Software Development 2 Coursework (SET11103)

"Get out of my swamp"

Grigori Zogka

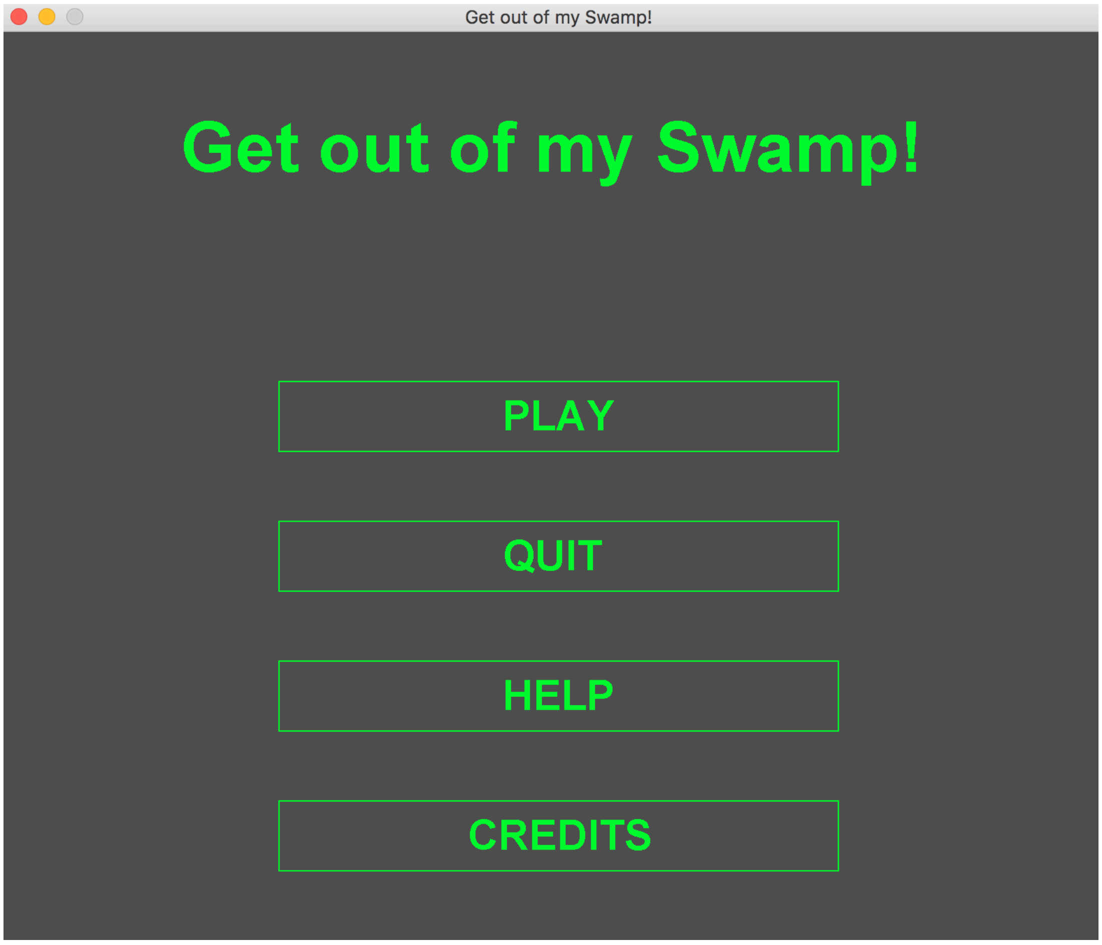
Matriculation number: 40418304

Table of Contents

Application overview

This section includes the application description and an overview o the applications functionality.

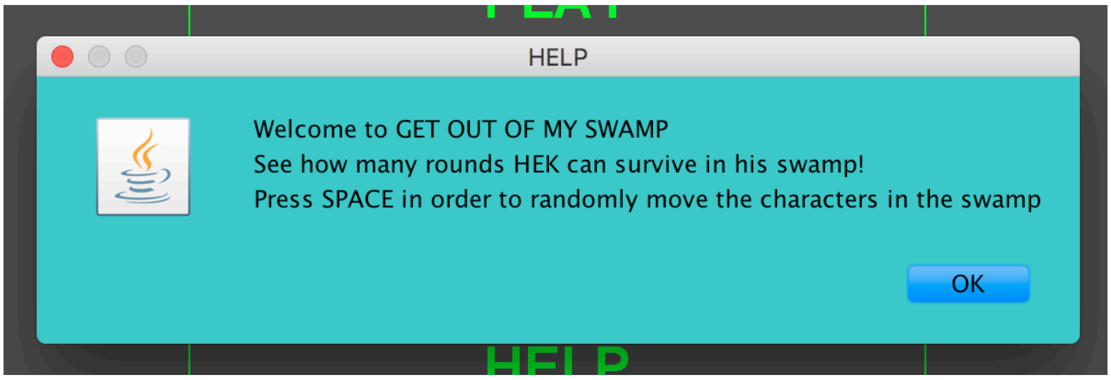
When starting the application the main menu is being called. The main menu includes four basic options for the user to choose. The user can use the mouse to click on any of the buttons.



The fist Button is the “PLAY” button. This is the button that initiates the game.

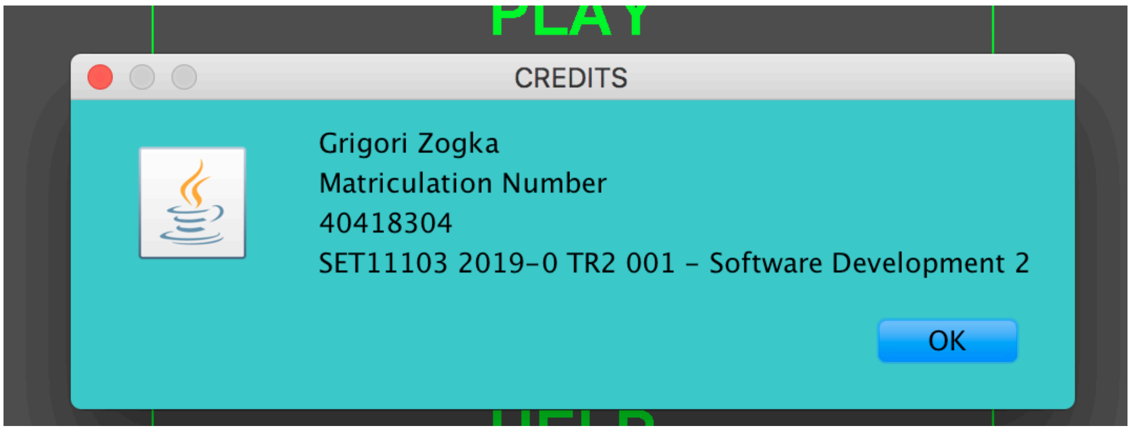
The “QUIT” button can be used by the use if he chooses to close the application.

The “HELP” button provide some basic information you the application and and the game controls. The game can be played just by pressing the SPACE key on the keyboard, that essentially moves all the objects on the grid.



Welcome to GET OUT OF MY SWAMP
See how many rounds HEK can survive in his swamp!
Press SPACE in order to randomly move the characters in the swamp

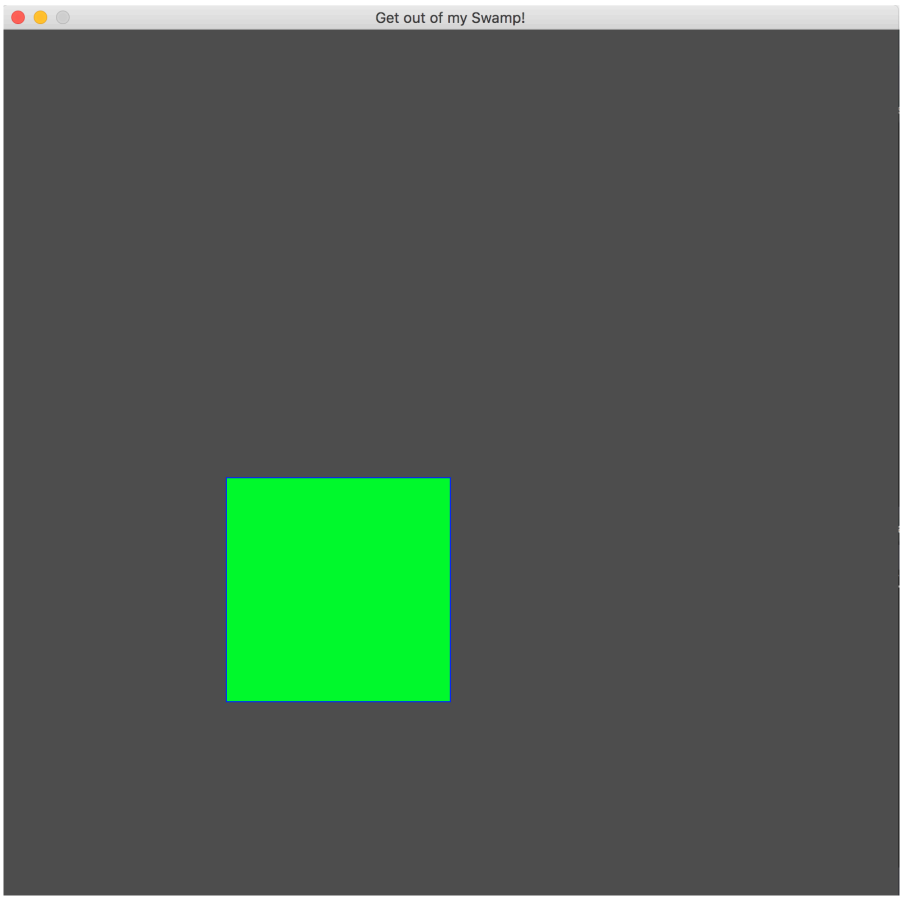
The “CREDITS” button provides some basic information about the project:



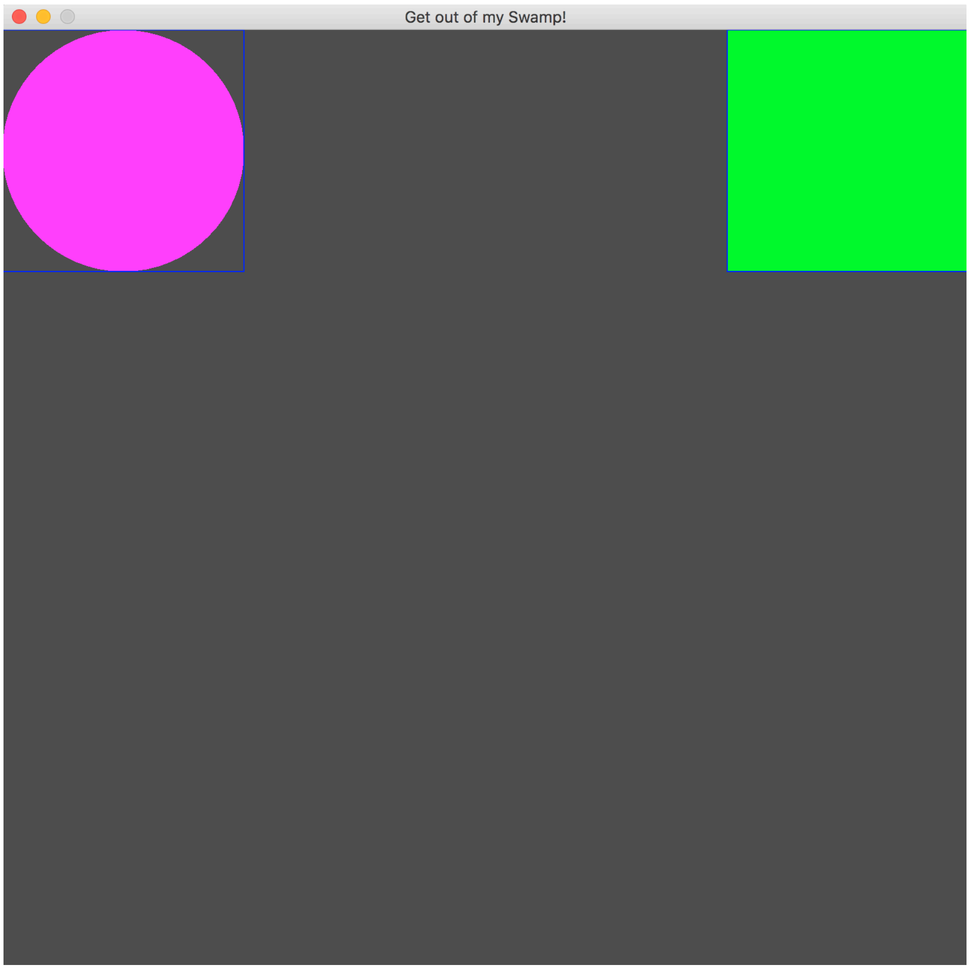
Grigori Zogka
Matriculation Number
40418304
SET11103 2019-0 TR2 001 - Software Development 2

Starting the game

When the user clicks on “PLAY” the game starts and he is able to view the game grid and the Ogre - “Hek” in a random position inside the game grid (except from 0,0). The grid has been set up to be 800x800 and each entity in the Grid I approximately 200x200, this analogy constitutes a 4x4 grid between the game entities and the game grid-screen. “Hek” is being visually represented by a green square.

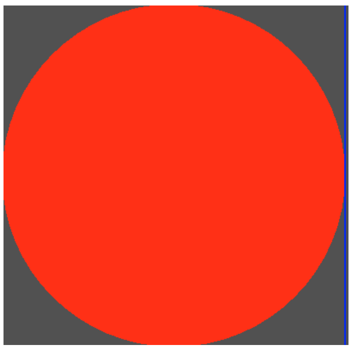


When the user presses the SPACE key on the keyboard the Ogre moves and there is a 1 to 3 possibility for an enemy to be summoned in the swamp, always on the tom left corner of the grid - position 0,0.

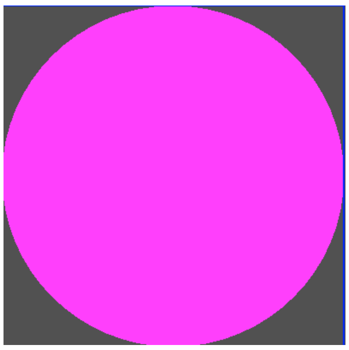


The enemies in the swamp are being represented by circles. Each circle has a distinct colour and represents a deferent type of enemy.

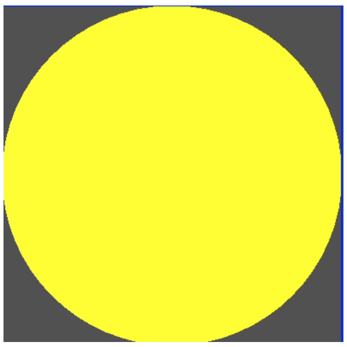
RED for Parrot



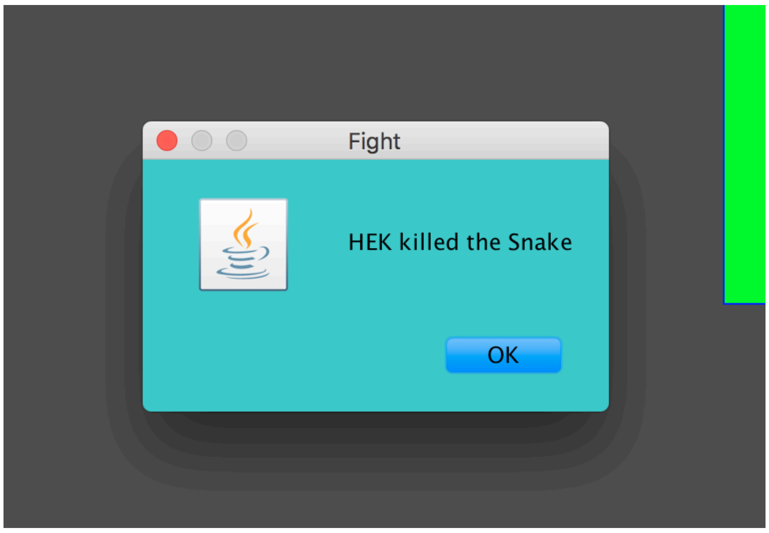
MAGENTA for Snake



RED for Donkey

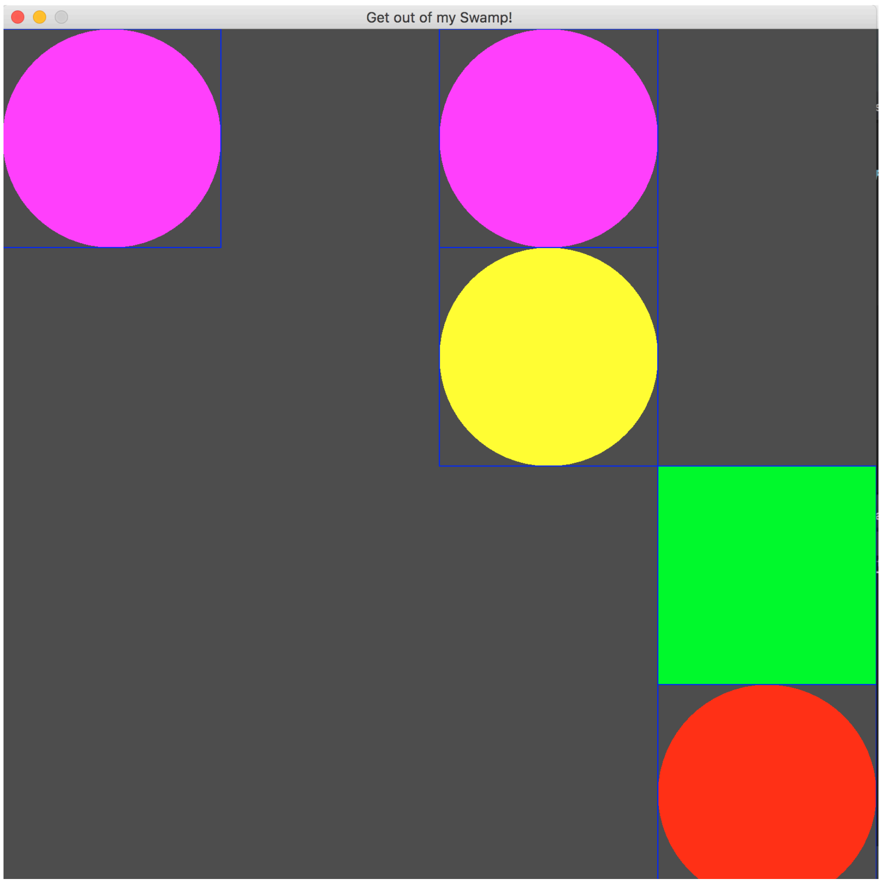


When the Ogre and any of the above enemies meet in the same coordinates they fight and the Ogre kills the enemy. After the enemy has been killed it is being removed from the swamp. The player is being notified any time an Enemy dies wit an JOptionPane message window.

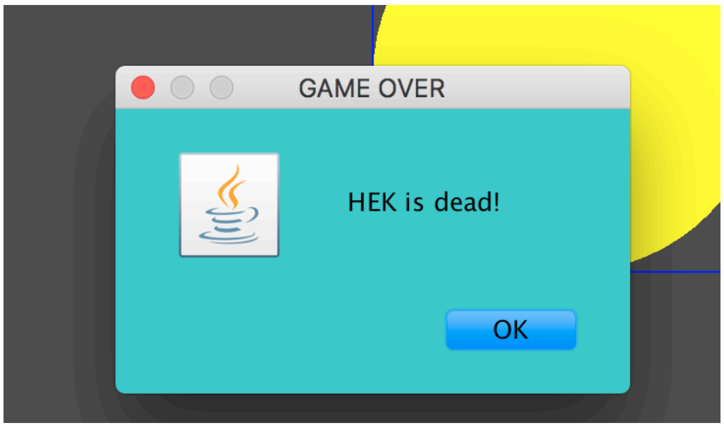

Fight

HEK killed the Snake

After the enemies enter the swamp these also move randomly in the swamp every time the player presses SPACE.



If the Ogre moves into the same coordinates with two or more enemies the the Ogre dies and the player is being notified with the below message.



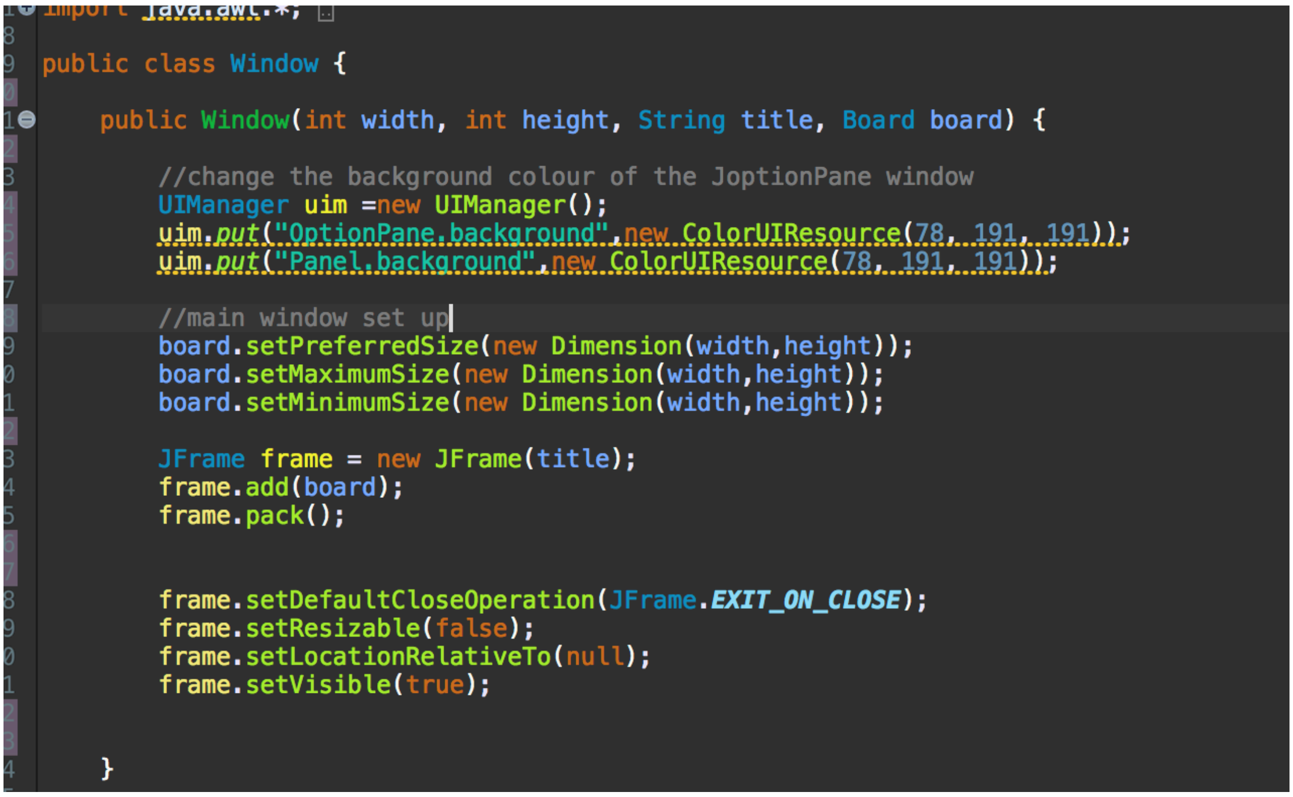
HEK is dead!

The death of the Ogre constitutes the end of the game. After the Payer clicks OK on the above shown JOptionPane window tenth game ends and the Payer is being redirected back to the main menu.

Creating the Swamp

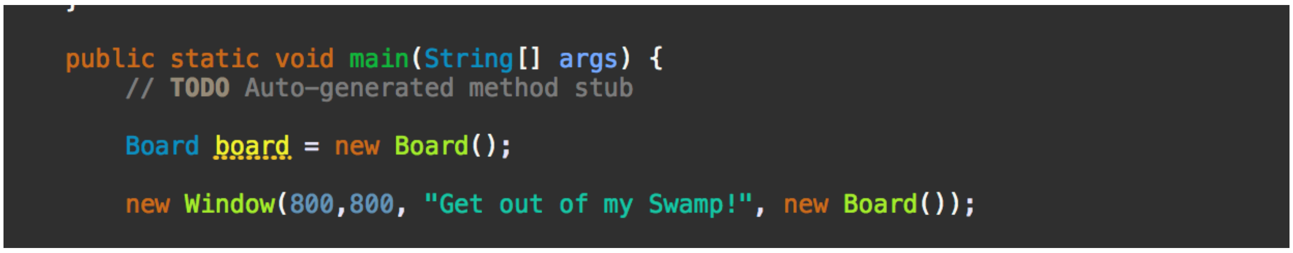
The Swamp has for the application has been created using jPanel, Frame and g Graphics.

In the class Window all the basic window properties are being defined.



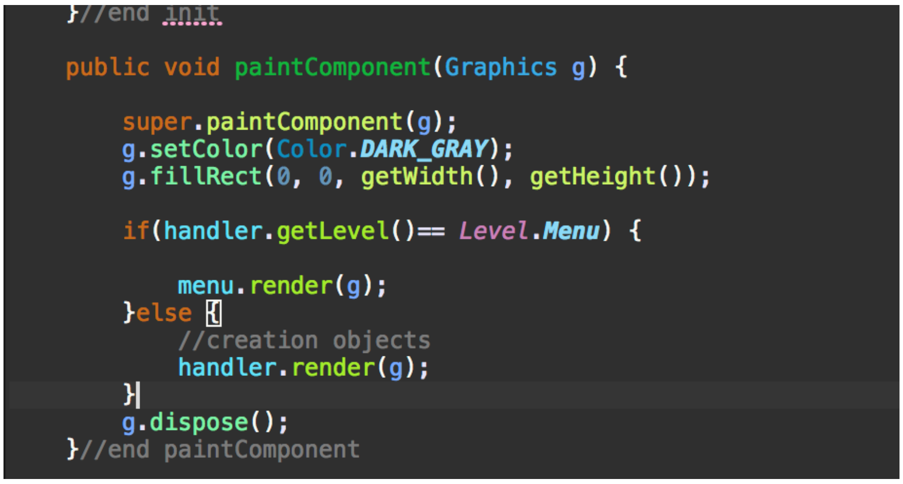
```java
import java.awt.*;

public class Window {

    public Window(int width, int height, String title, Board board) {

        //change the background colour of the JoptionPane window
        UIManager uim =new UIManager();
        uim.put("OptionPane.background",new ColorUIResource(78, 191, 191));
        uim.put("Panel.background",new ColorUIResource(78, 191, 191));

        //main window set up
        board.setPreferredSize(new Dimension(width,height));
        board.setMaximumSize(new Dimension(width,height));
        board.setMinimumSize(new Dimension(width,height));

        JFrame frame = new JFrame(title);
        frame.add(board);
        frame.pack();


        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);


    }
```

The window is being initiated on the main class and is being set up to 800x800 pixels.



```
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Board board = new Board();

        new Window(800,800, "Get out of my Swamp!", new Board());
```

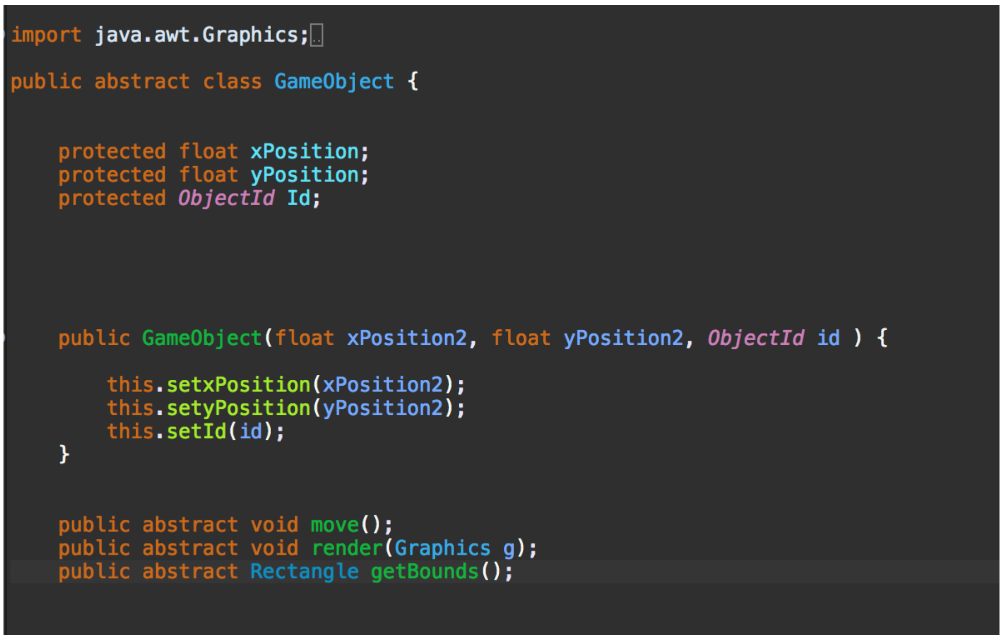
In order to actually daw the main window and set its colour to “DARK_GRAY” I used the paintComponent(Graphics g) method on class Board.



```
    }//end init

    public void paintComponent(Graphics g) {

        super.paintComponent(g);
        g.setColor(Color.DARK_GRAY);
        g.fillRect(0, 0, getWidth(), getHeight());

        if(handler.getLevel()== Level.Menu) {

            menu.render(g);
        }else {
            //creation objects
            handler.render(g);
        }
        g.dispose();
    }//end paintComponent
```

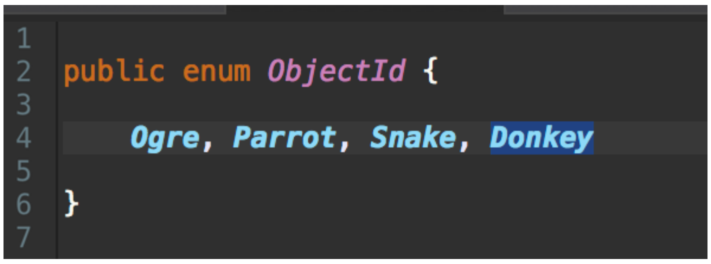
Creating the Game Entities

All the main entities of the game (Ogre, Donkey, Parrot and Snake) derive from the same abstract class - class GameObject. In this class (GameObject) all the main attributes of the Entities are being defined and these are being inherited in each Entity.



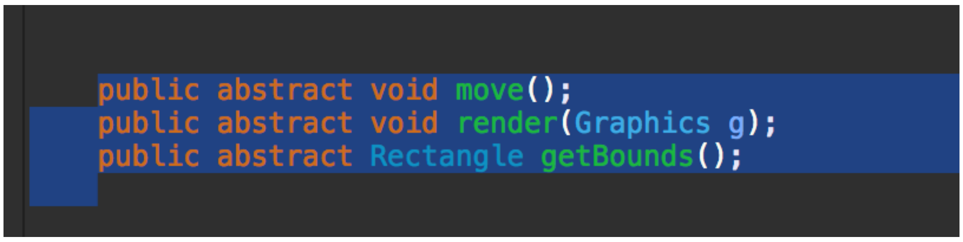
```java
import java.awt.Graphics;

public abstract class GameObject {

    protected float xPosition;
    protected float yPosition;
    protected ObjectId Id;



    public GameObject(float xPosition2, float yPosition2, ObjectId id ) {

        this.setxPosition(xPosition2);
        this.setyPosition(yPosition2);
        this.setId(id);
    }

    public abstract void move();
    public abstract void render(Graphics g);
    public abstract Rectangle getBounds();
```

All the Entities share the same basic attributes like the possible coordinates on the x and y axis of our window and call also get a specific enumerator assigned to them in order to define and] manage the time of Entity that is being generated in the game.



```java

public enum ObjectId {

    Ogre, Parrot, Snake, Donkey

}

```

I have created three abstract classes that are being defined in class GameObject. These classes are required and implemented for each of the game Entities.



```java
    public abstract void move();
    public abstract void render(Graphics g);
    public abstract Rectangle getBounds();
```

The **move( );** method is required in order to set movement restrictions in our objects so that they can only move inside the window we created.

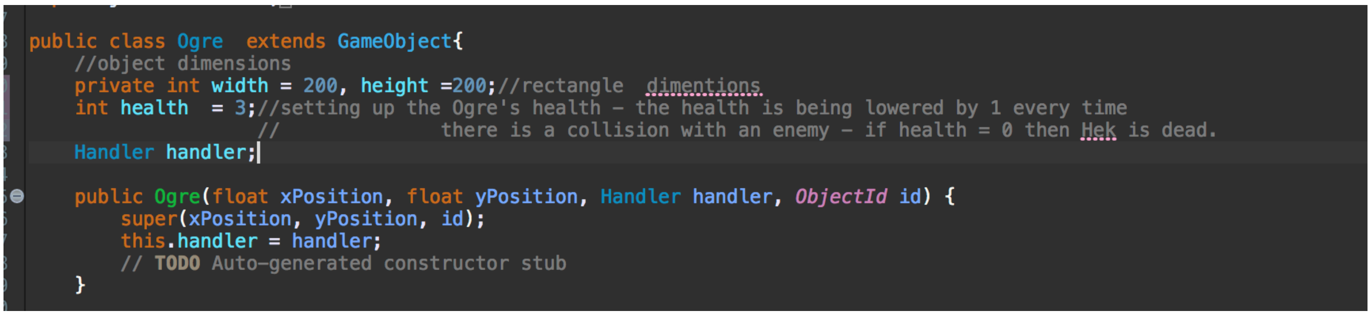
The **render(Graphics g);** method utilises the Graphics g, Java library and generates the required graphics in the window.

The **Rectangle getBounds();** method issuing used in order to set up the rectangle bounds for each game object. The bounds are required in order to check if there is any collision between the objects.

Each deferent entity (Ogre, Donkey, Parrot and Snake) in the game is a different object that essentially inherits its attributes from the abstract class GameObject.

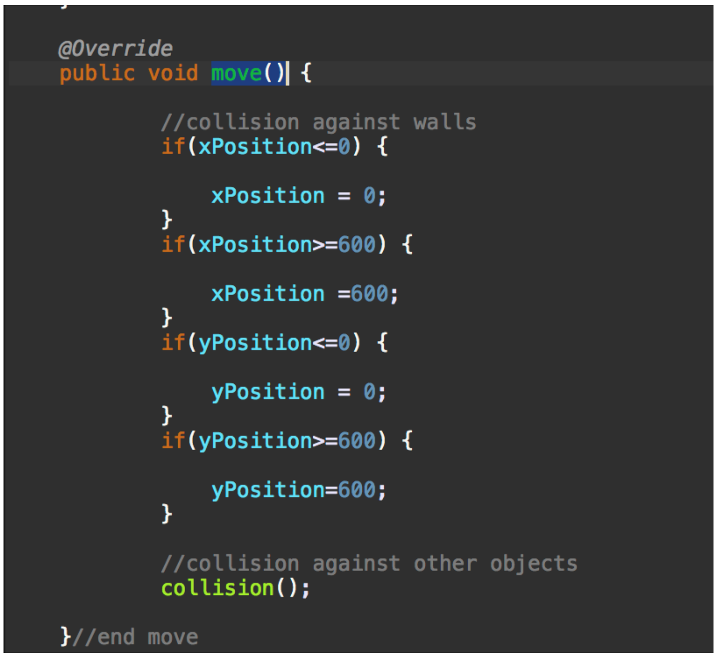
Stating with the Ogre class. Ogre class follows the same pattern and works the same way as all other Game Objects with the exception of the the tribute "Health".- Ogre's health - the health is being lowered by 1 every time there is a collision with an enemy - if health is 0 then the Ogre is dead. This is deferent for all the other game objets as for them just a collision with a single object is enough to determine the there are dead and theta there must be removed from the game.

The Constructor sett all the required parameters in order to instantiate the object when required.

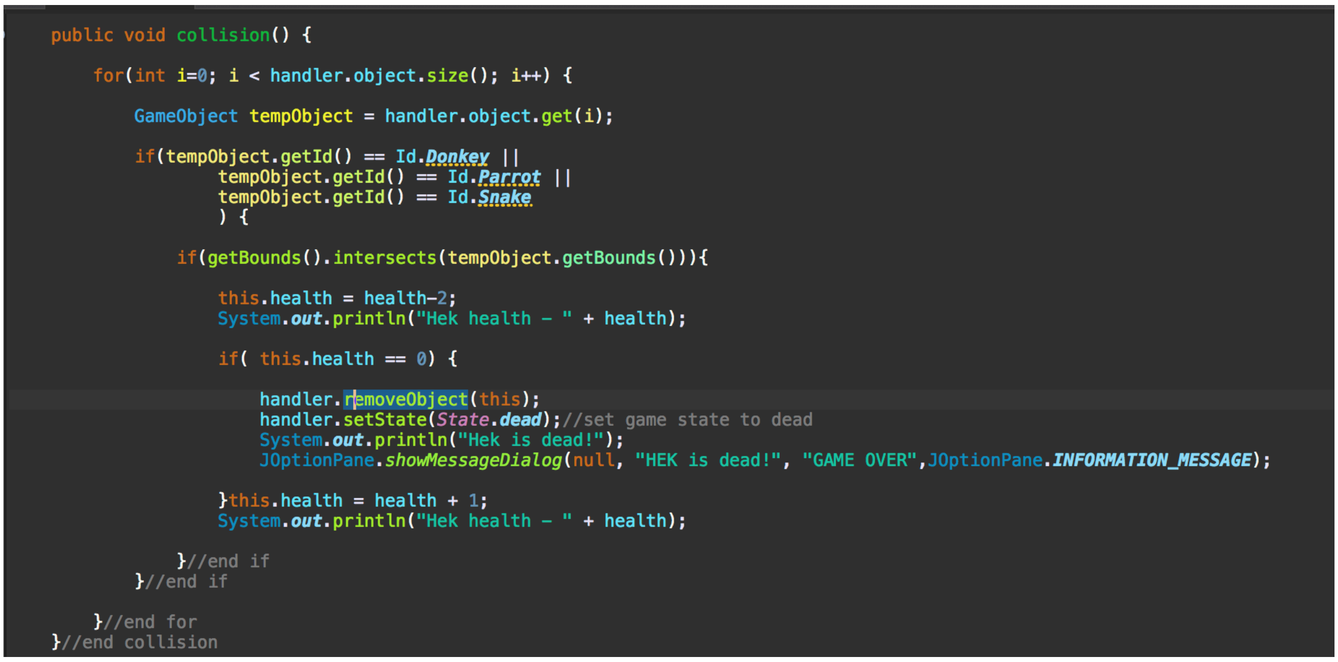


```
public class Ogre  extends GameObject{
    //object dimensions
    private int width = 200, height =200;//rectangle  dimentions
    int health  = 3;//setting up the Ogre's health - the health is being lowered by 1 every time
                //               there is a collision with an enemy - if health = 0 then Hek is dead.
    Handler handler;

    public Ogre(float xPosition, float yPosition, Handler handler, ObjectId id) {
        super(xPosition, yPosition, id);
        this.handler = handler;
        // TODO Auto-generated constructor stub
    }
```

The **move( );** method is being utilised in order to ensure that each object can not move outside the created game window. In this method method **collision( );** is also been called.

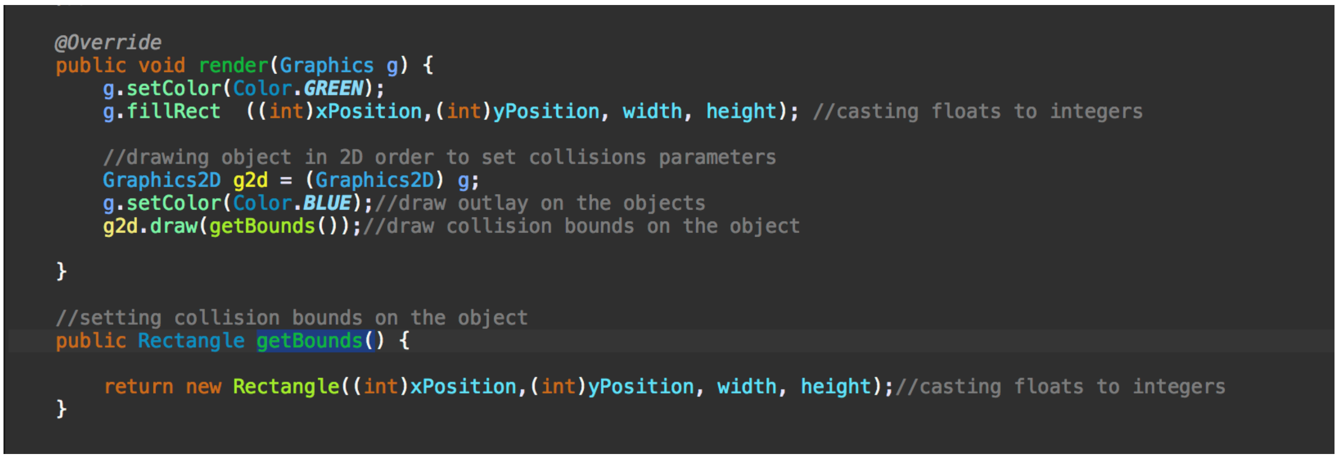


```
    @Override
    public void move() {

            //collision against walls
            if(xPosition<=0) {

                xPosition = 0;
            }
            if(xPosition>=600) {

                xPosition =600;
            }
            if(yPosition<=0) {

                yPosition = 0;
            }
            if(yPosition>=600) {

                yPosition=600;
            }

            //collision against other objects
            collision();

    }//end move
```


```java
public void collision() {

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

        GameObject tempObject = handler.object.get(i);

        if(tempObject.getId() == Id.Donkey ||
                tempObject.getId() == Id.Parrot ||
                tempObject.getId() == Id.Snake
                ) {

            if(getBounds().intersects(tempObject.getBounds())){

                this.health = health-2;
                System.out.println("Hek health - " + health);

                if( this.health == 0) {

                    handler.removeObject(this);
                    handler.setState(State.dead);//set game state to dead
                    System.out.println("Hek is dead!");
                    JOptionPane.showMessageDialog(null, "HEK is dead!", "GAME OVER",JOptionPane.INFORMATION_MESSAGE);

                }this.health = health + 1;
                System.out.println("Hek health - " + health);

            }//end if
        }//end if

    }//end for
}//end collision
```

Method **collision( );** determines what happens if there is any collision between the objects.

Method **render(Graphics g);** essentially renders the object in the game window. Here we also define each objects size and colour. The rendered rectangle is being dined through method **getBounds( );.**

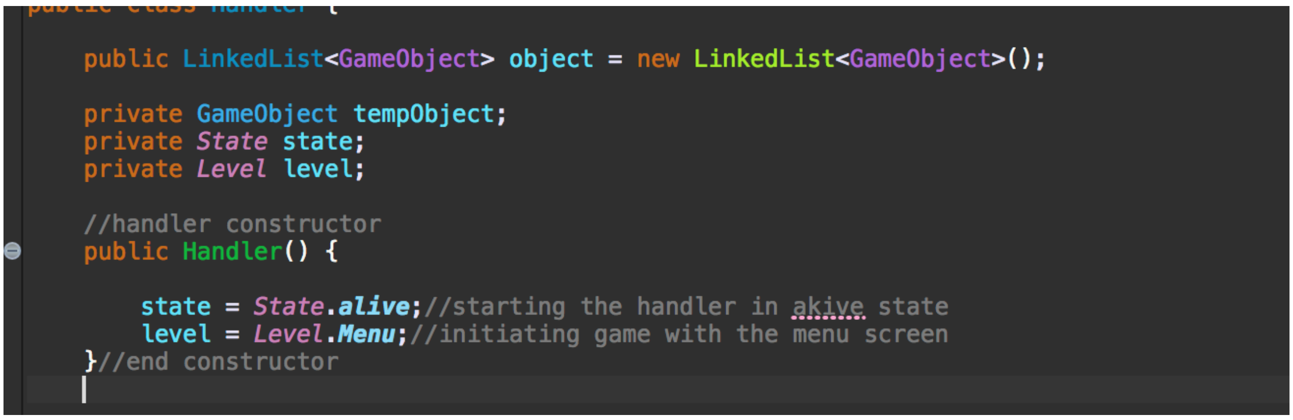

```java
@Override
public void render(Graphics g) {
    g.setColor(Color.GREEN);
    g.fillRect  ((int)xPosition,(int)yPosition, width, height); //casting floats to integers

    //drawing object in 2D order to set collisions parameters
    Graphics2D g2d = (Graphics2D) g;
    g.setColor(Color.BLUE);//draw outlay on the objects
    g2d.draw(getBounds());//draw collision bounds on the object

}

//setting collision bounds on the object
public Rectangle getBounds() {

    return new Rectangle((int)xPosition,(int)yPosition, width, height);//casting floats to integers
}
```

Game Logic

In order to manage the Entities - Objects in the Game the **Handler** class was created.

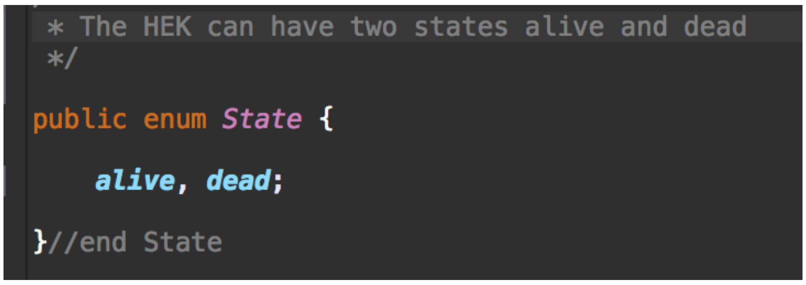
In the **Handler** Class I created a LinkedList that will include al the objects that are being added to the game.

Within the Handler constructor I set up the status of the **Handler** based on two enumerators:



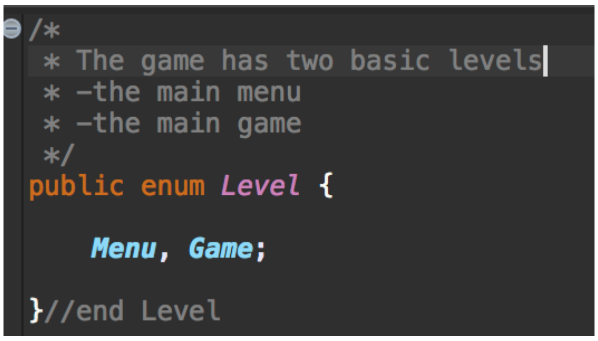
```
    public LinkedList<GameObject> object = new LinkedList<GameObject>();

    private GameObject tempObject;
    private State state;
    private Level level;

    //handler constructor
    public Handler() {

        state = State.alive;//starting the handler in akive state
        level = Level.Menu;//initiating game with the menu screen
    }//end constructor
```

- **Sate:** referring to the two statuses that the Ogre can have **alive** (when ogre is alive) and **dead** when the Ogre is dead (health = 0). Hasler constructor is being set up to alive.



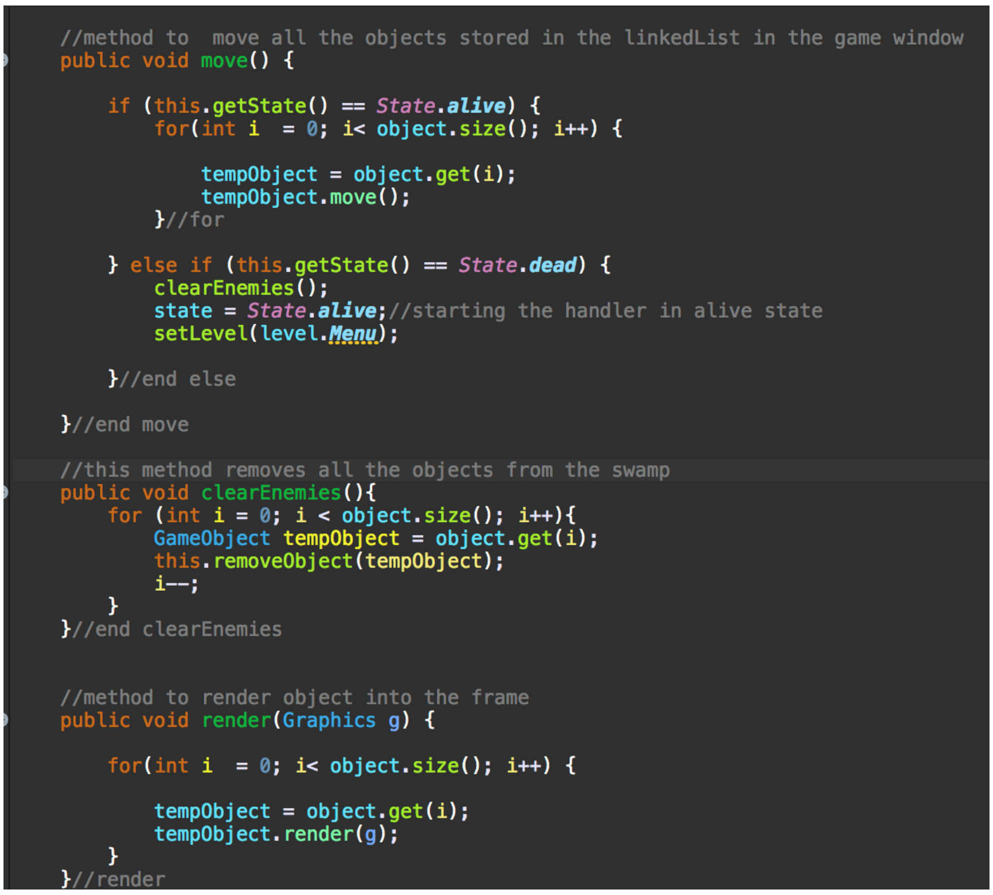
```
 * The HEK can have two states alive and dead
 */

public enum State {

    alive, dead;

}//end State
```

- **Level:** referring to the two statuses that the Game can have **Menu** (for the main menu) and **Game** (for the main game). Hasler constructor is being set up to **Menu**.



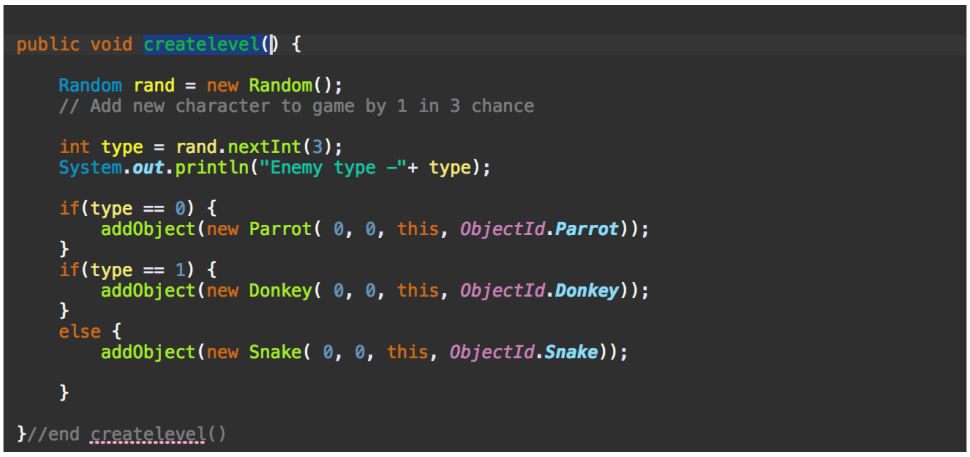
```
/*
 * The game has two basic levels
 * -the main menu
 * -the main game
 */
public enum Level {

    Menu, Game;

}//end Level
```

The **move( );** method and the **ender(Graphics g);** are being utilised in order to move and subsequently render the individual objects stored in "object" linked list in the game window, while **clearEnemies( );** method can be used in order to remove all the objects from the game window.

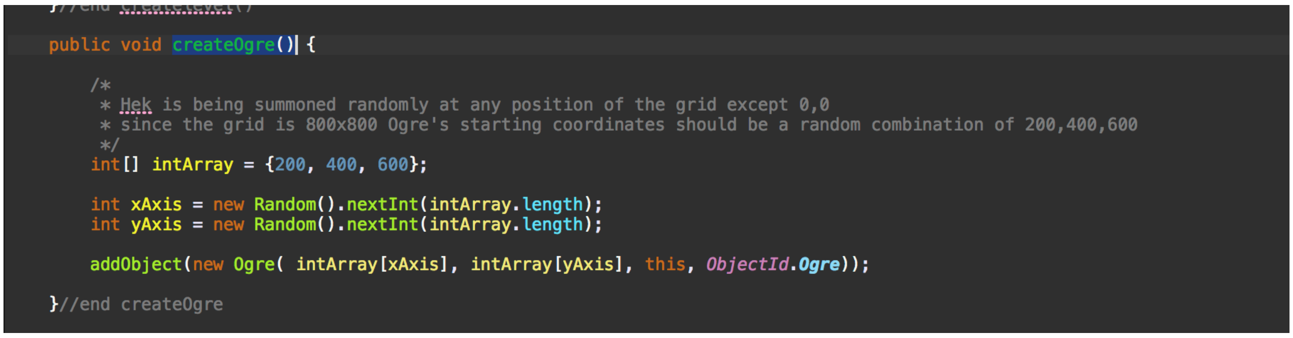


```
    //method to  move all the objects stored in the linkedList in the game window
    public void move() {

        if (this.getState() == State.alive) {
            for(int i  = 0; i< object.size(); i++) {

                tempObject = object.get(i);
                tempObject.move();
            }//for

        } else if (this.getState() == State.dead) {
            clearEnemies();
            state = State.alive;//starting the handler in alive state
            setLevel(level.Menu);

        }//end else

    }//end move

    //this method removes all the objects from the swamp
    public void clearEnemies(){
        for (int i = 0; i < object.size(); i++){
            GameObject tempObject = object.get(i);
            this.removeObject(tempObject);
            i--;
        }
    }//end clearEnemies


    //method to render object into the frame
    public void render(Graphics g) {

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

            tempObject = object.get(i);
            tempObject.render(g);
        }
    }//render
```

The **createlevel**( ); method is being used in order to randomly generate any of the three available enemies in the swamp. A random key generator that generates one random number between 0 and 2 defines what enemy will be summoned in the swamp. Enemies are always being summoned in the 0,0 coordinates - top left part of the game window.



```
public void createlevel() {

    Random rand = new Random();
    // Add new character to game by 1 in 3 chance

    int type = rand.nextInt(3);
    System.out.println("Enemy type -"+ type);

    if(type == 0) {
        addObject(new Parrot( 0, 0, this, ObjectId.Parrot));
    }
    if(type == 1) {
        addObject(new Donkey( 0, 0, this, ObjectId.Donkey));
    }
    else {
        addObject(new Snake( 0, 0, this, ObjectId.Snake));

    }

}//end createlevel()
```

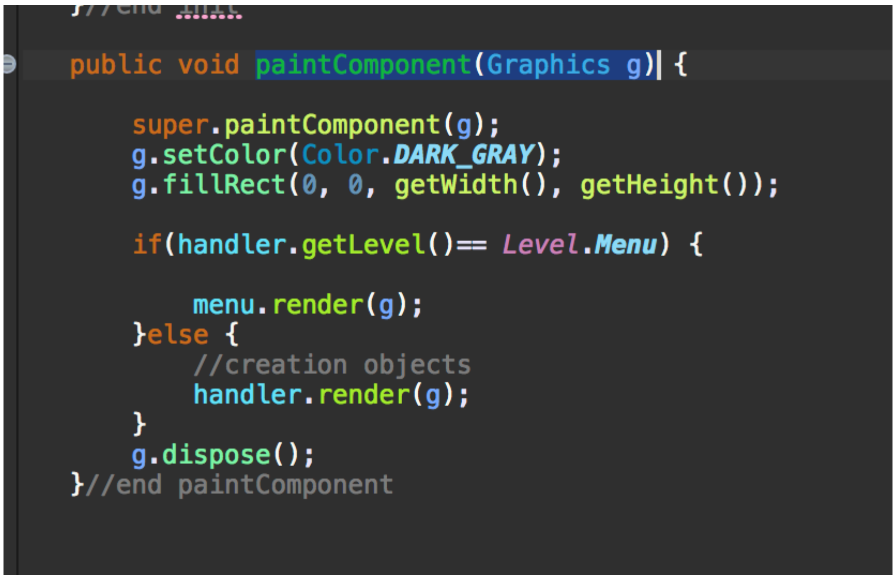


```java
}//end createLevel()

public void createOgre() {

    /*
     * Hek is being summoned randomly at any position of the grid except 0,0
     * since the grid is 800x800 Ogre's starting coordinates should be a random combination of 200,400,600
     */
    int[] intArray = {200, 400, 600};

    int xAxis = new Random().nextInt(intArray.length);
    int yAxis = new Random().nextInt(intArray.length);

    addObject(new Ogre( intArray[xAxis], intArray[yAxis], this, ObjectId.Ogre));

}//end createOgre
```

The **createOgre();** method ensures that the main character - the Ogre Her is being summoned in random positions in the game window.

Class "**Board**" mainly sets up the game logic and the game stages.

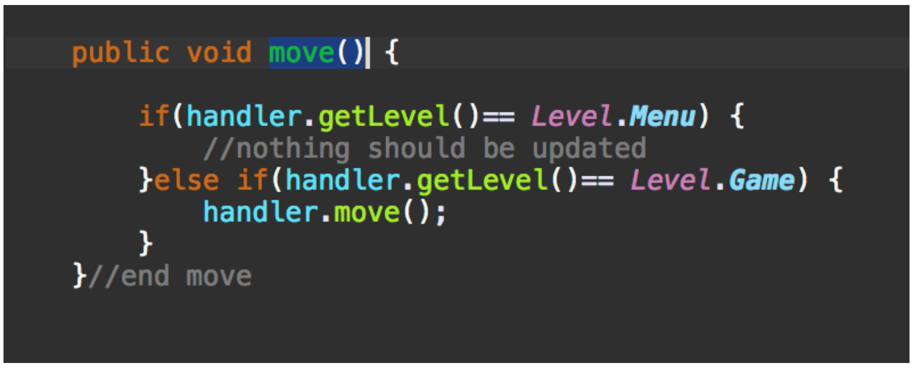
The **init ();** method initiates all the functions that the that game has (Menu, Handler, addKeyListener, MouseListener) and **paintComponent(Graphics g)** renders the window's properties while setting the parameters for rendering when the game is stein and in Menu and when the game is seated in the Game level.



```java
}//end init

public void paintComponent(Graphics g) {

    super.paintComponent(g);
    g.setColor(Color.DARK_GRAY);
    g.fillRect(0, 0, getWidth(), getHeight());

    if(handler.getLevel()== Level.Menu) {

        menu.render(g);
    }else {
        //creation objects
        handler.render(g);
    }
    g.dispose();
}//end paintComponent
```

Method **run ( );** initiates the game loop, while method **start( );** starts the game thread.

With **method( );** move the move parameter from Handle class is being passed to the Board class while at the same time ensuring that that this is any active when the level is being set to Game (not Menu).

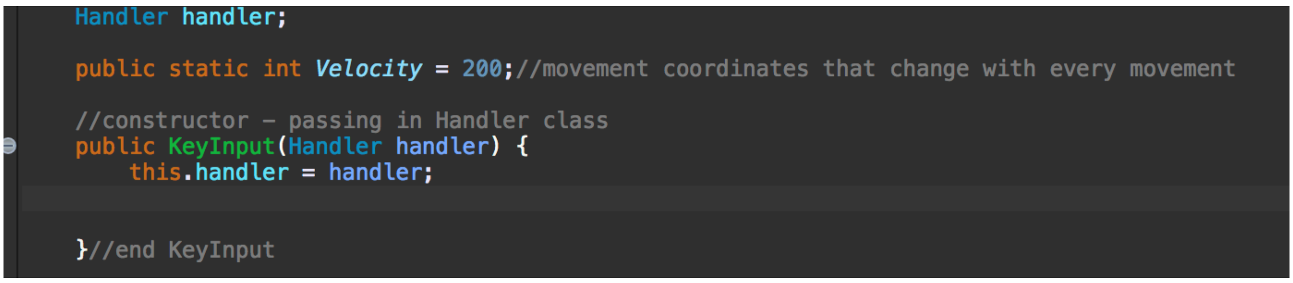


```java
public void move() {

    if(handler.getLevel()== Level.Menu) {
        //nothing should be updated
    }else if(handler.getLevel()== Level.Game) {
        handler.move();
    }
}//end move
```

Movement

In order to move the game objects in the game window KeyInput class is being utilised. This class extends KeyAdapter and provides all the movement functionality required in order to play the game. Since the movement for al the character is randomly generated Ann the action for the game are being initiated with the use of only one key through the keyboard - SPACE.

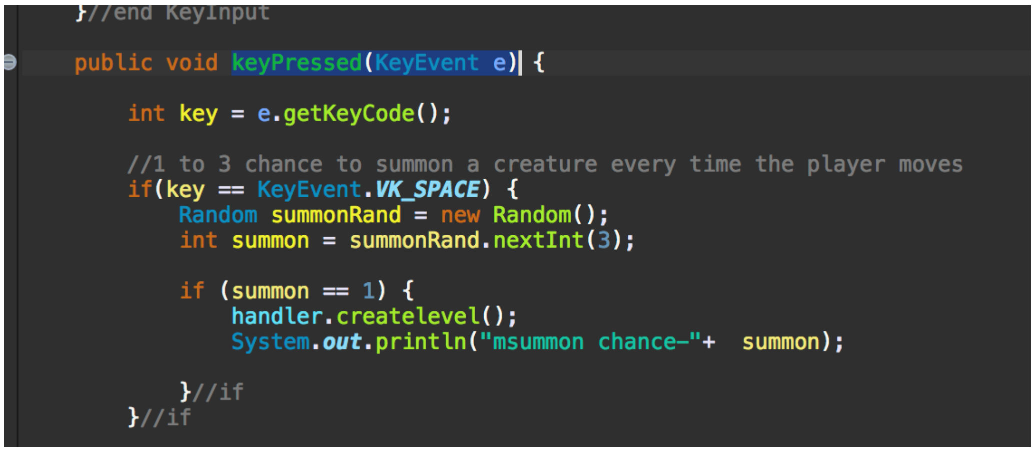
In the constructor of the class the class “Handler" is being instantiated in order to manage the movement of the various game objects bu using its methods.



```
    Handler handler;

    public static int Velocity = 200;//movement coordinates that change with every movement

    //constructor - passing in Handler class
    public KeyInput(Handler handler) {
        this.handler = handler;


    }//end KeyInput
```

BY the use of method **keyPressed(KeyEvent e);** space is being set up as the initiator of two basic functions.

The first one is to determine if an Enemy will be summoned the swamp. This is managed by a random key generator that determines with one to three possibilities if an enemy will enter the swamp.



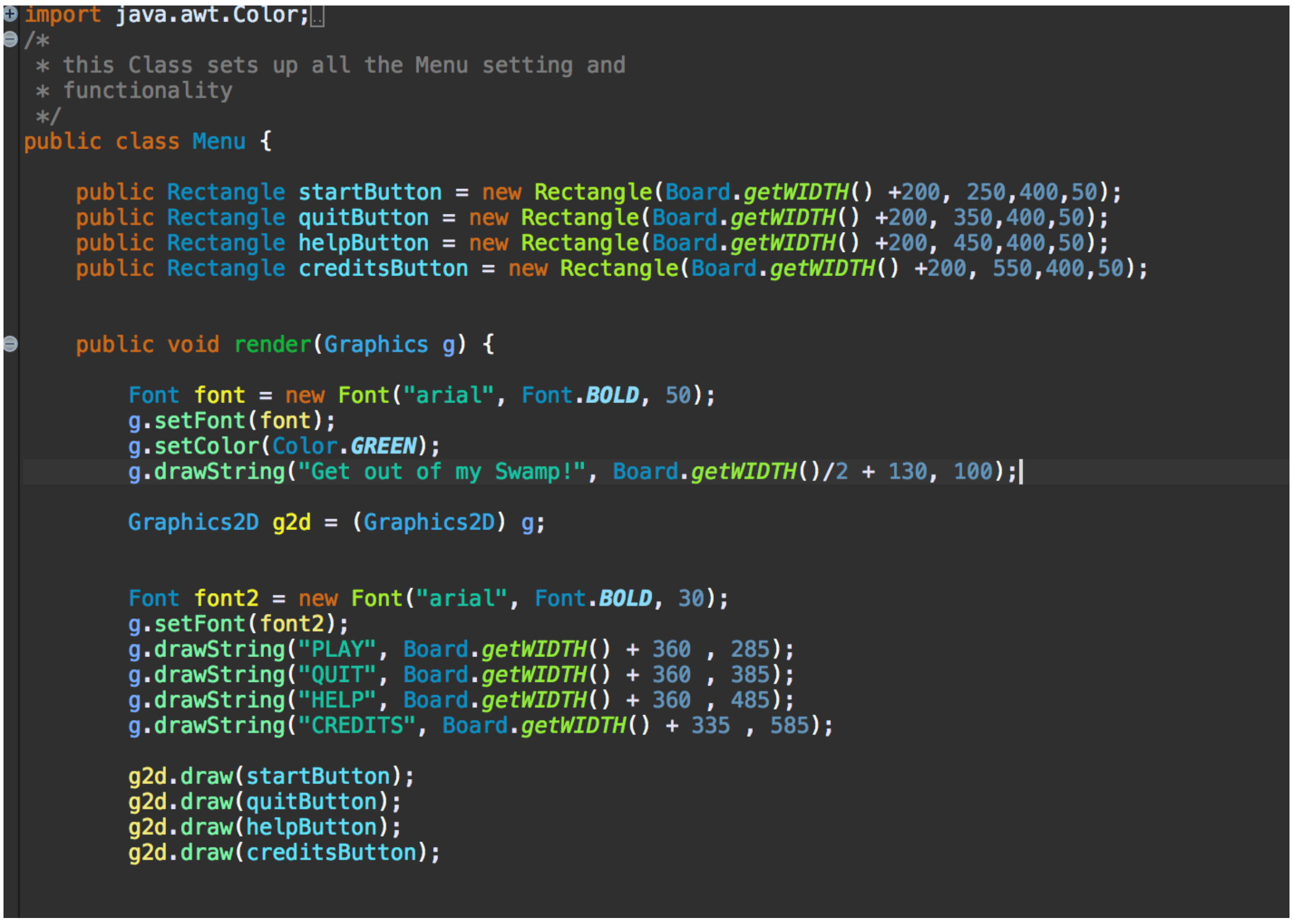
```
    }//end KeyInput

    public void keyPressed(KeyEvent e)| {

        int key = e.getKeyCode();

        //1 to 3 chance to summon a creature every time the player moves
        if(key == KeyEvent.VK_SPACE) {
            Random summonRand = new Random();
            int summon = summonRand.nextInt(3);

            if (summon == 1) {
                handler.createlevel();
                System.out.println("msummon chance-"+  summon);

            }//if
        }//if
```

The second function the press of key SPACE is performed is random move for all the objects (all enemies and the Ogre) in the swamp. The move is being decided based on the a random generated number between 0 and 8 for each game object.

Menu

The main menu for the game is being set up in class Menu. In this class all the properties, buttons and dimensions of the Menu is being determined. Class menu is the instantiated and initiated in the Board Class.

The Manu buttons can be selected with the mouse. All the button functionality of the class is managed in the **MouseInput** class. In this class I have set up the the coordinated that the Player need to click for each button and the functionality of the buttons.



```
import java.awt.Color;
/*
 * this Class sets up all the Menu setting and
 * functionality
 */
public class Menu {

    public Rectangle startButton = new Rectangle(Board.getWIDTH() +200, 250,400,50);
    public Rectangle quitButton = new Rectangle(Board.getWIDTH() +200, 350,400,50);
    public Rectangle helpButton = new Rectangle(Board.getWIDTH() +200, 450,400,50);
    public Rectangle creditsButton = new Rectangle(Board.getWIDTH() +200, 550,400,50);


    public void render(Graphics g) {

        Font font = new Font("arial", Font.BOLD, 50);
        g.setFont(font);
        g.setColor(Color.GREEN);
        g.drawString("Get out of my Swamp!", Board.getWIDTH()/2 + 130, 100);

        Graphics2D g2d = (Graphics2D) g;


        Font font2 = new Font("arial", Font.BOLD, 30);
        g.setFont(font2);
        g.drawString("PLAY", Board.getWIDTH() + 360 , 285);
        g.drawString("QUIT", Board.getWIDTH() + 360 , 385);
        g.drawString("HELP", Board.getWIDTH() + 360 , 485);
        g.drawString("CREDITS", Board.getWIDTH() + 335 , 585);

        g2d.draw(startButton);
        g2d.draw(quitButton);
        g2d.draw(helpButton);
        g2d.draw(creditsButton);
```

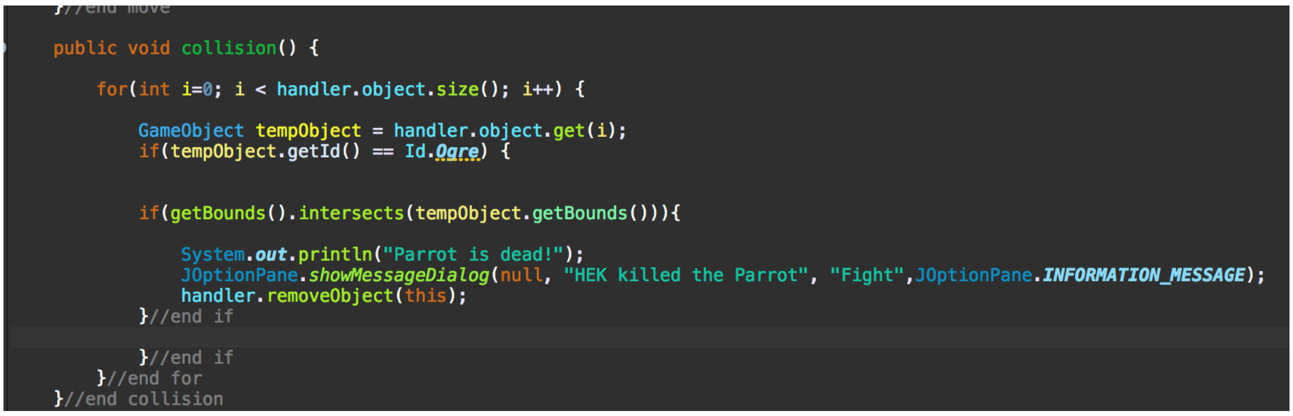
Techniques not taught on the module

There are few techniques used in the project that have not been taught in the module.

Collision detection

Collision detection has been used in order to determine if a game object Ogre, Donkey, Snake, Parrot) has the same coordinates with another object. For each game object specific bounds have been set up in order to check if there is collision.

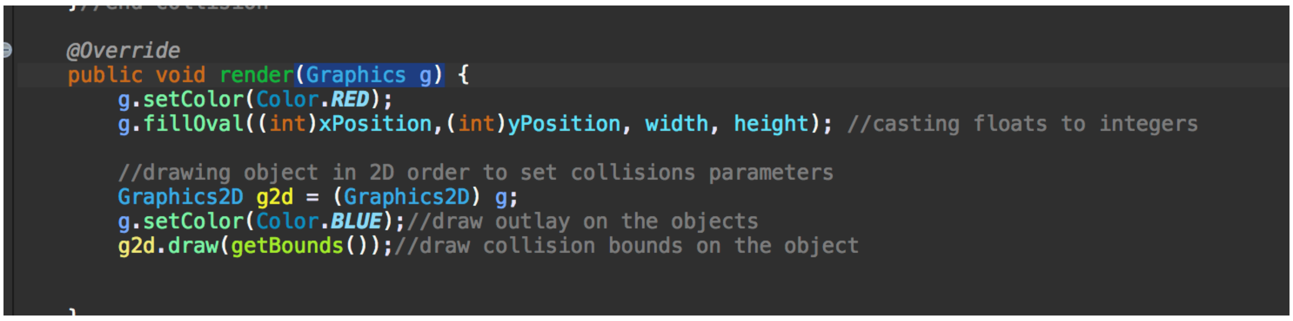
On the be bellow example collision is used in order to determine if the Parrot is killed when colliding with the Ogre.



```
}//end move

public void collision() {

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

        GameObject tempObject = handler.object.get(i);
        if(tempObject.getId() == Id.Ogre) {


        if(getBounds().intersects(tempObject.getBounds())){

            System.out.println("Parrot is dead!");
            JOptionPane.showMessageDialog(null, "HEK killed the Parrot", "Fight",JOptionPane.INFORMATION_MESSAGE);
            handler.removeObject(this);
        }//end if

        }//end if
    }//end for
}//end collision
```

Graphics g

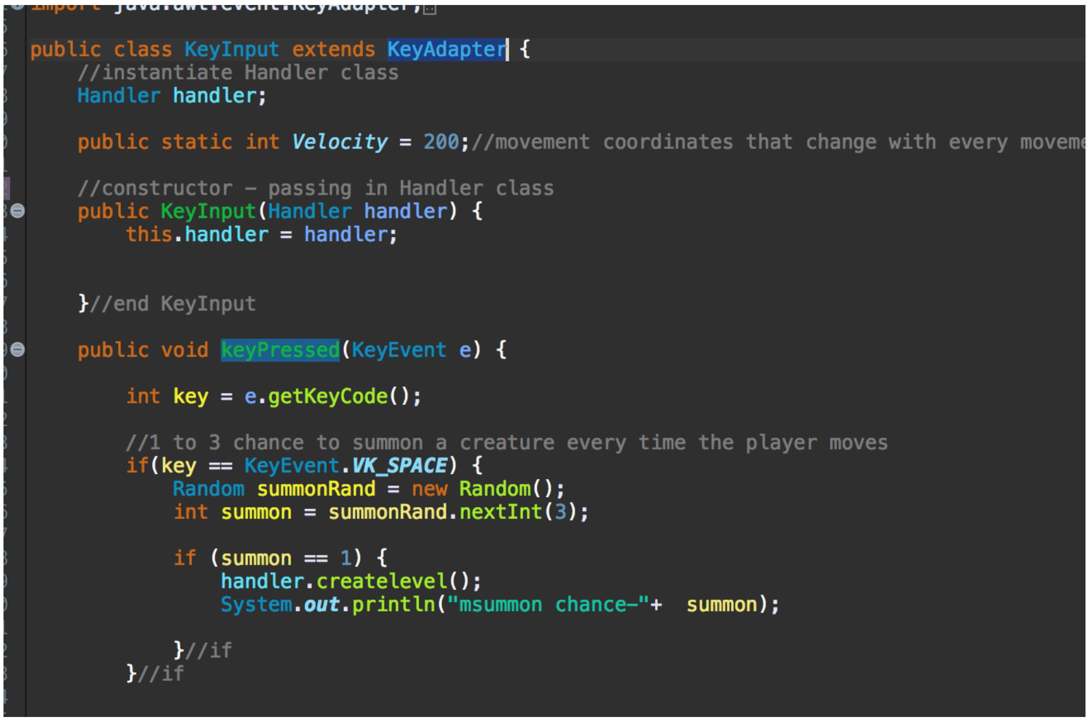
Graphics and graphics2d libraries have been used in order to create the individual game object shapes and display them on theme window.



```
}//end collision

@Override
public void render(Graphics g) {
    g.setColor(Color.RED);
    g.fillOval((int)xPosition,(int)yPosition, width, height); //casting floats to integers

    //drawing object in 2D order to set collisions parameters
    Graphics2D g2d = (Graphics2D) g;
    g.setColor(Color.BLUE);//draw outlay on the objects
    g2d.draw(getBounds());//draw collision bounds on the object


}
```

KeyAdapter

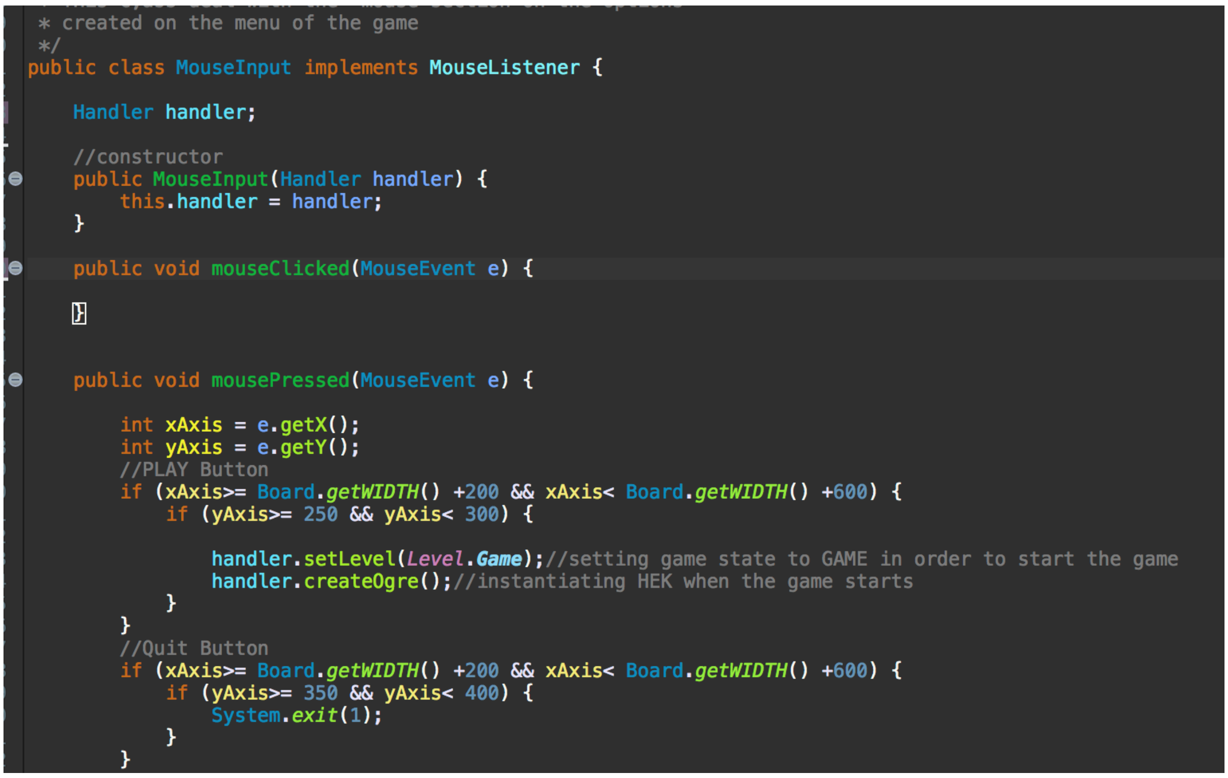
In order to manage the game objects movements in the game window KeyAdapter functionality has been utilised. This functionality allowed me set specific function when a key was pressed.



```
import java.awt.event.KeyAdapter;

public class KeyInput extends KeyAdapter {
    //instantiate Handler class
    Handler handler;

    public static int Velocity = 200;//movement coordinates that change with every moveme

    //constructor - passing in Handler class
    public KeyInput(Handler handler) {
        this.handler = handler;


    }//end KeyInput

    public void keyPressed(KeyEvent e) {

        int key = e.getKeyCode();

        //1 to 3 chance to summon a creature every time the player moves
        if(key == KeyEvent.VK_SPACE) {
            Random summonRand = new Random();
            int summon = summonRand.nextInt(3);

            if (summon == 1) {
                handler.createlevel();
                System.out.println("msummon chance-"+  summon);

            }//if
        }//if
```

MouseListener

MouseListener was utilised in order to set up the functionality of the main Manu by using the mouse.



```java
 * created on the menu of the game
 */
public class MouseInput implements MouseListener {

    Handler handler;

    //constructor
    public MouseInput(Handler handler) {
        this.handler = handler;
    }

    public void mouseClicked(MouseEvent e) {

    }


    public void mousePressed(MouseEvent e) {

        int xAxis = e.getX();
        int yAxis = e.getY();
        //PLAY Button
        if (xAxis>= Board.getWIDTH() +200 && xAxis< Board.getWIDTH() +600) {
            if (yAxis>= 250 && yAxis< 300) {

                handler.setLevel(Level.Game);//setting game state to GAME in order to start the game
                handler.createOgre();//instantiating HEK when the game starts
            }
        }
        //Quit Button
        if (xAxis>= Board.getWIDTH() +200 && xAxis< Board.getWIDTH() +600) {
            if (yAxis>= 350 && yAxis< 400) {
                System.exit(1);
            }
        }
```

Completeness, extensibility and maintainability

This Java application manages to to achieve all the required functionality based on the requirements of the coursework.

The User Interface created is clear and intuitive. With the use of the mouse the Player - User can navigate the main Manu while the came can be controlled with the use of a single key - SPACE. The Game window constitutes a 4x4 Grid.

When it comes to the actual game all the entities - enemies enter the swamp from the top left corner while the Ogre- Hek is being randomly summoned in the swamp (all positions except from the top left corner). The interactions between the Game objects is has been set ups as requested with Hek killing an enemy he shares the same coordinates with him and Hek being killed if he meets more that two enemies. The game finishes when Hek is dead.

When it comes to extensibility, the game can be easily extended with new enemies. The use of polymorphic programming when creating the Game Objects makes each game object easily identified while at the same time the use of the abstract class GameObject in order to set up the individual entries makes shared attributes of the game objects easy to amend.

The clear distinction between the functionality of each class of the application makes the application easily maintainable. All the basic attributes and structure of the game objects are being defined in the GameObject class, while their individual attributes are being defines on the game objects individual classes (Ogre, Donkey, Snake, Parrot). Handler class deals with the objects actions in the game and Board class initiates the game and sets up the individual aspects of the game in the window. All the movement in the game is being set up in the KeyInput class. Class Menu sets up all the menu attributes and MouseInput class hets up the functionality of the menu.