Avd. Object Oriented Programing

Created	@May 30, 2023 3:39 PM
Class	AVD.OOP
Type	Lab & Lecture
Reviewed	☐

▼ ***Four Main Object Oriented Programming Concepts of Java***

▼ ***• Abstraction***

- Abstraction is a process of hiding implementation details and exposes only the functionality to the user. In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.
- Abstraction is also defined as the way we indicate functionalism and mechanical ideas of a concept that can be broken down to the plain nature of the function such as what it does
- example: *A driver will focus on the car functionality (Start/Stop -> Accelerate/ Break), he/she does not bother about how the Accelerate/ brake mechanism works internally.*
- Key-points of Abstract:
    - The class should be abstract if a class has one or many abstract methods
    - Allows constructors, static methods and final method
    - Abstract class can’t be instantiated directly with the ***new*** operator.

▼ ***Example (code)***

```
// Abstract class
public abstract class Car {
    public abstract void stop();
}

// Concrete class
public class Honda extends Car {
    // Hiding implementation details
    @Override public void stop()
    {
        System.out.println("Honda::Stop");
        System.out.println(
            "Mechanism to stop the car using break");
    }
}
```

```
public class Main {
    public static void main(String args[])
    {
        Car obj
            = new Honda(); // Car object =>contents of Honda
        obj.stop(); // call the method
    }
}
```

▼ • *Encapsulation*

- Encapsulation is the process of wrapping code and data together into a single unit.
- In order to achieve encapsulation in java follow certain steps as proposed below:
    - Declare the variables as private
    - Declare the **setters and getters** to set and get the variable values

---

advantages of encapsulation in java as follows:

Control Over Data: *We can write the logic in the setter method to not store the negative values for an Integer. So by this way we can control the data.*

Data Hiding: *The data members are private so other class can’t access the data members.*

Easy to test: *Unit testing is easy for encapsulated classes*

▼ *Example (code)*

```
// AJavaclasswhichisafullyencapsulatedclass.
publicclass Car
{

    // privatevariable
    privateStringname;

    // gettermethodforname
    publicStringgetName()
  {
     returnname;

  }

    // settermethodforname
    publicvoidsetName(Stringname)
  {
     this.name = name
  }

}


// Javaclasstotesttheencapsulatedclass.
public class Test
{
   publicstaticvoidmain(String[]args)
  {

      // creatinginstanceoftheencapsulatedclass
      Carcar
      = newCar();

      // settingvalueinthenamemember
      car.setName("Honda");

      // gettingvalueofthenamemember
      System.out.println(car.getName());

  }

}
```

▼ ***<u>• Inheritance</u>***

- **Inheritance** is the process of one class inheriting properties and methods from another class in Java. Inheritance is used when we have **is-a** relationship between objects. Inheritance in Java is implemented using **extends** keyword.
- ***The 5-levels of inheritance:***
    - **Single Inheritance:** Class B inherits Class A using extends keyword
        - imagery rep:

```
+----------+        +----------+
| Class A  |        | Class B  |
+----------+        +----------+
       |                    ^
       |       Extends      |
       +--------------------+
 Parent Class           Child Class
```

Class B inherits from Class A in a single level inheritance.

- **Multilevel Inheritance:** Class C inherits class B and B inherits class A using extends keyword
    - Imagery rep:

```
+-------+         +-------+         +-------+
| Class A|        |Class B|         |Class C|
+-------+         +-------+         +-------+
     ^                 ^                 ^
     |                 |                 |
     +-----------------+-----------------+
         Inherit         Inherit
         Extend          Extend
```

- **Hierarchy Inheritance:** Class B and C inherits class A in hierarchy order using extends keyword
    - Imagery rep:

```
        +-------+
        |Class A|
        +-------+
             ^
             |   Extends
        +-------+
        |Class B|
        +-------+
             ^
             |   Extends
        +-------+
        |Class C|
        +-------+
```

- **Multiple Inheritance:** Class C inherits Class A and B. Here A and B both are superclass and C is only one child class. Java is not supporting Multiple Inheritance, but we can implement using Interfaces.
    - Imagery rep:

```
classDiagram
    ClassA <|-- ClassC
    ClassB <|-- ClassC
    ClassA : +methodA()
    ClassB : +methodB()
    ClassC : +methodC()
```

- **Hybrid Inheritance:** Class D inherits class B and class C. Class B and C inherits A. Here same again Class D inherits two superclass, so Java is not supporting Hybrid Inheritance as well.
    - Imagery rep:

```
       +-------+         +-------+
       |Class A|         |Class B|
       +-------+         +-------+
            ^                 ^
            |   Extends       |
       +-------+         +-------+
       |Class C|         |Class D|
       +-------+         +-------+
            ^                 ^
            |   Extends       |
       +-------+         +-------+
       |       |              |
       |Class A|               |
       |       |               |
       +-------+               |
                              |
                           +-------+
                           |Class D|
                           +-------+
```

Example (code)

```
// super class
class Car {
  // the Car class have one field
  public String wheelStatus;
  public int noOfWheels;

  // the Car class has one constructor
  public Car(String wheelStatus, int noOfWheels)
  {
    this.wheelStatus = wheelStatus;
    this.noOfWheels = noOfWheels;
  }

  // the Car class has three methods
  public void applyBrake()
  {
    wheelStatus = "Stop" System.out.println(
      "Stop the car using break");
  }

  // toString() method to print info of Car
  public String toString()
  {
    return ("No of wheels in car " + noOfWheels + "\n"
        + "status of the wheels " + wheelStatus);
  }
}

// sub class
class Honda extends Car {

  // the Honda subclass adds one more field
  public Boolean alloyWheel;

  // the Honda subclass has one constructor
  public Honda(String wheelStatus, int noOfWheels,
        Boolean alloyWheel)
  {
    // invoking super-class(Car) constructor
    super(wheelStatus, noOfWheels);
    alloyWheel = alloyWheel;
  }

  // the Honda subclass adds one more method
  public void setAlloyWheel(Boolean alloyWheel)
  {
    alloyWheel = alloyWheel;
  }

  // overriding toString() method of Car to print more
  // info
  @Override public String toString()
  {
    return (super.toString() + "\nCar alloy wheel "
        + alloyWheel);
  }
}

// driver class
public class Main {
  public static void main(String args[])
  {

    Honda honda = new Honda(3, 100, 25);
    System.out.println(honda.toString());
  }
}
```

▼ *Polymorphism*

- Polymorphism is the ability to perform many things in many ways. The word Polymorphism is from two different Greek words- poly and morphs. “Poly” means many, and “Morphs” means forms. So polymorphism means many forms. The polymorphism can be present in the case of inheritance also. The functions behave differently based on the actual implementation.
- There are two types of polymorphism:
  1. **Static or Compile-time Polymorphism**: when the compiler is able to determine the actual function, it’s called **compile-time** polymorphism. Compile-time polymorphism can be achieved by **method overloading** in java. When different functions in a class have the same name but different signatures, it’s called method overloading. A method signature contains the name and method arguments. So, overloaded methods have different arguments. The arguments might differ in the numbers or the type of arguments.
  2. **Dynamic or Run-time Polymorphism**: **w**hen the compiler is not able to determine whether it’s superclass method or sub-class method it’s called **run-time** polymorphism. The run-time polymorphism is achieved by **method overriding**. When the superclass method is overridden in the subclass, it’s called method overriding.

▼ *Class #3*

- Use case Diagram
  - Primary audience is developers

    - also Costumers
- UML (Unified Modeling Language): Unified Modeling Language (UML) is a visual modeling language that helps software developers visualize and construct new systems. UML is not a programming language, but rather a set of rules for drawing diagrams.
- OMG UML
- 4+1 View Model —- Different diagrams for different facets
    - Logical View, process view, development view, physical view, use case view.

- EXCERSICE:

Pair) Create a use case diagram for the Omok game application by identifying the following elements:

- Actors
- System boundary
- Use cases

Additionally, provide concise descriptions for the identified actors and use cases. Refer to the first lessons for the various types of games that will be supported. Even when working in pairs, please submit your work individually.

- Actors: Player (1vPC, 1v1) 2 modes of playing
- System Boundaries
- Use cases:

▼ ***Class #5***

-

▼ ***Class #10***

-

▼ ***How to build a basic GUI (Graphical User Interface)***

▼ ***CODE***

▼ Descriptive Text

1. `JFrame frame = new JFrame("My App");`: This line creates a new JFrame object. The JFrame is the top-level container for your Swing application. This frame will have the title "My App".

2. `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`: This line tells the Java runtime what it should do when the user closes the JFrame. In this case, the application will terminate.
3. `frame.setSize(300, 200);`: This line sets the width and height of the JFrame to 300 and 200 pixels, respectively.
4. `JPanel panel = new JPanel();`: A JPanel is another type of container that can be placed inside other containers (like JFrames). You can add components to a JPanel.
5. `JButton button = new JButton("Click");`: This line creates a JButton, which is a clickable button with the label "Click".
6. `button.addActionListener(new ActionListener() {...`: This is where an action listener is added to the button. An action listener is a piece of code that is triggered (or "listened to") when the button is clicked.
    - Within this action listener, a counter is incremented each time the button is clicked. The text of the button is then updated to show the number of times it has been clicked.
7. `panel.add(button);`: This line adds the button to the panel.
8. `frame.add(panel);`: This adds the panel (which now contains the button) to the frame.
9. `frame.setVisible(true);`: This line makes the JFrame visible. Until this line is called, the JFrame won't actually show up on the screen.

Note that the variable `counter` isn't defined in this code snippet. For the code to work as intended, you would need to define `counter` somewhere in the scope of the ActionListener, typically as a class variable or final local variable in case of anonymous class.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Main {
    static int counter;
    public static void main(String[] args) {

        JFrame frame = new JFrame("My App");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        frame.setSize(300, 200);
        JPanel panel = new JPanel();
        JButton button = new JButton("Click");

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                counter++;
                String text = "Button was clicked: "+counter+" Times";
                button.setText(text);
            }
        });


        panel.add(button);
        frame.add(panel);

        frame.setVisible(true);
    }
}
```

💡 IMPORTANT HERIARCHY

```
JFrame
  └── JPanel
          ├── JButton
```

1. `JFrame` is a top-level container. This means it represents a window in your application. A JFrame has features you'd expect from a window: you can minimize it, maximize it, close it, and it typically has a title bar where you can set the name of your application. It's the outermost container and you usually only have one JFrame per application (though you can have more if necessary).
2. `JPanel` is a lighter-weight container that can exist inside a JFrame (or inside other containers). It's typically used to group related components together. For example, if you have a form with several text fields and labels, you might put all of them in one JPanel. This can make it easier to organize your GUI and handle component layout.
3. A `JButton` is a GUI component in Java's Swing library that represents a clickable button. It can be used to trigger actions when it's clicked by the user.

Essentially, JFrames are used as the primary windows of your application, while JPanels are used for organization and layout within those windows.

▼ ***JFrames and GUI properties***

▼ *CODE*

```
import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

public class keyboardTest {
    Integer xpos = 128;
    Integer ypos = 128;
    Integer speed = 4;
    void go(){
        JFrame frame = new JFrame("Image mover");
        frame.setSize(512, 512);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        JLabel image = getImage("src/images/turing.jpg", 128, 128);
        image.setBounds(xpos, ypos, 128, 128);
        frame.add(image);

        //JLabel image1 = getImage("src/images/rachmaninoff.jpg", 128, 128);
        //image1.setBounds(300, 100, 128, 128);
        //frame.add(image1);
        frame.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                char pressedKey = e.getKeyChar();
                System.out.println(pressedKey);
                if (pressedKey == 'a'){
                    xpos -= speed;
                    image.setBounds(xpos, ypos, 256, 256);
                }
                else if (pressedKey == 'd'){
                    xpos += speed;
                    image.setBounds(xpos, ypos, 256, 256);
                }
                else if (pressedKey == 'w'){
                    ypos -= speed;
                    image.setBounds(xpos, ypos, 256, 256);
                }
                else if (pressedKey == 's'){
                    ypos += speed;
                    image.setBounds(xpos, ypos, 256, 256);
                }

                // super.keyPressed(e);
            }
        });

        frame.setVisible(true);
    }

    JLabel getImage(String path, int width, int height){
        try {
            BufferedImage brImage = ImageIO.read(new File(path));

            Image sampleImage = brImage.getScaledInstance(width, height, Image.SCALE_SMOOTH);

            return new JLabel(new ImageIcon(sampleImage));

        }catch (IOException e){
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args) {
        keyboardTest keyboardTest = new keyboardTest();
        keyboardTest.go();
    }
}
```

Overview

The provided Java code is a demonstration of basic GUI development with `JFrame` and `JLabel` in Swing. It provides a simple mechanism to load an image into the application window and move it around with keyboard inputs. It also implements basic error handling for file input/output exceptions.

Code Breakdown

Package and Import Statements

The `javax.swing` package is used for building graphical user interfaces (GUIs) in Java. It includes the classes `JFrame` and `JLabel`. The `java.awt` and `java.awt.event` packages are used for event handling and GUI management, while `javax.imageio.ImageIO` and `java.awt.image.BufferedImage` are used for image processing.

Main Class and Method

The main class, `keyboardTest`, has a `go` method which initializes the GUI and adds behavior.

The go() Method

This method creates a `JFrame` and loads an image from the local file system using `getImage()`. The image is then added to the frame as a `JLabel`, and a key listener is set up on the frame to change the image's position based on the key press events.

The getImage() Method

This method takes a path to an image file and the desired width and height of the image as parameters. It uses `ImageIO.read(new File(path))` to load the image, scales it to the desired size using `getScaledInstance()`, and then wraps it in a `JLabel` before returning it.

Keyboard Listener

This is an implementation of a `KeyAdapter`, an abstract adapter class for receiving keyboard events. The `keyPressed` method is overridden to specify the actions to be taken when the keys 'w', 'a', 's', and 'd' are pressed. These keys change the position of the image on the screen.

How to Use

To use this program, you need an image file located at "src/images/turing.jpg". When you run the program, a JFrame window will open displaying this image. You can then use the 'w', 'a', 's', and 'd' keys to move the image around the frame.

Considerations and Limitations

- Make sure that the image path is correctly set and the image file is accessible; otherwise, an IOException may be thrown.
- The size and position of the image are hardcoded in this example, and the window size is fixed. In a more complex application, you would likely need to handle different sizes and aspect ratios, and allow for resizable windows.

Possible Extensions

This basic framework can be extended to handle more complex interactions, multiple images, animations, etc. For instance, you could add additional listeners to handle mouse events, or additional keys to rotate or resize the image.

▼ ***Action Listerners and GUI extends Animation***

▼ ***Mouse Event Listeners***

▼ **CODE**

```
import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

public class MouseTest {
    void go(){
        JFrame frame = new JFrame("Image mover");
        frame.setSize(512, 512);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        JLabel image = getImage("src/images/turing.jpg", 128, 128);
        image.setBounds(128, 128, 128, 128);
        frame.add(image);

        //action listening and moving on action listening features
        // event based
        frame.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                int x = e.getX()-8;
                int y = e.getY()-31;
                image.setBounds(x-128, y-128, 256, 256);
                System.out.println("X: "+x + "\nY: "+y);
            }
        });

        // Motion detection and following with image
        frame.addMouseMotionListener(new MouseAdapter() {
            @Override
            public void mouseMoved(MouseEvent e) {
                int x = e.getX()-8;
                int y = e.getY()-31;
                image.setBounds(x-128, y-128, 256, 256);

                super.mouseMoved(e);
            }
        });

        frame.setVisible(true);
    }

    JLabel getImage(String path, int width, int height){
        try {
            BufferedImage brImage = ImageIO.read(new File(path));

            Image sampleImage = brImage.getScaledInstance(width, height, Image.SCALE_SMOOTH);

            return new JLabel(new ImageIcon(sampleImage));

        }catch (IOException e){
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args) {
        MouseTest mouseTest = new MouseTest();
        mouseTest.go();
    }
}
```

Overview

The provided Java code expands on the principles of GUI development with `JFrame` and `JLabel` in Swing by adding interactivity using mouse events. It offers a simple mechanism to load an image into the application window and move it around based on the mouse actions.

Code Breakdown

Package and Import Statements

Like the previous example, this code relies on the `javax.swing`, `java.awt`, `java.awt.event`, and `javax.imageio.ImageIO` packages for GUI development, event handling, and image processing.

Main Class and Method

The main class, `MouseTest`, contains a `go()` method that initializes the GUI and adds behavior.

The go() Method

This method creates a `JFrame` and loads an image from the file system using `getImage()`. The image is then added to the frame as a `JLabel`, and a mouse listener as well as a mouse motion listener are set up on the frame. These listeners change the image's position based on the mouse click and mouse move events.

The getImage() Method

Similar to the previous code, this method reads an image file, scales it to the given width and height, wraps it in a `JLabel` and then returns it.

Mouse Listener

A `MouseAdapter` is used here to handle mouse events. The `mouseClicked` method is overridden to change the image's position when the mouse is clicked on the frame.

Mouse Motion Listener

Another `MouseAdapter` is used to handle mouse motion events. The `mouseMoved` method is overridden to change the image's position as the mouse cursor moves around the frame.

How to Use

Ensure you have an image file at the specified path, "src/images/turing.jpg". When you run the program, a JFrame window opens displaying the image. You can then move the image by clicking on the window or by moving the mouse cursor around the window.

Considerations and Limitations

- Make sure the image path is correct and accessible to prevent IOExceptions.
- The position of the image is calculated by subtracting the border and title bar offsets from the reported mouse event coordinates. This code may not work correctly if the JFrame style, border size, or title bar size is changed.
- The window size is fixed. You might need to adjust the code to allow for resizable windows.

Possible Extensions

You can extend this code to create more sophisticated interactions. For example, you could add drag-and-drop functionality, allow resizing or rotation of the image, or handle additional mouse events like right-clicks or mouse wheel movements.

▼ *Animations*

▼ *CODE*

```
import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

public class AnimationTest {
    void go(){
        JFrame frame = new JFrame("Image mover");
        frame.setSize(512, 512);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        int x = 0;
        int y = 128;
        int speed = 4;
        int direction = 1;
        JLabel image = getImage("src/images/turing.jpg", 128, 128);
        image.setBounds(128, 128, 128, 128);
        frame.add(image);

        frame.setVisible(true);
        for (int i = 0; i < 300; i++) {
            x = x + (speed * direction);
            image.setBounds(x, y, 256, 256);
            if (x + 256 >= 512) direction = -1;
            if (x <= 0) direction = 1;
            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        frame.setVisible(true);
    }

    JLabel getImage(String path, int width, int height){
        try {
            BufferedImage brImage = ImageIO.read(new File(path));
            Image sampleImage = brImage.getScaledInstance(width, height, Image.SCALE_SMOOTH);

            return new JLabel(new ImageIcon(sampleImage));

        }catch (IOException e){
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args) {
        AnimationTest animationTest = new AnimationTest();
        animationTest.go();
    }
}
```

Overview

The provided Java code demonstrates a simple image animation within a JFrame GUI. It loads an image into the application window and moves it horizontally across the frame, bouncing back when it hits the frame boundaries.

Code Breakdown

Package and Import Statements

This code, like the previous examples, uses the `javax.swing`, `java.awt`, `java.awt.event`, and `javax.imageio.ImageIO` packages for GUI development and image processing.

Main Class and Method

The main class, `AnimationTest`, contains a `go()` method which initializes the GUI and handles the animation.

The go() Method

This method creates a `JFrame`, loads an image from the file system using the `getImage()` method, and adds the image to the frame as a `JLabel`. The frame is set visible before a loop starts which animates the image, moving it horizontally and changing direction when it hits a boundary. The speed and direction of the movement are controlled by the `speed` and `direction` variables.

The thread is made to sleep for 50ms between each frame of the animation using `Thread.sleep(50)`, which creates the illusion of smooth movement.

The getImage() Method

Similar to the previous examples, this method reads an image file, scales it to the desired dimensions, wraps it in a `JLabel`, and returns it.

How to Use

Ensure you have an image file at the specified path, "src/images/turing.jpg". When you run the program, a JFrame window will open and the image will start moving horizontally across the frame, bouncing back at the edges.

Considerations and Limitations

- The image path must be correct and accessible to prevent IOExceptions.
- The speed of the animation and the size of the frame and image are hardcoded in this example. Adjust these values as needed.
- This simple animation doesn't handle user interactions or changes to the window size.

Possible Extensions

You can extend this code to create more complex animations, handle user interactions, or adjust the animation based on the size of the window. For instance, you could add keyboard or mouse event listeners to control the animation, or modify the code to resize the image or adjust the speed based on the size of the JFrame.

▼ ***OMOK GAME PROJECT***

Gomoku Console Game Documentation

This document describes the structure and functionality of a console-based Gomoku game, also known as Omok. The game comprises several classes and interfaces that manage the game logic, player actions (both human and AI), and the game board.

Table of Contents

Introduction

Gomoku, or Omok, is an abstract strategy board game. The goal is to line up five stones of your color in a row, horizontally, vertically, or diagonally. This document covers the implementation of a console-based version of Gomoku, detailing the classes and interfaces involved.

Player Interface

```
public interface Player {
    int[] move = new int[2];
    int[] makeMove(Stone[][] board);
    Stone getStoneType();
}
```

Methods:

- `int[] makeMove(Stone[][] board);`

  Gets the player's next move.

- `Stone getStoneType();`

  Retrieves the type of stone the player is using.

Stone Enum

```
public enum Stone {
    EMPTY,
    BLACK,
    WHITE
}
```

Description:

Represents the possible states of a space on the game board: empty, occupied by a black stone, or occupied by a white stone.

HumanPlayer Class

```
public class HumanPlayer implements Player {
    private Stone stoneType;

    public HumanPlayer(Stone stoneType) {
        this.stoneType = stoneType;
    }

    @Override
    public int[] makeMove(Stone[][] board) {
        // Implementation for getting human player's move...
    }

    @Override
    public Stone getStoneType() {
        return stoneType;
    }
}
```

Methods:

- `public HumanPlayer(Stone stoneType);`

  Constructor specifying the stone type for the human player.

- `public int[] makeMove(Stone[][] board);`

  Retrieves and validates the human player's move.

- `public Stone getStoneType();`

  Gets the type of stone the player uses.

Board Class

```
public class Board {
    private static final int BOARD_SIZE = 15;
    private Stone[][] grid = new Stone[BOARD_SIZE][BOARD_SIZE];

    public void init_board() {
        // Implementation...
    }

    public void renderBoard() {
        // Implementation...
    }

    // Other methods...
}
```

Methods:

- `public void init_board();`

  Initializes the game board to a starting state.

- `public void renderBoard();`

  Prints the current state of the game board to the console.

- `public void clearScreen();`

  Clears the console screen.

- `public Stone[][] getGrid();`

Getter for the current state of the game board.

- `public int getSize();`

  Retrieves the size of the game board.

- `public void setGrid(Stone[][] board);`

  Setter for the current state of the game board.

AIPlayer Class

```
public class AIPlayer implements Player {
    private Board board = new Board();
    private static final int MAX_DEPTH = 2;
    private Stone stoneType;

    // Constructors and methods...
}
```

Methods:

- `public AIPlayer(Stone stoneType, Stone[][] currentBoard);`

  Constructor that specifies the stone type and current state of the game board for the AI.

- `public int[] bestMove();`

  Determines the best possible move using the minimax algorithm.

- `private int minimax(int depth, Stone player, int alpha, int beta);`

  Core AI strategy using a recursive minimax algorithm with alpha-beta pruning.

- `private boolean gameIsOver();`

  Checks if the game has ended based on the current state of the board.

- `private int evaluateBoard();`

  Scores the current state of the game board from the AI's perspective.

- `private int evaluatePosition(int x, int y, int dx, int dy);`

  Scores a specific position on the game board.

- `public int[] makeMove(Stone[][] board);`

  Executes the best move determined by the AI.

- `public Stone getStoneType();`

  Retrieves the type of stone the AI player uses.

Overview

This Gomoku project is a console-based version of the traditional board game, often known as "Five in a Row," where the objective is for a player to line up five of their stones in a row, whether horizontally, vertically, or diagonally, on a 15x15 grid board. The project is developed in Java and it simulates the game allowing two players to play against each other, one of them being a human and the other an AI (Artificial Intelligence).

Components

The project consists of several main components: the game board, the players (human and AI), and the stones. Here's how each component fits into the overall game:

1. Game Board (`Board` Class):

- The `Board` class represents the game board. It's responsible for initializing the board, rendering the board state to the console, and managing the state of the board (which positions are occupied by which stones).
- The board is a 15x15 grid, represented internally as a 2D array of `Stone` enums, which can have a value of `EMPTY`, `BLACK`, or `WHITE`.

2. Players (`HumanPlayer` and `AIPlayer` Classes):

- The `HumanPlayer` and `AIPlayer` classes both implement the `Player` interface, allowing them to be used interchangeably in the game logic. This design makes it easy to extend the game with new types of players in the future.
- The `HumanPlayer` class handles input from a human player, allowing them to enter their moves via the console.
- The `AIPlayer` class contains logic for the computer-controlled player. It uses the minimax algorithm (a decision rule for minimizing the possible loss for a worst-case scenario) to determine its moves. This class is more complex, as it contains the logic for evaluating the current board state, predicting future moves, and choosing the optimal move.

3. Stones (`Stone` Enum):

- The `Stone` enum represents the possible states of a position on the board: `EMPTY` (no stone), `BLACK` (occupied by a black stone), or `WHITE` (occupied by a white stone).

4. Gameplay Logic (`OmokGameConsole` Class):

- The `OmokGameConsole` class drives the game, handling the game loop and the players' turns. It checks for win conditions, player moves, and the end of the game.

Flow of the Game

1. **Initialization:**
    - The game initializes with the creation of the `OmokGameConsole` instance, setting up the board and the players. The human player always goes first, using black stones.
2. **Game Loop:**
    - The game enters a loop that continues until a win condition is met (one player gets five stones in a row).
    - On each loop, the board is rendered to the console, and the current player is prompted to make a move.
        - If the player is human, they enter their move via the console.
        - If the player is the AI, it calculates the best move using the minimax algorithm.
    - The game checks if the move results in a win condition. If it does, the game ends and a winner is declared. If not, the next player takes their turn.
3. **End of Game:**
    - Once a player wins, the game loop breaks, and the game ends, declaring the winner.

This project is a simplified simulation of the real-world game, aimed at providing an interactive and competitive experience for the user, combining traditional Gomoku rules and AI algorithms.