TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

T-Rex Runner Game

Student: Alexandra Ilovan
Coordinating professor: Bogdan Nicușor Bindea
Computer Science, Automation and Computer Science
Group 30422, second year of study

Contents

1. Introduction

1.1. Short Description

This application is a reconstruction of the renown Dinosaur Game, which can be played when browsing offline on Google Chrome.

Task: Consider implementing the Chrome Dinosaur Game in the form of a Desktop application, using Java Swing, instead of a Web application.

This version of the T-Rex Runner Game is integrated in a 5-view application that contains the following: a *game-menu window*, a *game-rules window*, a *player-ranking window*, a *window designed for registering a new player* and the *window of the game itself*.

The T-Rex Runner Game is an *endless runner game*, that is, a game in which the player must dodge obstacles as they automatically and continuously scroll onto the screen, with the simple goal of not crashing into anything for as long as possible. In this case, the player controls a running T-Rex as it tries to dodge cacti and flying pterodactyls, and receives points according to the type of obstacle that has been successfully avoided. Once the dinosaur crashes into one of the obstacles, the dinosaur dies and the game ends. The game can be played infinitely, and for each try the score and high-score obtained by the current player will be saved and updated.

1.2. Used Technologies

The application was implemented using Java and Java Swing, and the code was written in IntelliJ IDEA.

Java is a high-level, class-based, object-oriented, general-purpose programming language, that has been designed such that it has as few implementation dependencies as possible. The syntax of Java is similar to that of C and C++, but is has fewer low-level facilities than either of them.

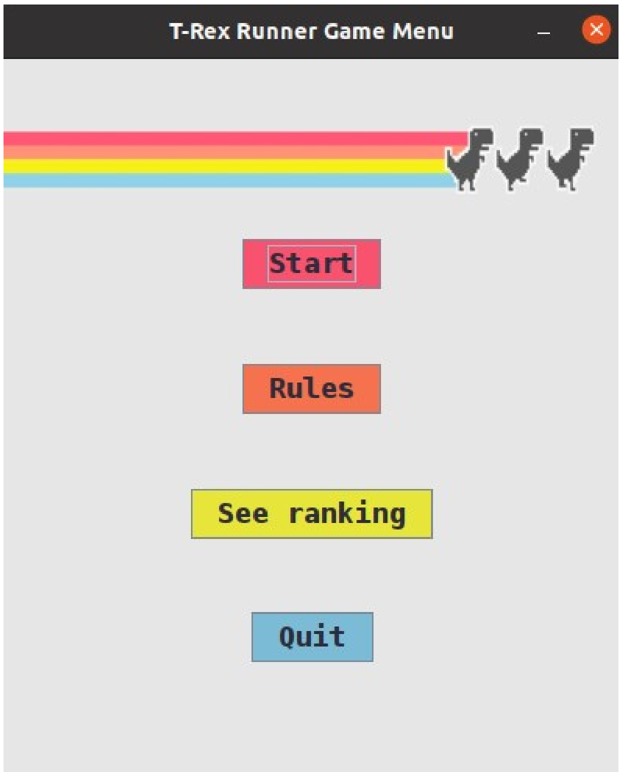
Java Swing is a GUI widget toolkit for Java. It is part of Oracle's Java Foundation Classes (JFC), and it is used to create window-based applications. It is built on top of the AWT (Abstract Windowing Toolkit) API and entirely written in Java.

IntelliJ IDEA, developed by *JetBrains*, is an integrated development environment (IDE) written in Java for developing computer software. It was released in January 2001 and was one of the first available Java IDEs that allowed advanced code navigation and had code refactoring capabilities integrated.

2. GUI

The Graphical User Interface of the application consists of 5 views, and in the following lines each view and its functionality will be described.

When running the application, the following view appears:



This window represents the menu of the game, and it has the following features:

- **Start** button: when pressed, it opens a new window that with player-registration purposes

- **Rules** button: it opens a new window that displays the rules of the game and the controls needed to move the T-Rex character

- **See ranking** button: it opens a window that displays a table showing information about all of the games that have been played so far

- **Quit** button: exits the application

Note: The window is non-resizable (it has a fixed size). The main components used in the making of this window were JButtons.

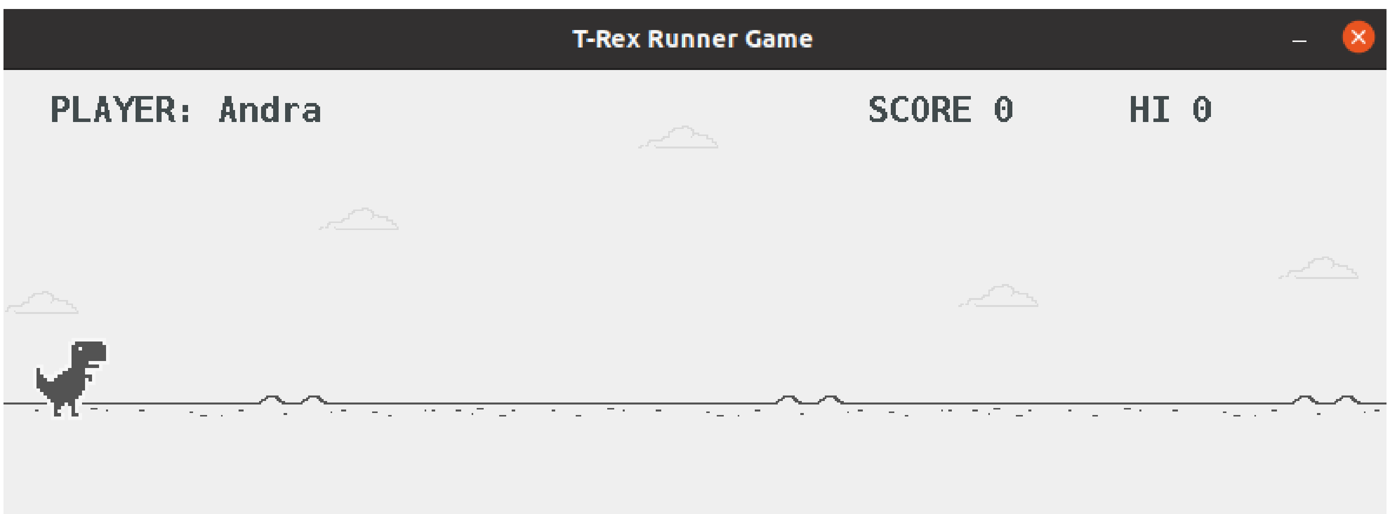
Upon pressing the Start button, the menu window closes and the player-registration window opens in its place:

This window contains a JLabel that prompts the player to input their name, a JTextField, in which the name can be entered and 2 JButtons: one that redirects the player to the Menu (**Back**) and one that opens the window of the actual game (**Next**).

In case the entered text is too long (has more than 20 characters), a warning message will be displayed in red text (it is actually a JLabel that is made visible), and in the case in which the player tries to press the **Next** button without entering their name, a dialog of an error message will pop up.

After hitting **OK**, the user can try to enter their name again, without the application breaking.

If the entered name is valid and the user presses the **Next** button, the current window will be disposed and the following one will open:



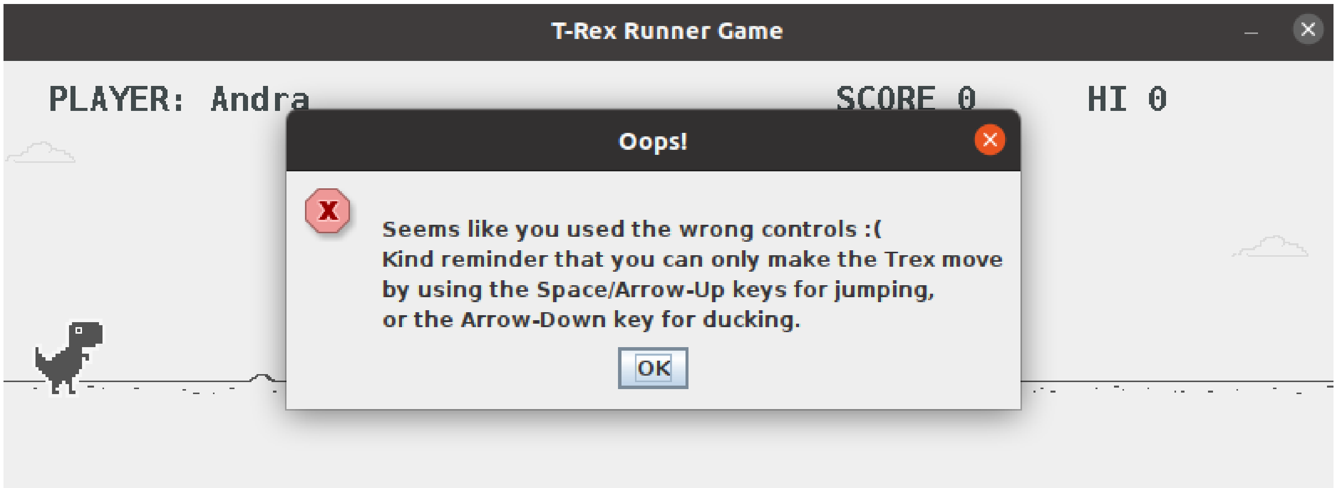
At the beginning, the score and high-score (HI) will be both set to 0 and once the user hits the Space bar or the Arrow-Up key, the game will begin, and the score will increase by 25 for each cactus passed, and by 30 for each pterodactyl that has been avoided. The player's name, score and high-score, along with all the other game elements (the T-Rex, the ground, the clouds and the obstacles) were all drawn onto the JPanel by using a Graphics object.

Here are a few screen prints of the playthrough:

This is what it looks like when the T-Rex dies and the game ends:

In order to restart the game, the player can either click the button below the “GAME OVER” text or hit the Space bar.

If any other key besides the Space bar, Arrow-Up and Arrow-Down keys is pressed, the following error message will appear and the game will end:

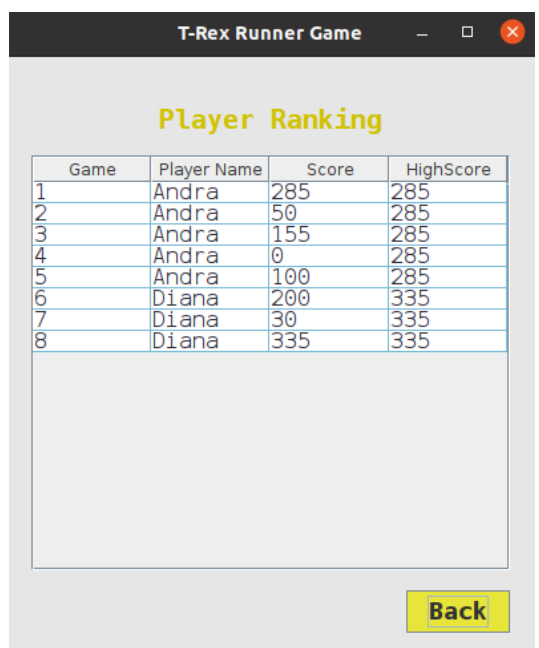


Once the gaming window is closed, the user is redirected to the Menu window.

If the user presses the **Rules** button, the following window will be shown:

This window presents the rules of the game, written inside a JTextArea component.

A **Back to Menu** button is also provided so that the user can return to the menu and play the game after reading the rules.



The application also allows the user to see a ranking of all the games played so far, and it does so if the **See Ranking** button is pressed.

This window provides a JTable in which the number of the games, the players and their corresponding scores and highscores are displayed.

By pressing the **Back** button, the user will be taken back to the Menu.

3. Implementation

3.1. Application Structure

The application consists of five packages (six including the **src** package) that are structured as follows:

- **src**
  - **game**
    - game.**managers**
    - game.**objects**
  - **utilities**
  - **view**

UML diagram:

3.2. Code Breakdown

In the lines that follow I will be presenting briefly each package along with its contents and add a few explanations of the more relevant blocks of code.

1. The **game** package

```
game
  managers
    ObstacleManager
    RankingManager
  objects
    Cactus
    Ground
    Obstacle
    Player
    Pterodactyl
    Sky
    Trex
    TrexState
```

The game package contains all the elements that make up the model of the application. The package is divided in turn into two other packages:
- **game.objects**: contains all the main objects of the game > **Player**, **Trex**, obstacle-type-objects (**Cactus**, **Pterodactyl** – both extend the **Obstacle** class), landscape-type-objects (**Ground**, **Sky**)
- **game.managers**: the classes inside this package have the role of managing the objects from the game.objects package

There are two managers, one that manages the obstacles of the game (**ObstacleManager**), and one that manages the players and ranks their scores (**RankingManager**).

The **Obstacle** abstract class

```java
public abstract class Obstacle {

    public abstract Rectangle getObjectBounds();    // returns the bounds of the obstacle as a rectangle
    public abstract void drawObstacle(Graphics graphics);   // draws the obstacle
    public abstract void updateObstaclePos();   // updates the position of the obstacle in order for it to be redrawn
    public abstract boolean obstacleIsOutOfBounds();    // checks if an obstacle is out of the screen bounds
    public abstract boolean obstaclePassed();   // checks if the trex passed the given obstacle
    public abstract boolean scoreWasAssigned();    // returns true if the score was assigned to the player
    public abstract void setScoreAssignation(boolean scoreWasAssigned);     // scoreWasAssigned = true, if the score was
                                                                            // assigned and false otherwise
}
```

The **Cactus** class – extends <u>Obstacle</u>

```java
    // Overriding Obstacle methods
    @Override
    public Rectangle getObjectBounds() { return cactusRect; }

    @Override
    public void drawObstacle(Graphics graphics)
    {
        graphics.drawImage(cactusImg, x, y, width: cactusImg.getWidth() / 2 , height: cactusImg.getHeight() / 2,
                observer: null);
    }
```

```java

    @Override
    public void updateObstaclePos()
    {
        x -= 2;

        cactusRect.x = this.x;
        cactusRect.y = this.y;
        cactusRect.width = cactusImg.getWidth() / 2;
        cactusRect.height = cactusImg.getHeight() / 2;
    }

    @Override
    public boolean obstacleIsOutOfBounds()
    {
        if (x + cactusImg.getWidth()/2 < 0)
            return true;

        return false;
    }

    @Override
    public boolean obstaclePassed()
    {
        if (trex.getX() > x)   // checks if the trex passed the cactus
            return true;

        return false;
    }
```

```java
    @Override
    public boolean scoreWasAssigned()
    {
        return scoreWasAssigned;
    }

    @Override
    public void setScoreAssignation(boolean scoreWasAssigned)
    {
        this.scoreWasAssigned = scoreWasAssigned;
    }
}
```

The **Pterodactyl** class – extends Obstacle

In order to draw a pterodactyl object on the panel, an animation is needed since a pterodactyl is not a static obstacle, like a cactus is.

The constructor of the Pterodactyl class looks like this:

```java
public class Pterodactyl extends Obstacle {

    private int x, y;   // coordinates of the position of the pterodactyl
    private Rectangle pterodactylRect;  // rectagle that determines the bounds of the pterodactyl
    private Animation pterodactylAnimation;
    private Trex trex;
    private boolean scoreWasAssigned = false;

    public Pterodactyl(Trex trex)
    {
        // creating the pterodactyl flying-animation
        pterodactylAnimation = new Animation(200);
        pterodactylAnimation.addFrame(Resources.getResourceImg("resources/original/pterodactyl1.png"));
        pterodactylAnimation.addFrame(Resources.getResourceImg("resources/original/pterodactyl2.png"));

        pterodactylRect = new Rectangle();
        this.trex = trex;
    }
```

The overriding of the Obstacle methods:

```java
    @Override
    public Rectangle getObjectBounds()
    {
        return pterodactylRect;
    }

    @Override
    public void drawObstacle(Graphics graphics)
    {
        graphics.drawImage(pterodactylAnimation.getFrame(), x, y, pterodactylAnimation.getFrame().getWidth()/2,
                pterodactylAnimation.getFrame().getHeight()/2, null);
    }

    @Override
    public void updateObstaclePos()
    {
        x -= 2;
        pterodactylRect.x = x;
        pterodactylRect.y = y;
        pterodactylRect.width = pterodactylAnimation.getFrame().getWidth() / 2;
        pterodactylRect.height = pterodactylAnimation.getFrame().getHeight() / 2;
        pterodactylAnimation.updateFrame();
    }

    @Override
    public boolean obstacleIsOutOfBounds()
    {
        if (x + pterodactylAnimation.getFrame().getWidth()/2 < 0)
            return true;

        return false;
    }
```

```java
    @Override
    public boolean obstaclePassed()
    {
        if (trex.getX() > x)  // if the trex's x pos is larger than the pterodactyl's x pos it means that
            return true;      // they have passed by each other

        return false;
    }

    @Override
    public boolean scoreWasAssigned() { return scoreWasAssigned; }

    @Override
    public void setScoreAssignation(boolean scoreWasAssigned) { this.scoreWasAssigned = scoreWasAssigned; }
}
```

The **Sky** class - contains a nested class **Cloud** (whose fields are the positions of the cloud), and it creates a list of clouds that are then drawn in random positions of the game panel

```java
public class Sky {

    private class Cloud {
        double x, y;    // the coordinates of the cloud
    }

    private BufferedImage cloudImg;
    private ArrayList<Cloud> cloudList;

    public Sky()
    {
        cloudImg = Resources.getResourceImg("resources/original/cloud.png");
        cloudList = new ArrayList<Cloud>();

        // generate 5 clouds positioned randomly at equal distances
        for (int i = 0; i < 5; ++i)
        {
            Cloud newCloud = new Cloud();
            newCloud.x = i * 2 * cloudImg.getWidth();
            generateRandomCloudPosition(newCloud);
        }
    }

    // method that adds a cloud with a random position on the y-axis to the list
    private void generateRandomCloudPosition(Cloud randCloud)
    {
        Random randomPos = new Random();
        randCloud.y = randomPos.nextDouble(20, 130);

        cloudList.add(randCloud);
    }
```

The **updateSky()** method is the most important method of this class, since this is the method that facilitates the redrawal of the clouds in the correct positions, giving the sky its dynamicity.

```java
    // method that updates the position of the clouds on the sky(changes their x coordinate to make it look like they're moving)
    public void updateSky()
    {
        Iterator<Cloud> it = cloudList.iterator();
        Cloud firstCloud = it.next();
        firstCloud.x -= 0.5f; // start shifting the position of the first cloud in the list

        // while there are still clouds in the list, we shift each one's position by 0.5f to the left of the screen
        while (it.hasNext())
        {
            Cloud nextCloud = it.next();
            nextCloud.x -= 0.5f;
        }

        // if the x coordinate of the first cloud in the list is bigger than the width of the cloud image (aka the cloud has
        // moved out of the screen bounds)
        if (firstCloud.x < -cloudImg.getWidth())
        {
            cloudList.remove(firstCloud);   // remove the cloud from the list
            firstCloud.x = WINDOW_WIDTH;    // set the x-pos of the cloud back to the right side of the window
            generateRandomCloudPosition(firstCloud);    // and add a new randomly-generated cloud to the list so that
                                                        // the cloud will never stop appearing on the sky
        }
    }
}
```

The **Ground** class is similar to the **Sky** class, in that it also has a nested class **GroundImage**, a method that generates random types of ground and then it assign objects of type GroundImage to it, and an update method.

```java
public class Ground {

    private class GroundImage {
        double x;   // the x coordinate of the ground object
        BufferedImage image;
    }

    private BufferedImage ground1, ground2, ground3;   // there are 3 types of grounds that can be generated
    private ArrayList<GroundImage> imgList;

    public Ground()
    {
        imgList = new ArrayList<GroundImage>();

        ground1 = Resources.getResourceImg( path: "resources/original/ground1.png");
        ground2 = Resources.getResourceImg( path: "resources/original/ground2.png");
        ground3 = Resources.getResourceImg( path: "resources/original/ground3.png");

        // we imagine that the default size of an image is the size of the 3rd ground image (because that is the smallest
        // out of the three - I failed at cropping them equally :) and it SHOWS)
        int imgSize = WINDOW_WIDTH / ground3.getWidth() + 2;   // the amount ground-images that fit onto the screen

        // generate images of the random types of ground and add them to the image list
        for (int i = 0; i < imgSize; ++i)
        {
            GroundImage gi = new GroundImage();
            gi.x = i * ground3.getWidth();
            setGroundImage(gi);
            imgList.add(gi);
        }
```

```java
public void updateGround()
{
    Iterator<GroundImage> it = imgList.iterator();
    GroundImage firstPart = it.next();
    // by decreaing the x coordinate, we create the illusion that the ground is moving (and combined with the movement
    // of the trex's legs, we get the illusion that the trex is running continuously)
    firstPart.x--;
    double prevX = firstPart.x;

    // while there are still images in the list, we keep updating their x-positions
    while (it.hasNext())
    {
        GroundImage img = it.next();
        img.x = prevX + ground3.getWidth();
        prevX = img.x;
    }

    // if the first image of the ground in the list is greater than its width(the first image is out of the screen)
    if (firstPart.x < -ground3.getWidth())
    {
        imgList.remove(firstPart);  // we remove the image from the list
        firstPart.x = prevX + ground3.getWidth();
        setGroundImage(firstPart);  // and add a new one
        imgList.add(firstPart);
    }
}
```

The **Trex** class has four methods designated for drawing the Trex in different instances, a method that gets the bounds of these drawings, a method that is called when the Trex jumps, and most importantly, an update method.

```java
// methods that draw the Trex on the panel in all its different states
@
public void drawRunningTrex(Graphics graphics)
{
    graphics.drawImage(trexRunningAnimation.getFrame(), (int)x, (int)y , width: trexRunningAnimation.getFrame().getWidth()/2,
            height: trexRunningAnimation.getFrame().getHeight()/2, observer: null);
}

@
public void drawJumpingTrex(Graphics graphics)
{
    graphics.drawImage(trexJumpingIcon, (int)x, (int)y, width: trexJumpingIcon.getWidth() / 2,
            height: trexJumpingIcon.getHeight() / 2, observer: null);
}

@
public void drawDuckingTrex(Graphics graphics)
{
    graphics.drawImage(trexDuckingAnimation.getFrame(), (int)x, (int)(y + 17), trexDuckingAnimation.getFrame().getWidth(),
            trexDuckingAnimation.getFrame().getHeight(), observer: null);
}

@
public void drawDeadTrex(Graphics graphics)
{
    graphics.drawImage(trexDeadIcon, (int)x, (int)y, width: trexDeadIcon.getWidth() / 2,
            height: trexDeadIcon.getHeight() / 2, observer: null);
}
```

```java
    // method that redraws the trex and deals with the situations when the Trex is out of bounds(out of the window)
    public void updateTrexMovements()
    {
        // updating the animation frames
        trexRunningAnimation.updateFrame();
        trexDuckingAnimation.updateFrame();

        // if the Trex goes under the ground level
        if (y >= GROUND - trexRunningAnimation.getFrame().getHeight() / 2)
        {
            speedY = 0;
            y = GROUND - trexRunningAnimation.getFrame().getHeight() / 2;
        }
        else
        {
            speedY += GRAVITY;
            y += speedY;
        }

        // setting the upper bounds that the trex can reach when jumping
        if (y <= SKY)
            y = SKY;

        // configuring the rectangle sizes
        trexRectUp.x = (int)x;
        trexRectUp.y = (int)y;
        trexRectUp.width = trexJumpingIcon.getWidth() / 2;
        trexRectUp.height = trexJumpingIcon.getHeight() / 2;

        trexRectDown.x = (int)x;
        trexRectDown.y = (int)y;
        trexRectDown.width = trexDuckingAnimation.getFrame().getWidth();
        trexRectDown.height = trexDuckingAnimation.getFrame().getHeight();
```

```java
    // method that changes the position of the trex on the y-axis when jumping
    public void jump()
    {
        // for a jump (one hit of a space/arrow-up key) the trex's position on the y-axis diminishes by 3 pixels
        speedY = -3;
        y += speedY;
    }

    // method that returns the bounds of the trex object
    public Rectangle getObjectBounds()
    {
        if (isDown)
            return trexRectDown;

        return trexRectUp;
    }
}
```

The **Player** class is a simple class that stores the name of the current player and the score obtained in the current game.

```java
public class Player {

    private String playerName;
    private int score;  // the score this player got for ONE game

    // no-parameter Player constructor
    public Player() {}

    // Getters & Setters region
    public String getPlayerName() { return playerName; }

    public void setPlayerName(String playerName) { this.playerName = playerName; }

    public int getScore() { return score; }

    public void setScore(int score) { this.score = score; }
    // end region
}
```

The **RankingManager** class has a significant importance in storing information about the game. It uses an ArrayList for storing the players and their scores. The downside of using an ArrayList for this task, however, is that once the application is closed, all the data is disposed. It has a method for adding players to the list, a method that returns the list with all the data it has gathered so far, a method that returns a list of the scores of one individual player, selected by name and a method that returns a player's high-score.

```java
    // method that returns the ranking list of the players
    public ArrayList<Player> getPlayerRanking() { return rankingList; }

    // method that adds a player to the ranking list
    public void addPLayerToRankingList(Player player) { rankingList.add(player); }

    // method that returns a player's highest score
    public int getPlayerHS(Player player)
    {
        int highScore = -1;
        ArrayList<Player> selectedPlayerScores = getRankingByPlayer(player.getPlayerName());

        // compare all the scores of a player in order to determine the highest one
        for (int i = 0; i < selectedPlayerScores.size(); ++i)
            if (selectedPlayerScores.get(i).getScore() >= highScore)
                highScore = selectedPlayerScores.get(i).getScore();

        return highScore;
    }

    // method that returns a list of all the scores obtained by a single specific player
    private ArrayList<Player> getRankingByPlayer(String playerName)
    {
        ArrayList<Player> selectedPlayers = new ArrayList<>();

        for (int i = 0; i < rankingList.size(); ++i)
            if (rankingList.get(i).getPlayerName() == playerName)
                selectedPlayers.add(rankingList.get(i));

        return selectedPlayers;
    }
}
```

ObstacleManager is yet another class that stays at the base of this application. It manages what type of obstacle will appear next by randomizing the selection process: by using a Random object, it takes an int-type variable that receives a random value from 0 to 9. In the case when the returned value is 5, the next obstacle will be a pterodactyl, while in any other case it will be a cactus. Therefore, if we do the math, it means that the chance for a pterodactyl to appear is of 10%. The reason why I implemented such a low probability for the existence of a pterodactyl-obstacle is because I wanted to mimic the scarcity with which they appear in the original dinosaur game.

```java
    // function that randomizes whether the next obstacle will be a cactus or a pterodactyl
    private void getRandomObstacle()
    {
        randomObstacle = new Random();
        int nextObstacle = randomObstacle.nextInt( bound: 10);

        // Note: in the original game, the pterodactyls appear much less than the cacti
        // since the pterodactyl can appear in 1/10 cases, and the generated numbers are
        // randomized, there will be a 10% probability for the pterodactyl to appear
        switch(nextObstacle) {
            case 5:
                obstacleList.add(getRandomPterodactyl());
                nextObstacleIsPterodactyl = true;
                break;
            default:
                obstacleList.add(getRandomCactus());
                nextObstacleIsPterodactyl = false;
                break;
        }
    }
```

There is also a method that resets all the obstacles, meaning that they are all removed from the obstacle list once the game ends.

```java
    // method that resets all the obstacles (called when the game ends)
    public void resetObstacles()
    {
        obstacleList.clear();
        getRandomObstacle();
        intersects = false;
    }
```

Obviously, an update method cannot be missing from this class either, and it is explained in the following picture:

```java
public void updateObstacle()
{
    // for each obstacle in the list, we update its x-position
    for (Obstacle obstacle : obstacleList)
    {
        obstacle.updateObstaclePos();

        // if the obstacle has been passed by the trex without collision, and the score was not yet assigned
        if (obstacle.obstaclePassed() && !obstacle.scoreWasAssigned() && !intersects)
        {
            // if obstacle was a pterodactyl, add 30 points
            if (nextObstacleIsPterodactyl)
                panel.increaseScoreBy(30);
            else
                panel.increaseScoreBy(25);  // otherwise add 25

            obstacle.setScoreAssignation(true);   // and set the score as having been assigned
        }

        // if the rectangle of the trex and of the obstacle intersect(if the trex and obstacle have a collision)
        if (obstacle.getObjectBounds().intersects(trex.getObjectBounds()))
        {
            intersects = true;
            trex.setDead(true);     // then the trex dies
        }
    }

    Obstacle firstObstacle = obstacleList.get(0);
    // if the first obstacle in the list is out of the screen bounds
    if (firstObstacle.obstacleIsOutOfBounds())
    {
        obstacleList.remove(firstObstacle); // remove it and add a new one to the list
        getRandomObstacle();
```

2. The **utilities** package contains two classes: **Animation** and **Resources.**

The **Resources** class contains a public method that is used to retrieve an image from the resources of the project, with a given path.

```java
public class Resources {

    // method used for retrieving images from the Resources of the project
    public static BufferedImage getResourceImg(String path)
    {
        BufferedImage resourceImg = null;

        try
        {
            resourceImg = ImageIO.read(new File(path));
        }
        catch (IOException e)
        {
            e.printStackTrace();
            System.out.println("Image not found.");
        }

        return resourceImg;
    }
}
```

The **Animation** class creates a model that is later used in the animating of the Trex and pterodactyl objects. Its constructor assigns the value of the delta time (which is the amount of time it takes a frame to change) and instantiates a list of all the frames needed for the animation.

```java
public class Animation {

    private ArrayList<BufferedImage> frameList; // list of all the frames of the animation
    private int frameIndex = 0;
    private long prevTime;  // the last time at which a frame has changed
    private int deltaTime;  // the time it takes for a frame to change

    public Animation(int deltaTime)
    {
        this.deltaTime = deltaTime;
        frameList = new ArrayList<>();
    }

```

Its methods are the **addFrame()** method, which adds a frame to the frame list, the **getFrame()** method, which returns a frame, and the **updateFrame()** method, which updates the frames of the animation, as the name suggests.

```java
    // method that adds a frame(of type BufferedImage) to the frame list
    public void addFrame(BufferedImage frame) { frameList.add(frame); }

    // method that returns a frame
    public BufferedImage getFrame()
    {
        if (!frameList.isEmpty())
        {
            return frameList.get(frameIndex);
        }

        return null;
    }

    // method that updates the frames in order to create the animation
    public void updateFrame()
    {
        // if the difference between the current time and the last time a frame changed is greater than the delta time
        if (System.currentTimeMillis() - prevTime > deltaTime)
        {
            frameIndex++;  // the frame is updated

            // once the frame index is beyond the total number of frames
            if (frameIndex >= frameList.size())
                frameIndex = 0;   // we set it back to 0 (the first frame in the list)

            prevTime = System.currentTimeMillis();
        }
    }
```

3. The **view** package contains the classes that create all the windows of the application and they can be seen in the image attached:

- view
  - GamePanel
  - GameView
  - RunApplication
  - MenuView
    - MenuView
    - MenuView.form
  - NewPlayerView
    - NewPlayerView
    - NewPlayerView.form
  - RankingView
    - RankingView
    - RankingView.form
  - RulesView
    - RulesView
    - RulesView.form

The **GameView**, **MenuView**, **NewPlayerView**, **RankingView** and **RulesView** classes are all pretty similar, mainly containing the constructors of the windows and action listeners for all the components inside them. The files with the same names that have the .form extension are the GUI designers generated by Java Swing, in which most of the appearance functions were configured.

In the following lines, I will be focusing in more depth on the **GamePanel** class, since that is where most of the game functionalities were implemented.

RunApplication is the class that contains the main() method.

All the important objects used in the making of the game are instantiated in the GamePanel constructor as follows:

```java
public GamePanel(String playerName)
{
    thread = new Thread( target: this);  // turn the panel into a thread

    this.playerName = playerName;

    // initializing the game objects
    trex = new Trex();
    groundObj = new Ground();
    sky = new Sky();
    obstacleManager = new ObstacleManager(trex,  gPanel: this);

    this.setLayout(new GridBagLayout());
    this.setAlignmentY(100);

    gameOverImg = Resources.getResourceImg( path: "resources/original/gameOver.png");
    restartImg = Resources.getResourceImg( path: "resources/original/restart2.png");
    restartBtn = new JButton(new ImageIcon(restartImg));

    showRestartButton();
}
```

Since this game is an endless runner type of game, the implementation of the Runnable interface was needed in order to make the panel run continuously. Here is the overriding of the run() method from Runnable:

```java
@Override
public void run()
{
    while (true)
    {
        trex.updateTrexMovements();

        if (trex.isDead())
            characterState = TrexState.DEAD;

        // if the Trex is dead, the game will end, which means that the restart button's visibility has to be set to true
        if (characterState == TrexState.DEAD)
        {
            restartBtn.setVisible(true);
            restartBtn.setEnabled(true);
        }

        // if the character state is either one besides the initial or dead states, it means the game is running
        // so all the game objects need to keep on being updated(redrawn)
        if (characterState != TrexState.INITIAL_STATE && characterState != TrexState.DEAD)
        {
            sky.updateSky();
            groundObj.updateGround();
            obstacleManager.updateObstacle();
            restartBtn.setVisible(false);
            restartBtn.setEnabled(false);
            restartBtn.setFocusable(false);
        }

        repaint();
```

The following lines of code need to be written in order to reduce the speed with which the application runs.

```
        try
        {
            thread.sleep(5);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
```

The functionality of the game overall, the component painting, the way it reacted to key events, were all tied to the state of the Trex, given by the **TrexState enum,** which helped track the key moments of the game.

The painting of the components from the Panel was done by overriding the **paintComponent()** method from the class JComponent.

```
@Override
protected void paintComponent(Graphics graphics) {
    super.paintComponent(graphics);

    // painting the background
    graphics.setColor(new Color(239, 239, 239));
    graphics.fillRect(0, 0, getWidth(), getHeight());

    // drawing the landscape
    groundObj.drawGround(graphics);
    sky.drawClouds(graphics);
    obstacleManager.drawObstacle(graphics);

    // drawing the trex in its different states
    switch (characterState) {
        case INITIAL_STATE:
        case JUMPING:
            trex.drawJumpingTrex(graphics); // the trex is drawn the same way when jumping and in the initial stage
            break;
        case RUNNING:
            trex.drawRunningTrex(graphics);
            break;
        case DUCKING:
            trex.drawDuckingTrex(graphics);
            break;
        case DEAD:
            trex.drawDeadTrex(graphics);
            // when the trex is dead, it means that the game is over so the "Game Over" text is drawn as well
            graphics.drawImage(gameOverImg, (WINDOW_WIDTH - gameOverImg.getWidth()/2)/2,
                    (WINDOW_HEIGHT - gameOverImg.getHeight()/2)/2 - 70, gameOverImg.getWidth()/2,
                    gameOverImg.getHeight()/2, null);
            break;
    }
```

```java
                break;
        }

        // drawing the informationg about the current game: highscore, score, player name
        graphics.setColor(new Color(65, 74, 76));
        graphics.setFont(new Font("Monospaced", Font.BOLD, 20));
        graphics.drawString("HI " + String.valueOf(highScore), WINDOW_WIDTH - 150, 30);
        graphics.drawString("SCORE " + String.valueOf(score), WINDOW_WIDTH - 300, 30);
        graphics.drawString("PLAYER: " + playerName, 30, 30);
    }
```

Methods that deal with the game functionality:

```java
    private void restartGame()
    {
        savePlayerData();   // upon restarting the game, the data of the previously played game has to be saved
        restartBtn.setVisible(false);  // the restart button is made visible
        score = 0;   // the score is set back to 0
        trex.setDead(false);    // the trex is not dead anymore

        // setting the initial position of the trex and of the obstacles
        trex.setX(50);
        trex.setY(100);
        obstacleManager.resetObstacles();

        characterState = TrexState.INITIAL_STATE; // and the trex is brought back to its initial state
    }

    private void executeGame()
    {
        if (characterState == TrexState.DEAD)   // if the character is dead, restart the game
            restartGame();

        if (characterState == TrexState.INITIAL_STATE)  // after pressing the space bar for the first time to start playing
        {
            characterState = TrexState.JUMPING;     // the trex jumps and its position is shifted a bit
            trex.jump();
            trex.setX(50);
        }

        characterState = TrexState.JUMPING;
        trex.jump();
    }
```

```java
    // method that increases the score of the player
    public void increaseScoreBy(int points) { this.score += points; }

    // saving the current player's game stats to the ranking list
    public void savePlayerData()
    {
        Player newPLayer = new Player();
        newPLayer.setPlayerName(playerName);
        newPLayer.setScore(score);
        rankingManager.addPLayerToRankingList(newPLayer);
        highScore = rankingManager.getPlayerHS(newPLayer);
    }
```

The implementation of the Key Listeners:

```java
@Override
public void keyPressed(KeyEvent e)
{
    switch (e.getKeyCode()) {
        // dealing with the way the game responds when the space and arrow-up keys are pressed
        case KeyEvent.VK_SPACE:
        case KeyEvent.VK_KP_UP:
            executeGame();
            break;
        case KeyEvent.VK_KP_DOWN:   // when the arrow-down key is pressed, the trex ducks
            trex.setDown(true);
            characterState = TrexState.DUCKING;
            break;
        default:
            // if any other key is pressed the trex is killed, the game stops, and the following error message pops up
            characterState = TrexState.DEAD;
            String errorMsg = "\nSeems like you used the wrong controls :(\nKind reminder that you can only make the Trex move\nby " +
                    "using the Space/Arrow-Up keys for jumping,\nor the Arrow-Down key for ducking.";
            JOptionPane.showMessageDialog(new JFrame(), errorMsg, "Oops!", JOptionPane.ERROR_MESSAGE);
            restartBtn.setVisible(false);
            restartGame();
            break;
    }
}
```

```java
@Override
public void keyReleased(KeyEvent e)
{
    // when the space/arrow up keys are released, the trex keeps on running
    characterState = TrexState.RUNNING;

    // if the arrow-down key is released, the trex doesn't duck anymore
    if (e.getKeyCode() == KeyEvent.VK_KP_DOWN)
        trex.setDown(false);
}
```

4. Conclusions

4.1. What I learnt

- how to create a Java Swing applications
- how to work with Java Swing's component painting system
- how to use several JComponents, such as JButtons, JTextFields, JTables, JTextAreas and add their corresponding Action Listenersnn
- how to use the Runnable Interface
- how to extract data from other files of the project

4.2. Further Developments

- storing the game information in a database instead of an ArrayList
- adding a Night Mode feature, where the time of day in the game changes after a certain number of points is reached; this would add to the feeling of endlessness of the game
- adding sounds for the different actions that the Trex does (jumping, ducking) or for when the game ends
- adding other game modes/levels, such as an Equestrian or Gymnastics Mode