deod

University *of* New Orleans - Department *of* Computer Science

Target Clicker

A High Score Challenge Arcade Game

Rendered in Graphical User Interface (GUI): StdDraw API

Grading Rubric:

	Introduction:	*StdDraw API, Software Design: classes & methods*	10 minutes	
Goal 1	**(Game Class)**	*Designing the Game class*	10 minutes	+5 points (05)
Goal 2	**(Scene Class)**	*Adding a Scene to the Game*	10 minutes	+5 points (10)
Goal 3	**(Enemy Class)**	*Adding an Enemy to the Game*	10 minutes	+5 points (15)
Goal 4		*Game Loop & Game update methods*	10 minutes	+5 points (20)
Goal 5		*Measuring time in Java*	10 minutes	+5 points (25)
Goal 6	**(Player Class)**	*Add Player to the Game*	10 minutes	+5 points (30)
Goal 7		*Add mouse controls to Player*	10 minutes	+5 points (35)
Goal 8		*Add collision detection*	10 minutes	+5 points (40)
Goal 9		*Add HUD with Score to Game*	10 minutes	+5 points (45)
Goal 10		*Add click controls to Game*	10 minutes	+5 points (50)
	Homework:	*Refactor code for Zombie Game into different: classes, methods, and StdDraw Graphics.*	0 minutes	+50 points (100)

Overview:

Learning Objective

Use methods from external classes via an Application Programming Interface (API) and create your own static methods, with an emphasis of defining related methods/data within their own class files.
Design Concepts*: DRY principles, Encapsulation, Extensibility*

Application Objective

Build a graphical high score challenge arcade game. The game's objective is to click on targets as quickly as possible.

Game Architecture (Basic):

```
Start Game
Game Loop:
       Update Game
       Render Game
```

⇐ Step 1: Setup all the data necessary for playing the game
⇐ Step 2: Game Loops, which continues until the game is over

⇐ Each loop, the game updates player and enemy actions
⇐ Each loop, the game renders the new state to the display

Software Design using Methods & Classes

Methods contain a block of code that solves a small, specific task in the app. All algorithms in Java must be implemented within methods. Classes contain a collection of related methods and data. For example, the Math class contains variables and methods that provide more complex math operations. When designing software, break your algorithms into a collection of methods and group related methods/data into their own classes.

Software Design using Extensible Programming languages & APIs

Java is an extensible language which means that its functionality can grow beyond its initial release. Developers expand the language by defining new classes. Those classes may be used by other developers. Information about the class' methods is provided through an API. For example, see the Math class with its expanded set of Math operations. This homework uses the StdDraw class which contains methods/data used to draw graphics & listen for mouse/keyboard events.

External classes & methods this lab will depend on:

(class) ***Math*** → ***random()*** *(method)*
(class) ***StdDraw*** → ***draw()*** *(method)*
(class) ***System*** → ***getMiliiSeconds()*** *(method)*

StdDraw:

StdDraw is a Java class file that contains output operations that support drawing graphics.
See *Appendix A* for API or http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html

Starting this Lab *(Gitlab)*	*See code and assets on moodle.*

Iteration 1	[Create] Class: Game *(Contains all methods/data that manages the game rules)*

Plan:

Goal 1:
Create & comment an empty Game class that contains the game's logic/rules, and identify & stub the necessary methods of this class.

Game
main render update start

The Game class will contain all data/methods for managing the game's rules.

Game class should contain the **main** algorithm that runs the game, where the following actions occur: Initially **start** the game then run the game loop,

The game loop typically performs two tasks:
update game state then **render** game to display.

Implement:

Game.java

```
public class Game {
    public static void main(String[] args) {
        //Start game
        //Game loop:
            //1. update game
            //2. render game
    }

    public static void render() {
        //draw scene
        //draw enemy
        //draw player
    }

    public static void update() {
        //check for input
        //update player
        //update enemy
    }

    public static void start() {
        //setup all game data
    }
}
```

* *New code for this section is highlighted yellow*

Test:

Ensure the code compiles. There is no behavior to run and test at this stage.

Iteration	[Create] Class: Scene *(Contains all methods/data that manages the game scene or level)*

2

Plan:

Goal 2:
Draw Scene to the screen *(using the StdDraw class)*

Scene
image width, height
draw start

The Scene class will contain all data/methods that model the game scene.

In this game, the scene will appear as a static background image. Therefore, the Scene class requires three pieces of data:

1. name (with filepath) of the image file *(datatype: text)*
2. width in pixels of the image file *(datatype: int)*
3. height in pixels of the image file *(datatype: int)*

At this stage, the Scene class only needs two behaviors:
start: initialize all of the data of the scene by settings its data variables
draw: draw the scene's image to the canvas

After adding the Scene class, update the Game class' start() method to invoke Scene's start() method to initially setup a scene, and update Game class' render() method to invoke Scene's draw() method.

Implement:

Scene.java

```
public class Scene {

    //Scene data --> accessible only within this class since labeled private.
    private static String image;
    private static int width = 500;
    private static int height = 375;

    //Draws scene
    public static void draw() {
        StdDraw.picture(width/2, height/2, image);
    }

    public static void start() {
        //Setup canvas data (size & scale)
        StdDraw.setCanvasSize(width, height);     //set Canvas size for image size
        StdDraw.setXscale(0.0, width);            //set X=0 from right to left
        StdDraw.setYscale(height, 0.0);           //set Y=0 from top to bottom
        image = "assets/background.png";          //set scene image path
    }
}
```

Game.java → main()

```
public static void main(String[] args) {
    start();     //setup game
    update();    //update game
    render();    //render game
}
```

Game.java → render()

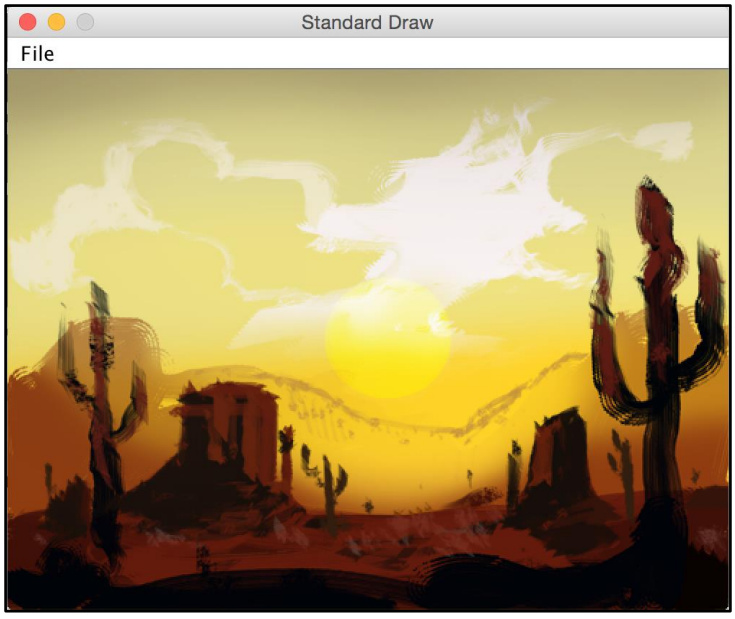
```
public static void render() {
    Scene.draw();          //draw scene
    StdDraw.show(100);     //show graphics
}
```

Game.java → start()

```
public static void start() {
    Scene.start();
}
```

Test:

Compile and execute the code. Your game program should render a static background image.



Iteration	**[Create] Class: Enemy**

3	*(Contains all methods/data that manages the enemy in game)*

Plan:

Goal 3:
Draw an Enemy to the screen *(using the StdDraw class)*

Enemy image width, height x, y draw start	The Enemy class will contain all data/methods that model the enemy. To draw an enemy to the scene, we need 5 pieces of data: name (with file path), the width, the height, and the location in the canvas. At this stage, the Enemy class only need two behaviors to achieve the goal: **start:** initialize all of the data of the enemy by settings its data variables **draw:** draw the enemy's image to the canvas
Scene image width, height draw start getWidth, getHeight	In order to spawn the Enemy within the confines of the Scene, we must able to ask the Scene for its width size and its height size. So we must add two methods in Scene: **getWidth:** returns the Scene's width size **getHeight:** returns the Scene's height size

After adding the Enemy class, update the Game class' start() method to invoke Enemy's start() method to initialize the enemy, and update Game class' render() method to invoke Enemy's draw() method.

Implement:

Scene.java → new methods

```
public static int getWidth() {
    return width;
}

public static int getHeight() {
    return height;
}
```

Note: *We must add these getter methods because we declared the class variables within each class as* `private`*. This encapsulates the data within the class, to ensure no other class can change the values of those variables without the class' permission.*

Enemy.java

```
public class Enemy {
```

```
    //Enemy data --> accessible by all enemy methods, scoped for as long as Game class runs.
    private static String image;
    private static int width;
    private static int height;
    private static double x;
    private static double y;

    //draw enemy
    public static void draw() {
        StdDraw.picture(x+width/2, y+height/2, image);
    }

    public static void start() {
        image = "assets/target.png";
        width = 32;
        height = 32;
        x = Math.random() * Scene.getWidth() - width;
        y = Math.random() * Scene.getHeight() - height;
    }
}
```

Game.java → render()

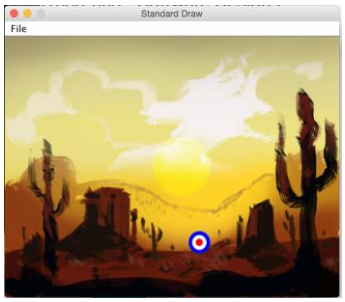
```
public static void render() {
    Scene.draw();       //draw scene
    Enemy.draw();       //draw enemy
    StdDraw.show(100); //show graphics
}
```

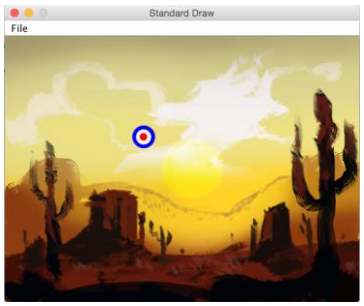
Game.java → start()

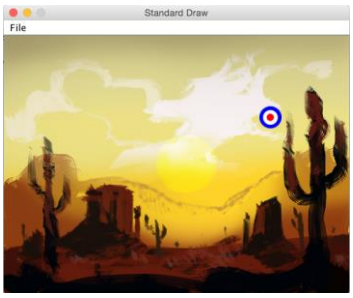
```
public static void start() {
    Scene.start();   //setup scene data
    Enemy.start();   //setup enemy data
}
```

Test:

Compile and execute. The game should now randomly draw the enemy image on top of the scene image.







*Above shows various screenshots of the expected output; Notice that the enemy image should randomly spawn somewhere within the scene image.

Iteration	**[Major Update] Class: Game *(Add Game Loop)***

4	*Game Loop & Game update to move enemy*

Plan:

Goal 4:
Move the enemy around the scene each time the game's update method is invoked in game loop.

Enemy image width, height x, y draw start move	*The Enemy class contains all data/methods that model the enemy.* The Enemy class needs a new behavior (method) to achieve our goal: **move:** randomize new values for x and y variables of Enemy
Game gameOver main render update start	*The Game class contains all data/methods that model game's rules* The Game class needs a new loop control variable for the game loop. Add the game loop into the main method, where the body of the loop repeatedly invokes the Game's **update()** and **render()** methods. Finally, to get the Enemy to move, call the **Enemy.move()** method within the Game's **update** method.

Implement:

Enemy.java → move()

```
public static void move() {
    x = Math.random() * Scene.getWidth() - width;
    y = Math.random() * Scene.getHeight() - height;
}
```

Enemy.java → update()

```
public static void update() {
    move();
}
```

Game.java → update()

```
public static void update() {
    Enemy.update();         //update enemy data
}
```

Game.java → class data

```
private static boolean gameOver = false;
```

Game.java → main()

```
public static void main(String[] args) {
    start();                          //Start Game Setup
    while (gameOver == false) {       //Game loop
        update();                     //Update game data
        render();                     //Render game
    }
}
```

Test:

Compile and execute. The game should now repeatedly randomize the enemy's location at high speed.

Iteration	**Use System API to measure time** *Move enemy after 1 second duration*

5

Plan:

Goal 5:
Slow down the enemy's movement such that it only occurs after a one second duration, instead of on every pass of the game update.

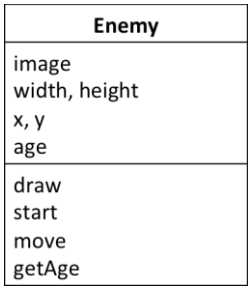
How to measure and compare time in software?
To do this, we must model and track the passage of time in between loops. The System class provides a method for doing this, as seen in the API.

System API

`static long`	**`currentTimeMillis()`** Returns the current time in milliseconds.

Source: *https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis()*

Below shows how we plan to use this method to track changes in time and respond accordingly.



The Enemy class contains all data/methods that model the enemy.

The Enemy class needs a new attribute *(variable)* and a new behavior *(method)* to achieve our goal:
time: represents the Enemy's starting time. (timestamp)

In the Enemy's ***start()*** & ***move()*** methods, reset time variable with the new current time value.

Afterwards, refactor the Game class's ***update()*** method, to get the current time, and subtract it from the Enemy's time, and if the difference is a second or greater, then invoke the Enemy's move method.

Implement:

Enemy.java → class data

```
private static long time;
```

Enemy.java → start()

```
public static void start() {
```

```
    image = "assets/target.png";
    width = 32;
    height = 32;
    x = Math.random() * Scene.getWidth() - width;
    y = Math.random() * Scene.getHeight() - height;
    time = System.currentTimeMillis();
}
```

Enemy.java → move()

```
public static void move() {
    x = Math.random() * Scene.getWidth() - width;        //new x position
    y = Math.random() * Scene.getHeight() - height;      //new y position
    time = System.currentTimeMillis();                    //new time for time
}
```

Enemy.java → update()

```
public static void update() {
    long now = System.currentTimeMillis();
    if (now - time > 1000) {
        Enemy.move();                                  //update enemy
    }
}
```

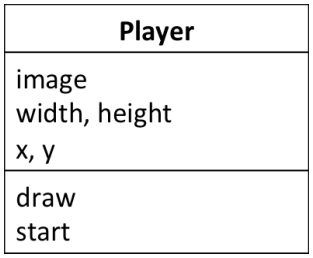
Test:

Compile and execute. The game should now wait to update the enemy's location in 1 second intervals.

Iteration 6	**[Create] Class: Player** *(Contains all methods/data that manages the player in game)*

Plan:

Goal 6:
Draw the Player to the screen *(using the StdDraw class)*



The Player class will contain all data/methods that model the player.

To draw the player to the scene, we need 5 pieces of data:
name (with file path), the width, the height, and the location in the canvas.

At this stage, the Player class only need two behaviors to achieve the goal:
start: initialize all of the data for the player by settings its data variables
draw: draw the player"s image to the canvas

After adding the Player class, update the Game class' ***start()*** method to invoke Enemy's ***start()*** method to initialize the enemy, and update Game class' ***render()*** method to invoke Enemy's ***draw()*** method.

Implement:

Player.java

```
public class Player {
    //Player data.
    private static String image;
    private static int width;
    private static int height;
    private static double x;
    private static double y;
}
```

Player.java → start()

```
public static void start() {
    image = "assets/aimer.png";
    width = 32;
    height = 32;
    x = Scene.getWidth()/2;
    y = Scene.getHeight()/2;
}
```

Player.java → draw()

```
public static void draw() {
    StdDraw.picture(x+width/2, y+height/2, image);
}
```

Game.java → start()

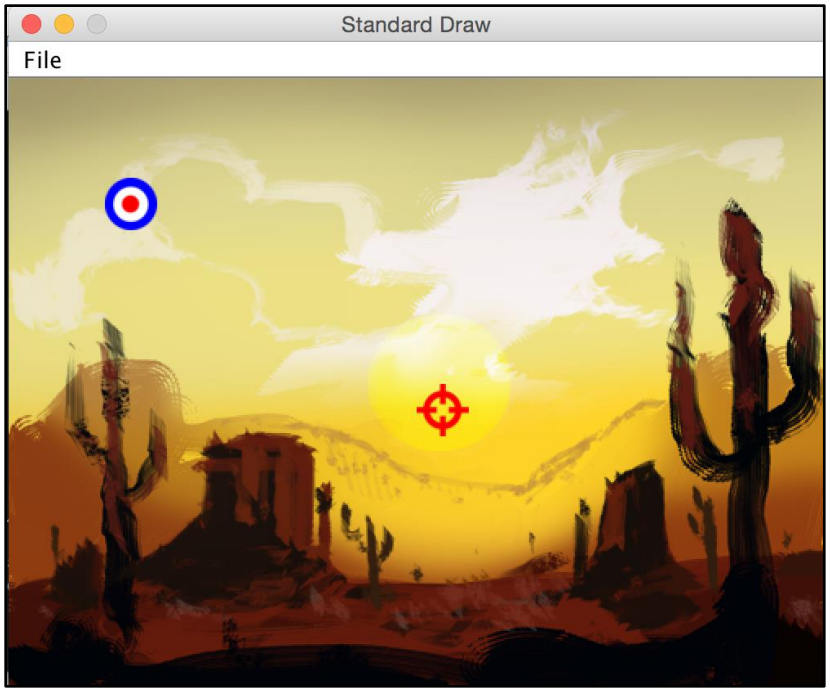
```
public static void start() {
    Scene.start();   //setup scene data
    Enemy.start();   //setup enemy data
    Player.start();  //setup player data
}
```

Game.java → render()

```
public static void render() {
    Scene.draw();        //draw scene
    Enemy.draw();        //draw enemy
    Player.draw();       //draw player
    StdDraw.show(100);   //show graphics
}
```

Test:

Compile and execute the code. The game now renders a player's image in the center of the Scene.



Iteration 7	**[Update] Class: Player *(Add mouse controls)*** *(Contains all methods/data that manages the player in game)*

Plan:

Goal 7:
Move the Player using mouse controls.

Player
image width, height x, y
draw start move

The Player class will contain all data/methods that model the player.

To move the player in the scene, we need to update 2 pieces of data: the location (x, y) coordinates in the canvas.

The Player class only needs 1 new behavior to achieve the goal:
move: update player's position based on cursor position.
See StdDraw API (Appendix A) for a method to get mouse's x and y values.

After adding the move method into the Player class, update the Game class' ***update()*** method to invoke Player's ***move()*** method to constantly update the player position with mouse position.

Implement:

Player.java → move()

```
public static void move() {
    x = StdDraw.mouseX() - width/2;
    y = StdDraw.mouseY() - height/2;
}
```

Player.java → update()

```
public static void update() {
    move();       //update player
}
```

Game.java → update()

```
public static void update() {
    Player.update();       //update player
    Enemy.update();        //update enemy
}
```

Test:
Compile and execute the code. The player image should now follow the cursor as it moves in the Scene.

Iteration	**[Update] Class: Enemy, Player**

8	*Add Collision detection to the Game*

Plan:

Goal 8:
Add collision detection into the Enemy class.

How to measure space in software?
We must check if the left side of both image overlaps with the right side of other image and must check if the top side of both images overlaps the bottom side of other image.

[Picture]

Player image width, height x, y draw start move getX, getY getHeight, getWidth	*The Player class will contain all data/methods that model the player.* The player needs to give its (x,y) location, width, and height to the enemy class so that it can compare the Player to the Enemy positions. The Player class needs 4 getter methods to achieve this goal: **getLeft(), getRight(), getBottom(), getRight():** Gets positional values.
Enemy image width, height x, y age draw start move getAge getX, getY, getHeight, getWidth isTouching	*The Enemy class will contain all data/methods that model the enemy.* The Enemy class will perform the collision detection. To make the logic more readable, implement getters for the Enemy too. The Enemy class needs 4 getter methods: **getLeft(), getRight(), getBottom(), getRight():** Gets positional values. The Enemy class then needs a **isTouching()** method. **isTouchingX(), isTouchingY()** returns boolean if touching on an axis. **isTouching()**: returns boolean if the sides of the player & enemy overlap.
After adding the ***isTouching()*** method into the Enemy class, update the Game class' ***update()*** method to invoke Enemy''s ***move()*** method if the player is touching the enemy.	

Implement:

Player.java → getter methods

```
public static double getLeft() {
    return x;
}

public static double getTop() {
    return y;
}

public static double getBottom() {
    return y + height;
}

public static double getRight() {
    return x + width;
}
```

Enemy.java → getter methods

```
public static double getLeft() {
    return x;
}

public static double getTop() {
    return y;
}

public static double getBottom() {
    return y + height;
}

public static double getRight() {
    return x + width;
}
```

Enemy.java → isTouchingX method (Collision Detection on X-axis)

```
public static boolean isTouchingX() {
    return Player.getLeft() <= Enemy.getRight() && Enemy.getLeft() <= Player.getRight();
}
```

Enemy.java → isTouchingY method (Collision Detection on Y-axis)

```
public static boolean isTouchingY() {
    return Player.getTop() <= Enemy.getBottom() && Enemy.getTop() <= Player.getBottom();
}
```

Enemy.java → isTouching method (Collision Detection)

```
public static boolean isTouching() {
    return isTouchingX() && isTouchingY();
}
```

Enemy.java → update()

```
public static void update() {
    long now = System.currentTimeMillis();          //get NOW time
    if (now - time > 1000) {              //check if over 1000 millisecs passed
        Enemy.move();                               //update enemy
    }
    if ( Enemy.isTouching() ) {
        Enemy.move();
    }
}
```

Test:

Compile and execute the code. The enemy image should now move in response to the player touching it..

Iteration 9	**[Update] Class: Game** *Add Scoring into the Game*

Plan:

Goal 9:
Add score counter as HUD to the game.

Game
gameOver score
main render update start

The Game class will contain all data/methods for managing the game's rules.

The Game class needs a new attribute *(variable)*:
score: represents the game score. (integer)

In the Game's ***update()*** method, increment the the score whenever the player touches the enemy. In the Game's **render()** method, call StdDraw to display the score as text in the Canvas.

Implement:

Game.java → class data

```
private static int score = 0;
```

Game.java → addScore()

```
public static void addScore(){
    score++;                                    //increment score
}
```

Game.java → render()

```
public static void render(){
    Scene.draw();                               //draw scene
    Enemy.draw();                               //draw enemy
    Player.draw();                              //draw player
    StdDraw.text(64,32,"Score: " + score);      //draw score
    StdDraw.show(100);                             //show graphics
}
```

Enemy.java → update()

```
public static void update() {
    long now = System.currentTimeMillis();          //get NOW time
    if (now - age > 1000) {                         //check if over 1000 millisecs passed
        Enemy.move();                               //update enemy
    }
    if ( Enemy.isTouching()  ) {
        Enemy.move();
        Game.addScore();
```

```
    }
}
```

Test: Compile and execute the code.

Iteration 10	**[Update] Class: Player** *Add click to attack to the game*

Plan:

Goal 10:
Add click events to the game with StdDraw..

Player
image width, height x, y isAttacking
draw start move getX, getY getHeight, getWidth isAttacking attack

Implement:

Player.java → class data

```
private static boolean isAttacking;
```

Player.java → isAttacking()

```
public static boolean isAttacking() {
    return isAttacking;
}
```

Player.java → start()

```
public static void start() {
    image = "assets/aimer.png";
    width = 32;
    height = 32;
    x = Scene.getWidth()/2;
    y = Scene.getHeight()/2;
    isAttacking = false;
}
```

Player.java → new method

```
public static void attack() {
    if ( StdDraw.mousePressed() ) {
        isAttacking = true;
```

```
    }
    else {
        isAttacking = false;
    }
}
```

Player.java → update()

```
public static void update() {
    move();       //player move logic
    attack();     //player attack logic
}
```

Enemy.java → update()

```
public static void update() {
    long now = System.currentTimeMillis();          //get NOW time
    if (now - time > 1000) {              //check if over 1000 millisecs passed
        Enemy.move();                               //update enemy
    }
    if ( Enemy.isTouching() && Player.isAttacking()  ) {
        Enemy.move();
        Game.addScore();
    }
}
```

Test:

Compile and execute the code. The game should now be complete.

Your Goal: Refactor Zombie Game *(50 points)*

Instructions:

Refactor your Zombie Apocalypse Game into a Graphics-based game with StdDraw class.

You should consider how to separate the game logic from the display logic, in a fashion similar to how it's been done here (Game.java, Scene.java, Player.java, Enemy.java).

You should also consider how to turn your old main method into a series of smaller methods. This refactoring should be in the HW3 folder and the HW2 folder version should remain unchanged.
(30 points)

For this project, you must source your own artwork, either by finding it online or making it yourself
(10 points)

Free game art:
https://opengameart.org/

Online Pixel editor:
https://www.piskelapp.com/

Write a readme.txt file detailing the custom features you added into your version of the game, explain how it works, and how to play.
(10 points)

Submitting:

Submit all homework files to both the lab's gitlab repo and Moodle (zipped file):

- TargetClicker (folder)
  - Game.java
  - Player.java
  - Enemy.java
  - Scene.java
- ZombieApocalyse (folder)
  - Game.java
  - Player.java
  - Enemy.java
  - Scene.java
  - *Any other class files for your game project*
- readme.txt

Appendix A: StdDraw API *(Abridged)*

StdDraw API *(Abridged)*

https://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html

setCanvasSize

```
public static void setCanvasSize(int canvasWidth,
                                 int canvasHeight)
```

Sets the canvas (drawing area) to be *width*-by-*height* pixels. This also erases the current drawing and resets the coordinate system, pen radius, pen color, and font back to their default values. Ordinarly, this method is called once, at the very beginning of a program.

Parameters:

`canvasWidth` - the width as a number of pixels

`canvasHeight` - the height as a number of pixels

Throws:

`IllegalArgumentException` - unless both `canvasWidth` and `canvasHeight` are positive

setXscale

```
public static void setXscale(double min,
                             double max)
```

Sets the *x*-scale to the specified range.

Parameters:

`min` - the minimum value of the *x*-scale

`max` - the maximum value of the *x*-scale

Throws:

`IllegalArgumentException` - if `(max == min)`

setYscale

```
public static void setYscale(double min,
                             double max)
```

Sets the *y*-scale to the specified range.

Parameters:

`min` - the minimum value of the *y*-scale

`max` - the maximum value of the *y*-scale

Throws:

`IllegalArgumentException` - if `(max == min)`

show

```
public static void show()
```

Copies offscreen buffer to onscreen buffer. There is no reason to call this method unless double buffering is enabled.

picture

```
public static void picture(double x,
                           double y,
                           String filename)
```

Draws the specified image centered at (*x*, *y*). The supported image formats are JPEG, PNG, and GIF. As an optimization, the picture is cached, so there is no performance penalty for redrawing the same image multiple times (e.g., in an animation). However, if you change the picture file after drawing it, subsequent calls will draw the original picture.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

Throws:

`IllegalArgumentException` - if the image filename is invalid

picture

```
public static void picture(double x,
                           double y,
                           String filename,
                           double scaledWidth,
                           double scaledHeight)
```

Draws the specified image centered at (*x*, *y*), rescaled to the specified bounding box. The supported image formats are JPEG, PNG, and GIF.

Parameters:

`x` - the center *x*-coordinate of the image

`y` - the center *y*-coordinate of the image

`filename` - the name of the image/picture, e.g., "ball.gif"

`scaledWidth` - the width of the scaled image (in screen coordinates)

`scaledHeight` - the height of the scaled image (in screen coordinates)

Throws:

`IllegalArgumentException` - if either `scaledWidth` or `scaledHeight` is negative

`IllegalArgumentException` - if the image filename is invalid

pause

```
public static void pause(int t)
```

Pause for t milliseconds. This method is intended to support computer animations.

Parameters:

`t` - number of milliseconds