Introduction

The program is a card game called "Durak," which means "fool" in Russian.

To play this game, you need a standard deck of 36 cards. The cards should have numbers 6, 7, 8, 9, 10, Jack, Queen, King and Ace, as well as four suits: HEARTS, DIAMONDS, CLUBS, and SPADES.

Game rules:

Before the game starts, the cards are shuffled randomly. Then, the trump suit is revealed. This is a special suit that can beat all other suit cards, but it cannot beat a higher-ranking card of the same suit.

Each player gets six cards. In our case, there are two players: one is a human player (us), and the other is a computer opponent (bot).

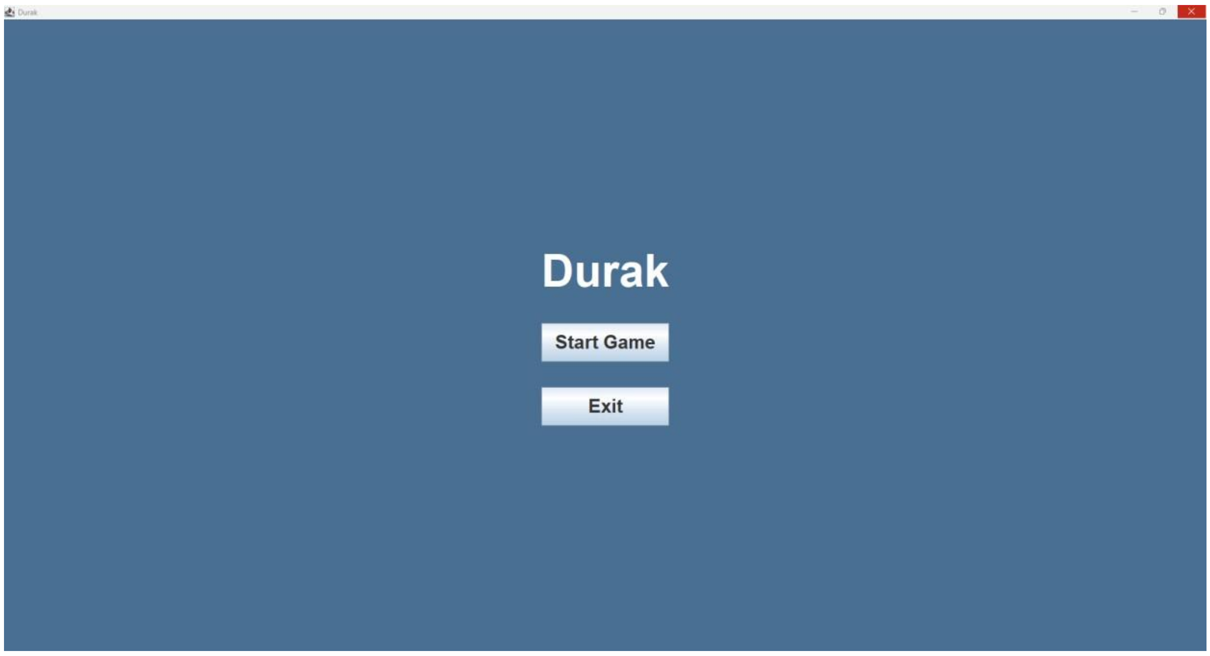
The game begins. As a player, we start by choosing one card from our hand to discard. The opponent (the bot) must then respond with a stronger card, either a higher-ranking card or a card of the trump suit. If the bot cannot beat our card, they take it, and the round ends. We then make our move again in the next round.

If the bot manages to beat our card, they make a move in the next round. After each round, both players check their hands. If a player has fewer than six cards left, they take a card from the deck until they have six cards again or until the deck runs out of cards.

The game continues until one player has no cards left. That player wins the game.

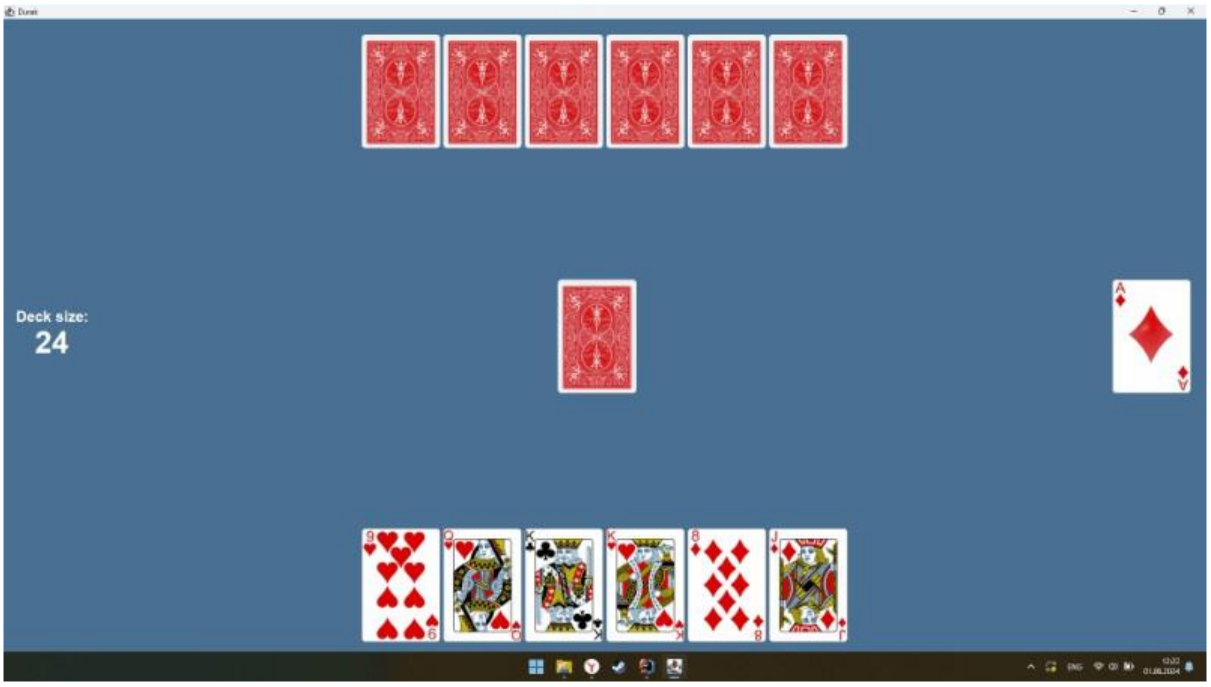
Interaction and graphics:

Now, let's discuss the interaction and graphics in the game. The graphics are built using JFrame Panels. When we launch the game, we are greeted with the main menu, which features buttons for starting the game and exiting it.



When we click on the "Start" button, we are taken to the game itself. As you can see, the game we're playing is "Durak," which was mentioned earlier.

Below, you'll find the cards that are in the player's hands. We can interact with these cards by clicking on them. On top of the screen, you'll see the cards of our opponent, which are not visible to us. In the middle, there is a card that is hidden when it's our turn and shown when it's our opponent's turn. On the left, you'll see the number of cards remaining in the deck. On the right, you'll find the trump suit.



After completing the game, an image of victory or lose pops up. 7 seconds after this the game will shut down.

OOP principles in this program:

Let's explore each of the object-oriented programming (OOP) principles along with exception handling, using examples from our Durak card game project:

1. Modularity

Modularity involves breaking down a software system into smaller, selfcontained modules, each responsible for a specific functionality. This promotes code organization, reusability, and maintainability.

In the Durak game project, modularity is achieved by separating different aspects of the game into distinct classes Each class handles specific functionalities, such as UI management, game logic, card handling, and player interactions. Namely, GameMenu, MainBeta2, Functions, Variables, Card, Deck,

GameParticipant, Player, and Bot. Also, Game52 – another mode of game, New Opponent as third player, and other classes similar as before . Let's look at each of them. The Card class generates cards from the following values: Rank and Suit. The Deck class generates an arraylist from elements created in
Card. Player and Bot for the logic of moving cards that inherit the properties of GameParticipant, which in turn inherits from the Functions class. The Variables class stores player cards and more. In the MainBeta2 there is whole game logic.

2. Hierarchy

Hierarchy refers to the arrangement of classes in a parent-child relationship, where child classes inherit attributes and behaviors from their parent classes. This facilitates code reuse and promotes polymorphism.

Hierarchy is demonstrated in the Durak game through class inheritance. For instance, the Player, Bot, Player52, Bot52 and Opponent classes inherit from the GameParticipant class, sharing common functionalities. This hierarchy allows both players and bots to be treated uniformly in the game logic while providing flexibility for extending their behaviors.

```
public class Player extends GameParticipant<Card> {
   @Override    public void
playCard(Card card) {
```

3. Composition

Composition involves constructing complex objects by combining simpler ones. This promotes code reuse, flexibility, and modularity by allowing objects to be composed of other objects.

Composition is seen in the Durak game's UI design using Swing components. For example, the GameMenu class is composed of buttons (JButton) and labels (JLabel) to create the game menu UI. By composing these components, the UI can be easily customized and extended.

Here’s another examples where composition is used:

GameParticipant and its subclasses (Player, Bot, Player52, Bot52 and Opponent): GameParticipant is an abstract class, and Player, Bot, Player52, Bot52 and Opponent are concrete subclasses. Here, Player and Bot are composed of GameParticipant through inheritance. They represent different types of participants in the game, and they inherit common behaviors from GameParticipant.

Variables class: This class contains various static methods and fields related to game variables and paths to resources. It's a composition of static fields and methods that are used across the game.

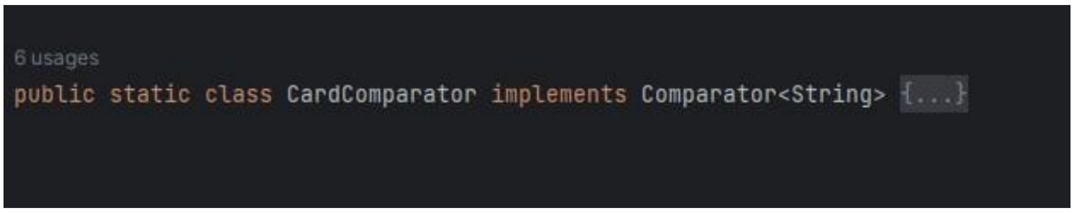
Functions class: This class contains utility methods that are used in the game logic. It's composed of various static methods that perform common functions such as card comparison and game state checking.

Deck class: This class represents a deck of cards. It contains a list of Card objects and methods to manipulate the deck, such as initializing, shuffling, and drawing cards. The Deck class is composed of a list of Card objects. In addition there is another deck class called "Deck52" which is deck for game with 52 cards.

4. Reuse

Reuse involves leveraging existing code or components for new purposes rather than reinventing them. This promotes efficiency, consistency, and maintainability by reducing redundancy.

The Functions class contains several utility methods (compare(), endGame(), give(), etc.) that are reused across different parts of the codebase. These methods encapsulate common functionality related to comparing card ranks, handling game ending logic, and managing card distribution.



```
6 usages
public static class CardComparator implements Comparator<String> {...}
```

Variables class stores reusable variables and resource paths (buttonSoundPath, menuSoundPath, gameSoundPath, etc.). These variables are reused throughout the application to maintain consistency and facilitate easy modification of resource paths.

5. Encapsulation

Encapsulation is the bundling of data (fields) and methods (functions) that operate on the data into a single unit, typically a class. It ensures that the internal representation of an object is hidden from the outside and only necessary operations are exposed. Encapsulation allows us to control how data is accessed and modified within an object.

For example, in the Deck class, the list of cards is encapsulated within the class, and operations such as shuffling, drawing a card, and retrieving the deck size are provided through public methods:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Deck {
    private final List<Card> cards; // Encapsulated data

    public Deck(int mode) {
        cards = new ArrayList<>();
        for (Suit suit : Suit.values()) {
            for (Rank rank : Rank.getRanks(mode)) {
                cards.add(new Card(suit, rank));
            }
        }
        shuffle();
    }
    public void shuffle() {
        Collections.shuffle(cards);
    }

    public Card drawCard() {
        return cards.isEmpty() ? null : cards.remove(0);
    }

    public int size() {
        return cards.size();
    }

    public List<Card> getCards() {
        return cards;
    }
}
```

Here, the cards list is encapsulated inside the Deck class, and direct access to it is restricted. Instead, controlled access is provided through methods like shuffle(), drawCard(), and getCards().

6. Subtyping

Subtyping allows objects of a subtype to be treated as objects of their supertype, enabling polymorphism and dynamic binding. This promotes flexibility and extensibility in object-oriented designs.

Let's consider the GameParticipant class and its subclasses Player and Bot. Here's how subtyping is implemented:

Inheritance: The Player and Bot classes extend the GameParticipant class, which establishes an "is-a" relationship. Player, Bot, Player52, Bot52 and Opponent are types of GameParticipant, inheriting its methods and properties.

```
public abstract class GameParticipant<T> {
public abstract void playCard();
}

public class Player extends GameParticipant<Card> {
   @Override    public void
playCard(Card card) {
        ...
   }
}

public class Bot extends GameParticipant<Card> {
   @Override    public void
playCard(Card card) {
        ...
   }
}
```

Polymorphism: Due to inheritance, instances of Player and Bot can be treated as instances of their superclass GameParticipant. This allows for polymorphic behavior, where methods defined in GameParticipant can be overridden in subclasses and invoked through superclass references.

```
player.playCard(null); // Calls playCard() method specific to Player class
bot.playCard(null);    // Calls playCard() method specific to Bot class
```

Substitutability: Because Player and Bot are subtypes of GameParticipant, instances of Player and Bot can be used interchangeably where GameParticipant is expected. This promotes code flexibility and reuse.

```
GameParticipant<Card> player = new Player();
GameParticipant<Card> bot = new Bot();
```

7. Information Hiding

Information hiding is a principle that restricts access to certain implementation details by using access modifiers such as private, protected, and public. The goal is to prevent unintended interference and to expose only what is necessary for other parts of the code.

For example, in the Variables class, private lists are hidden from direct access, and they can only be accessed through getter methods:

```
public class Variables {

    private static final ArrayList<String> bot = new ArrayList<>();

    private static final ArrayList<String> list = new ArrayList<>();



    public static ArrayList<String> getBot() {

        return bot;

    }



    public static ArrayList<String> getList() {

        return list;

    }

}
```

Here, bot and list are hidden from direct modification by declaring them private, ensuring that they can only be accessed through getBot() and getList(), which control how the data is exposed.

- **Encapsulation** is about grouping data and behavior inside a class, ensuring modularity and maintainability.

- **Information Hiding** is about restricting access to implementation details using access modifiers.

8. Abstraction

Abstraction is a fundamental concept in object-oriented programming that involves hiding the implementation details of a class and only showing the necessary features of an object to the outside world. It allows programmers to focus on what an object does rather than how it achieves its functionality. Abstraction is typically achieved through abstract classes, interfaces, and method signatures.

The GameParticipant class serves as an abstract representation of game participants (both players and bots). It defines the playCard() method, which must be implemented by its subclasses. This abstraction allows you to treat players and bots uniformly in certain contexts without worrying about their specific implementations.

```
public abstract class GameParticipant<T>  extends Functions {    public
abstract void playCard(T card);
}
```

Abstraction is also evident in method signatures. For example, methods like playCard() in the GameParticipant class and endGame(), give(), and compare() in the Functions class provide a high level of abstraction by defining what actions can be performed without exposing the underlying implementation details.

Subclasses of GameParticipant, such as Player and Bot, are required to implement this method according to their specific behavior in the game. This abstraction allows to define common behavior for game participants while leaving the specific implementation details to the subclasses.

Abstraction in this code promotes modularity, extensibility, and maintainability by allowing you to define clear boundaries between different components of application. It hides unnecessary details and complexity, making code easier to understand and work with.

9. Inheritance

Inheritance is a fundamental object-oriented programming concept that allows a class (subclass or child class) to inherit properties and behaviors from another class (superclass or parent class). In this code, inheritance is used to establish an "is-a" relationship between classes, where a subclass "is a" type of its superclass.

```
public abstract class GameParticipant<T> extends Functions {    public abstract void playCard(T card);
}
public class Bot extends GameParticipant<Card> {
```

GameParticipant is an abstract class that serves as the superclass defining a common behavior for game participants.
Player and Bot are subclasses that inherit from the GameParticipant class, indicating that both players and bots are game participants.
Both Player and Bot classes override the playCard() method inherited from the GameParticipant class to provide their own implementations specific to each type of game participant.

inheritance in the code promotes code reuse, polymorphism, abstraction, and extensibility, contributing to a more modular and maintainable design.

10. Polymorphism

Polymorphism allows objects of different classes to be treated uniformly through a common interface, enabling dynamic method binding and flexibility in method invocation.

Polymorphism is demonstrated in the Durak game by treating both player and bot objects as instances of the GameParticipant class. This allows for the invocation of the playCard() method on player and bot objects uniformly, with the specific implementation determined at runtime based on the object type.

In MainBeta2 class, there is interaction with instances of GameParticipant without needing to know whether they are players or bots. This promotes flexibility and simplifies code maintenance.

```
private static void onImageClick(int index) {
   GameParticipant player = new Player();     GameParticipant
bot = new Bot();     if (!imageClicked) {        clickedImageIndex
= index;        // q(clickedImageIndex);        imageClicked =
true; // Ensure the image is clicked        if (isTurn()) {
player.playCard();
            Functions.refresh();
        } else {
            bot.playCard();
            Functions.refresh();
        }
    }
}
```

Parametric polymorphism is implemented using generics, which allow classes and methods to be parameterized by types. This means that a single class or method can operate on objects of various types, specified as type parameters.

```
private static abstract class start<zxc> {
   public abstract void startGame(Deck deck, List<zxc> list, List<zxc> bot, String
trump);
}

private static class Game extends start<String> {
```

The Start class is modified to be a generic class, allowing it to work with lists of any specified type. The Game class now extends the generic Start class with String as the type parameter.

Here is also another example:

```
public abstract class GameParticipant<T> extends Functions {
public abstract void playCard(T card);
}

public class Player extends GameParticipant<Card> {
   @Override    public void
playCard(Card card) {
```

The GameParticipant class is now generic, allowing subclasses to specify the type of card they will work with. The Player class extends GameParticipant with String as the type parameter.

Parametric polymorphism enables subclasses to specify the type they work with, improving code reuse and clarity.

Overloading Polymorphism refers to the ability to define multiple methods with the same name but different parameter lists within the same class. These methods are differentiated based on the number and type of their parameters. Method overloading enhances code readability and organization by allowing similar operations to be accessed through a common method name.

```
public static void playSound(String filepath) {
```

```
public static void playSound(String filepath,boolean loop)
{
```

The first one is used for playing short, without looping. Namely, it was used in clicking buttons, clicking cards, winning and losing sound after game end.

Another one is used for playing background music with an option to loop. GameMenu and MainBeta2 background music.

Method overloading makes the code more readable and organized by grouping logically similar operations under a single method name. Clients of the class can use the appropriate playSound() method without worrying about the underlying implementation details. Overloaded methods provide flexibility in handling different scenarios, such as playing a sound once or looping it continuously.

```
public static void refresh(int a){//Refreshes images
        Game52.updateImages();
        Game52.waitForNextClick();
}

    public static void refresh(){//Refreshes images
        MainBeta2.updateImages();
        MainBeta2.waitForNextClick();
    }
```

11. Exception Handling

Exception handling is a critical aspect of robust software development that allows you to gracefully manage unexpected errors or exceptional situations that may arise during program execution. Exception handling is implemented to handle potential errors when dealing with file I/O, audio playback, and other operations that may throw exceptions.

```
public static void playClickSound(String filepath) {    try {
      File soundFile = new File(filepath);
      AudioInputStream audioStream =
AudioSystem.getAudioInputStream(soundFile);       Clip clip =
AudioSystem.getClip();       clip.open(audioStream);
      clip.start();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Code involves reading and writing files, which can throw IOException if there are issues accessing or manipulating the files. When playing audio files, exceptions like LineUnavailableException or UnsupportedAudioFileException can occur if there are issues with audio playback. Exception handling is used to catch and handle these exceptions to ensure uninterrupted execution of the program.