3);

     Outro.changingPlayers();
     }
     if(gameClosed){
       winnerByPoints();
       break;
     }
```

Caso o jogador tente puxar uma carta e o ArrayList do Deck estiver vazio, então está condição será acionada, mostrando para o jogador que não existem mais cartas a serem puxadas e verificando se o jogo está fechado.

4.2.4 – CARTA NÃO EXISTE

```
else if(choosedCard > players[playerCounter].checkSize()-1){

   Outro.LimpaConsole();

   System.out.println("Carta escolhida não existe!");
}
```

Caso o jogador tente escolher uma carta que está fora dos limites do ArrayList que representa sua mão, então esta condição será acionada, imprimindo que ele escolheu uma carta que não existe na sua mão.

4.2.5 – CARTA ESCOLHIDA É VÁLIDA

```
else
if(verifyValidCard(players[playerCounter].getCard(choosedCard))){

   currentCard = players[playerCounter].getCard(choosedCard);
   players[playerCounter].putCard(choosedCard);

   if(players[playerCounter].checkSize() == 1 && Deck.getDeckSize() >
1){
   if(!players[playerCounter].sayUno()){
       players[playerCounter].drawCard();
       players[playerCounter].drawCard();
     }
   }

   if(verifyWinner()){
      break;
   }

   if(currentCard instanceof ActionCards){
      ((ActionCards)currentCard).ExecuteAction();
   }

   if(gameClosed){
     winnerByPoints();
     break;
     }

   verifyPlayerCounter();
   Outro.changingPlayers();
   Outro.LimpaConsole();
}
```

Aqui temos a condição principal do jogo, quando uma carta for válida (possuir a mesma cor ou valor da carta da mesa).

Primeiro colocamos a carta que o jogador escolheu na mesa, atribuindo referenciando ela na variável currentCard (esta iremos usar a partir de agora para fazer as verificações) e removemos esta da mão do jogador.

Em seguida, se o jogador tiver com apenas uma carta na mão e o Deck tiver mais que 1 carta, verificamos se ele vai “falar” Uno (Método chamado a partir do próprio player e já explicada em tópicos anteriores). Se ele “falar” Uno (retorno do método for true) ele continua normalmente, caso ele não fale (retorno do método for false) ele puxa duas cartas do Deck. Na condição invertemos o valor do retorno da função para que fique mais legível.

Em seguida verificamos se temos um ganhador (já que o jogador colocou uma carta), se o jogador atual tiver sem cartas na mão, então ele é o vencedor, todas as verificações para determinar o vencedor, bem como a apresentação deste está dentro do método verifyWinner(). Tendo um vendedor, quebramos o gameloop e voltamos para a tela inicial.

Caso a condição anterior seja falsa (já que caso já tenhamos um vencedor, não é necessário verificar qual carta ele jogou para saber se ela executa alguma ação), verificamos se a carta lançada pelo jogador é uma ActionCard, através da palavra reservada instaceOf que nos permite saber se o objeto do tipo carta é uma instância do tipo ActionCard, se for, então fazemos o DownCast (neste caso totalmente seguros pois já verificamos que a carta atual é uma ActionCard, e executamos sua respectiva ação.

Como já comentado anteriormente, a carta Choose Color, é a única que é capaz de fechar o jogo completamente, então a próxima condição é verificar se o atributo designado para saber se o jogo ainda está ativo (gameClosed) é igual a false (significando que o jogo ainda está ativo e ainda existem cartas que possam ser jogadas por quaisquer jogadores), se for o bloco não será executado, caso seja true (significa que o jogo está fechado) o bloco executa e vamos para a contagem de pontos.

Passadas todas estas verificações, incrementamos o contador do jogador, fazemos a transição entre os jogadores e continuamos o jogo.

4.2.6 – CARTA INVÁLIDA ESCOLHIDA

```
else{

Outro.LimpaConsole();
System.out.println("Carta inválida escolhida");
}
```

Caso nenhuma das condições anteriores não seja executada, isso significa que, o jogador não escolheu passar, não escolheu puxar uma carta, e a carta que ele escolheu não era válida, neste caso apenas evidenciamos que a carta escolhida pelo jogador não é valida e ele tem que selecionar uma nova carta.

4.3 – MÉTODOS AUXILIARES

Alguns métodos são definidos na classe de jogo para realizar algumas verificações e determinar algumas condições para o jogo funcionar como deve.

4.3.1 – VERIFYPLAYERCOUNTER

```
public static void verifyPlayerCounter(){

    playerCounter+=direction;
    if(playerCounter == numPlayers){
        playerCounter = 0;
    }
    else if(playerCounter < 0){
        playerCounter = numPlayers-1;
    }
}
```

Este método é usado quando vamos fazer a troca de jogadores, nele aumentamos a seta de player pelo valor da direção (se for positivo aumenta, se for negativo diminui) e fazer as verificações necessárias para que o valor da seta não exceda os limites do array que armazena os jogadores.

4.3.2 – NEXTPLAYER

```
public static int NextPlayer(){

    if(playerCounter == 0 && direction < 0){
        return numPlayers-1;
    }
    else if(playerCounter == numPlayers-1 && direction > 0){
        return 0;
    }
    return playerCounter+direction;
}
```

Funcionamento semelhante ao método anterior, mas retorna o próximo player, esse método é usado por ações de cartas que afetam o próximo jogador, tais como Block e Draw. As verificações são semelhantes as usadas no método anterior mas esta retorna um inteiro referente ao próximo player.

4.3.3 – VERIFYVALIDCARD

```
public static boolean verifyValidCard(Card cartaPlayer){

    if(currentCard.getColor() == Color.WILD || cartaPlayer.getColor()
== Color.WILD)
        return true;
    else if(currentCard.getColor() == cartaPlayer.getColor())
        return true;
    else if(currentCard.getValue() == cartaPlayer.getValue())
        return true;
    else
        return false;
    }
```

Este é o método principal que faz o jogo ser possível, basicamente verifica se a cor e ou o valor são iguais a carta que está na mesa atualmente, caso algum deles seja, ou, caso a carta seja do tipo Wild (definido como cor nas cartas para facilitar a verificação, qualquer carta pode ser colocada em cima dela independente da sua cor ou valor) ele retorna true, caso nenhuma das condições descritas acima seja cumprida, retorna false.

4.3.4 – VERIFYWINNER

```
    public static boolean verifyWinner() throws InterruptedException,
IOException{

        if(players[playerCounter].checkSize() == 0){
            Outro.LimpaConsole();
            System.out.printf("Ganhador: %s\n\n",
players[playerCounter].getName());
            System.out.println("Digite e aperte Enter para voltar ao
Menu principal");
            sc.next();
            return true;
        }
        return false;
    }
```

Método que verifica se o jogador ganhou, simples funcionamento, verifica se o jogador atual está sem cartas na mão, se tiver declara ele como ganhador, e volta para o método RunGame(), para quebrar o gameloop e voltar para a tela inicial.

4.3.5 – WINNERBYPOINTS

```
    public static void winnerByPoints() throws InterruptedException,
IOException{

        Player wPlayer = players[0];
        int leastNumberOfPoints = 10000;
        int playerPoints = 0;
        for(Player player : players){
            playerPoints = 0;
            for(int j = 0; j < player.checkSize(); j++){
                playerPoints += player.getCard(j).getCardPoints();
            }
            if(playerPoints < leastNumberOfPoints){
                wPlayer = player;
                leastNumberOfPoints = playerPoints;
            }
        }
        Outro.LimpaConsole();
        System.out.printf("Ganhador %s com um total de %d
Pontos\n\n", wPlayer.getName(), leastNumberOfPoints);
        System.out.println("Digite e aperte Enter para voltar ao Menu
principal");
        sc.next();
    }
```

Caso o jogo feche, o jogo vai para a contagem de pontos, e chama esse método para saber quem é o ganhador (o jogador que estiver com menos pontos). Para começar definimos um player para inicializar a variável que referenciará o jogador vencedor, e colocamos um valor bem alto (impossível de se adquirir mesmo com todas as cartas na sua mão) e fazemos a checagem para saber qual player tem a menor quantidade de pontos, através de um laço for each no array de players, aquele que tiver o menor número de pontos será o vencedor, após isso (e após o usuário apertar alguma tecla + Enter, o jogo volta para o método RunGame() e quebra o gameloop.

4.3.6 – CLOSEGAME

Este método é usado apenas em duas ocasiões, depois de um player escolher uma carta através do método ExecuteAction da carta de tipo ChooseColor, pois se este escolher uma cor que não é mais possível jogar, então o jogo está fechado. E caso todas as cartas tenham sido puxadas do Deck e não existir mais nenhuma carta de cor semelhante aquela presente na mesa, então o jogo também está fechado para este caso.

```
    public static void CloseGame(){

        boolean exists = false;
        if(Deck.getDeckSize() != 0){
            for(int i = 0; i < Deck.getDeckSize(); i++){
                if(Game.getCurrentCard().getColor() ==
Deck.getCard(i).getColor()){
                    exists = true;
                    break;
                }
            }
        }
        if(!exists){
            for(Player player : players){
                for(int i = 0; i < player.checkSize(); i++){
                    if(Game.getCurrentCard().getColor() ==
player.getCard(i).getColor()){
                        exists = true;
                        break;
                    }
                }
            }
        }
        if(!exists){
            gameClosed = true;
        }
    }
```

Temos essencialmente duas verificações a fazer aqui, uma delas sendo verdadeira é suficiente pra dizer que o jogo não está totalmente fechado ainda. Primeiro ele verifica se a cor referente a carta presente na mesa no momento existe no Deck (ainda pode ser puxada) caso essa condição seja verdade, então o jogo está aberto, a carta existe então a segunda condição e a terceira não serão executadas. Se a carta não se fizer presente no Deck, é hora de verificarmos na mão de todos os players, se ela existir na mão de algum deles, da mesma forma, o jogo está aberto, a terceira condição é pulada, se a cor não se fizer presente em nenhum dos dois lugares citados acima, então o jogo está fechado, a carta não existe, e declaramos o atributo gameClosed como True, fechando o jogo e indo para a contagem de pontos.