

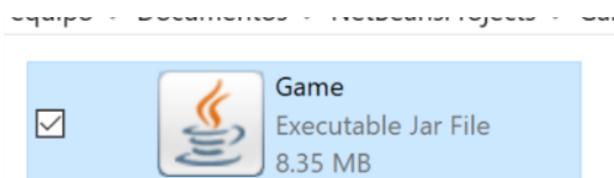
Todas las clases



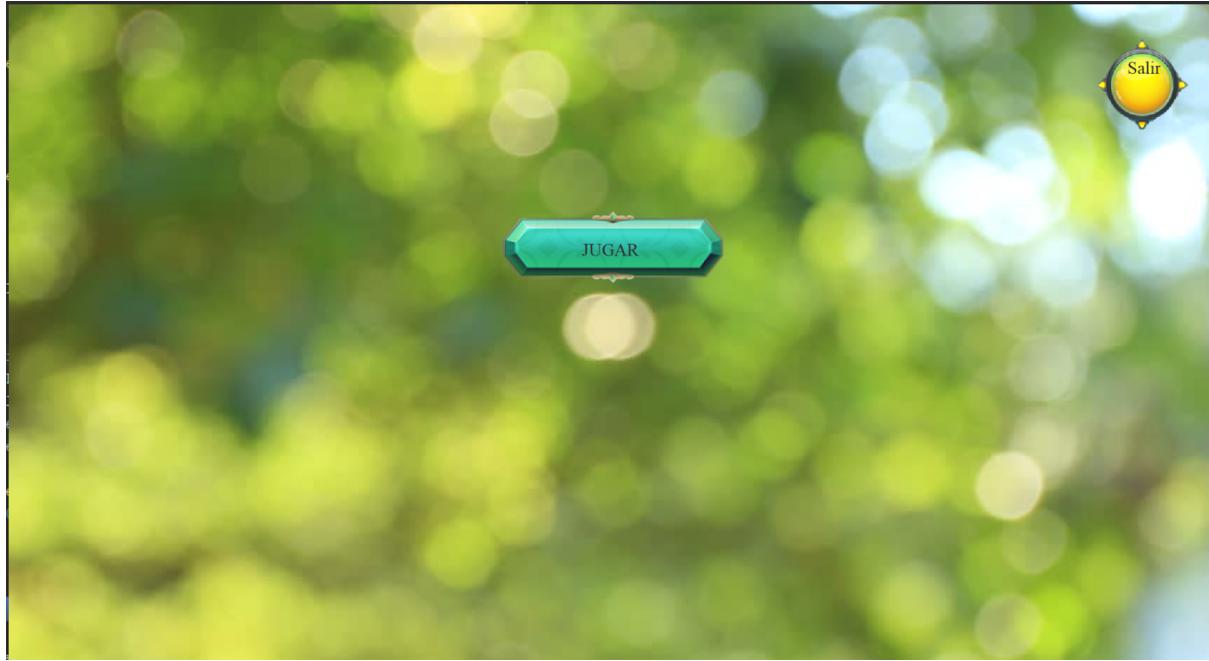
Lista de Técnicas

- El uso de la librería Swing de java para el manejo de gráficas
- El uso de Threads junto con el método CyclicBarrier y notifyAll y wait.
- El manejo de Rectangle para las colisiones y hitbox
- La creación de algoritmos para el manejo de daño y vida dependiendo del nivel

Archivo del programa



Pantalla principal



Creación de la ventana principal.

```
JFrame window = Settings.getWINDOW();
JLayeredPane parent = new JLayeredPane();
window.addKeyListener(new Key());
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
window.setSize(sizeX, sizeY);
window.setLocationRelativeTo(null);
window.setResizable(false);
window.setUndecorated(true);
```

Se crea una LayeredPane la cual contendrá todos los elementos del menú principal

```
JLayeredPane canvasM = new JLayeredPane();
parent.add(canvasM, new Integer(0));
canvasM.setBounds(0,0, sizeX, sizeX);
window.add(parent);
canvasM.setDoubleBuffered(true);
```

Para cada botón se crea un MouseListener para los respectivos botones.

```

button = ImageIO.read(new File("src/Resources/Images/buttons.png"));
JLabel buttonJ = new JLabel();
buttonJ.setText("JUGAR");
buttonJ.setFont(new Font("Serif", Font.PLAIN, 29));
buttonJ.setHorizontalTextPosition(JLabel.CENTER);
buttonJ.setVerticalTextPosition(JLabel.CENTER);
buttonJ.addMouseListener(new MouseListener() {
    @Override
    public void mouseClicked(MouseEvent e) {
        JLayeredPane canvasL = new JLayeredPane();
        parent.add(canvasL);

        JLabel loading = new JLabel();
        loading.setIcon(new ImageIcon(new ImageIcon("src/Resources/Images>LoadingScreen.jpg").getImage().getScaledInstance(sizeX, sizeY, Image.SCALE_DEFAULT)));
        loading.setBounds(0, 0, sizeX, sizeY);
        canvasL.add(loading, new Integer(0));
        canvasL.setBounds(0, 0, sizeX, sizeY);

        int MAX = 100;
        JProgressBar pb = new JProgressBar();
        pb.setMinimum(0);
        pb.setMaximum(MAX);
        pb.setStringPainted(true);
        pb.setBounds(100, 1000, 1720, 50);
        parent.remove(canvasM);
        canvasL.add(pb, new Integer(1));
        parent.revalidate();
        parent.paintImmediately(0, 0, sizeX, sizeY);
        try {
            Game g = new Game(window, parent, canvasL, pb, lock, pause);
        } catch (InterruptedException ex) {
            Logger.getLogger(Menu.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(Menu.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

@Override
public void mousePressed(MouseEvent e) {

```

Estos métodos se repiten tanto para los botones del menú principal como para el menú de pausa. Se cambia simplemente el texto, la posición y la acción a ejecutar.



Manejo de mapa

Para la imagen del fondo se utilizó una imagen base.



Utilice Tiled para poder crear el mapa y del cual importe a un archivo .csv. Este archivo venia con listas e índices dependiendo de cual recuadro se necesitaba en la posición.

Utilice dos matrices para establecer la capa principal y la capa que contuviera los objetos como la tienda o los troncos. Para poder implementar esto a la ventana principal divide la pantalla en secciones de 32x32 pixeles (El tamaño de cada subimagen de la imagen principal) y dependiendo del index de la matriz MAP inserta la imagen correcta.

```
for (int x = 0, y = 0; y < numSqY; x++) {  
  
    int n = map[y][x];  
    background[x][y] = new JLabel(new ImageIcon(  
        mapSrc.getSubimage(size*(n%16),size*(n/16),size,size)));  
    background[x][y].setBounds(x*size,y*size,size,size);  
    canvas.add(background[x][y],new Integer(0));  
    if(x == numSqX-1){  
        y++;  
        x=-1;  
    }  
}
```

Manejo de Personajes

Para el manejo de los sprites tanto del jugador como del enemigo se utilizó una clase exclusiva que maneja las actualizaciones.

```

Right = PlySettings.isRight();
Down = PlySettings.isDown();
Up = PlySettings.isUp();
Left = PlySettings.isLeft();
space = PlySettings.isSpace();
lastDir = PlySettings.getLastDir();

if((Right^Left) || (Up ^ Down)) {
    num++;
    move = true;
}
if(! (Right||Left||Up||Down)) {
    num++;
    move = false;
};

if(move)
    Skin = ((num%6)+5);
else if(space){
    PlySettings.setAttacking(true);
    try {
        attack(6, 0, 0);
    } catch (InterruptedException ex) {
        Logger.getLogger(PlayerSkin.class.getName()).log(Level.SEVERE, null, ex);
    }
    PlySettings.setAttacking(false);
}
else
    Skin = ((num%5));

switch(lastDir){
    case 0: Settings.setPlySkin(L[Skin]);
    break;
    case 1: Settings.setPlySkin(U[Skin]);
    break;
    case 2: Settings.setPlySkin(R[Skin]);
    break;
    case 3: Settings.setPlySkin(D[Skin]);
    break;
}

```

Se utilizó la variable num como parámetro definiendo el paso del tiempo, se consiguieron algoritmos que resultaron en la posición de la subimagen necesaria, esta posición se aloja en Skin.



Para el ataque se fue necesario un algoritmo más complejo dado que el cuarto ataque consecutivo tiene un número distintos de imágenes que los otros cinco ataques.

```
public void attack(int max, int sk,int inc) throws InterruptedException{
    int posX = PlySettings.getPosX();
    int posY = PlySettings.getPosY();
    boolean mover=false;
    int cantidad = 30;
    int pos = inc;

    for (int i = 0; i < max; i++) {

        shift = PlySettings.isShift();
        Thread.sleep(50);

        if(i==max-1){
            break;
        }
        if(i == (max - 3) && shift && sk+max+11 < 43){
            int limit = (inc==2)? 8: 6;
            attack(limit, sk+max, inc+ 1);
            break;
        }
        Skin= i+11+sk;
        if(inc == 0&&i ==1)
            areaAtaque(lastDir);
        if((inc>0&&i == 1 && max == 6)){
            cantidad = 30;
            mover = true;

            if(pos ==4)
                areaAtaqueU();
            else
                areaAtaque(lastDir);
        }else if(i==2 && max ==8){
            cantidad = 90;
            mover = true;
            areaAtaque(lastDir);
        }else{
            mover = false;
        }

        switch(lastDir){
```

```

switch(lastDir) {
    case 0: Settings.setPlySkin(L[Skin]);
    if(mover) {
        PlySettings.setPosX(posX-cantidad);
        posX -= cantidad;
    }
    break;
    case 1: Settings.setPlySkin(U[Skin]);
    if(mover) {
        PlySettings.setPosY(posY-cantidad);
        posY -= cantidad;
    }
    break;
    case 2: Settings.setPlySkin(R[Skin]);
    if(mover) {
        PlySettings.setPosX(posX+cantidad);
        posX += cantidad;
    }
    break;
    case 3: Settings.setPlySkin(D[Skin]);
    if(mover) {
        PlySettings.setPosY(posY+cantidad);
        posY += cantidad;
    }
    break;
}

```

Para este algoritmo se utilizó la recursividad como método para poder diferenciar qué ataque se está utilizando y ejecutar el código acorde. Para el manejo de las imágenes del enemigo se utilizaron los mismos métodos excepto la recursividad. Esto ya que el enemigo tiene un único ataque.

Manejo de la profundidad

Decidí utilizar LayeredPane como contenedor para los elementos gráficos gracias a su característica de profundidad. Esto ayudaría a controlar la perspectiva de la profundidad del personaje y los enemigos. Para esto en un array se guardan todos los componentes que

existen en la LayeredPane que contiene a los personajes. Después los ordena dependiendo de su posición en el eje Y les asigna su posición de profundidad dependiendo de este ordenamiento.

```
temp = new ArrayList<cha>;
Collections.sort(temp, (a, b) -> a.getc().gety() + a.getc().getheight()
    < b.getc().gety() + b.getc().getheight() ? -1 : a.getc().gety() + a.getc().getheight()
    == b.getc().gety() + b.getc().getheight() ? 0 : 1);

for (int i = 0; i < temp.size(); i++) {
    canvas.setLayer(temp.get(i).getc(), i);
}
```

Manejo de Multi Threads

La principal problemática al usar los multi threads que surgió durante el programa fue la sincronización entre todos. Para esto utilicé CyclicBarrier (Como CountDownLatch no puede ser reutilizado preferí el método que si puede ser reutilizado). Cada CyclicBarrier establece el inicio y la finalización de los procesos. Y existe una clase que se encarga de sincronizar estos Cyclic Barriers.

```
public void run() {

    while(true){
        try {
            cb.await();
        } catch (InterruptedException ex) {
            Logger.getLogger(PlyManager.class.getName()).log(Level.SEVERE, null, ex);
        } catch (BrokenBarrierException ex) {
            Logger.getLogger(PlyManager.class.getName()).log(Level.SEVERE, null, ex);
        }

        Settings.getCHAR().setBounds(PlySettings.getPosX(), PlySettings.getPosY(), 128, 160);
        Settings.getCHAR().setIcon(Settings.getPlySkin());
        try {
            cb2.await();
            cb3.await();
        } catch (InterruptedException ex) {
            Logger.getLogger(PlyManager.class.getName()).log(Level.SEVERE, null, ex);
        } catch (BrokenBarrierException ex) {
            Logger.getLogger(PlyManager.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

En este caso PlyManager se encarga de sincronizar las clases PlayerSkin y PlyMovement. EnemyManager se encarga de EnemyLimiter y EnemySkin. La clase que controla todos los threads y se encarga de actualizar la ventana es Window. Se encarga de sincronizar LayerManger, PlyManager y EnemyManager.

```

double enc = 0;
int mult = 1;
int time;
new Enemy(Settings.getCANVAS_2());
while(true){
    try {
        cb.await();
    } catch (InterruptedException | BrokenBarrierException ex) {
        Logger.getLogger(Window.class.getName()).log(Level.SEVERE, null, ex);
    }
    long start = System.currentTimeMillis();

    //enc = ((game.Level.getLevel())/(5+game.Level.getLevel()))+.2;
    curTime = System.nanoTime();
    time = (int)((curTime-startTime)/10000000000);
    if(time%5==0 && mult*5==time){
        mult++;
        new Enemy(Settings.getCANVAS_2());
    }

    counter++;
    canvas.repaint();
    canvas3.repaint();
    try {
        pauseThread();
    } catch (InterruptedException ex) {
        Logger.getLogger(Window.class.getName()).log(Level.SEVERE, null, ex);
    }
    long end = System.currentTimeMillis();
    long delay = ((20-(end-start))>=0)? 20-(end-start): 0;

    // System.out.println(delay);
    try {
        Thread.sleep(delay);
    } catch (InterruptedException ex) {
        Logger.getLogger(Window.class.getName()).log(Level.SEVERE, null, ex);
    }
    try {
        cb2.await();
    } catch (InterruptedException | BrokenBarrierException ex) {
        Logger.getLogger(Window.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

Pausa de Threads para el menú

Para poder pausar toda la ejecución del juego cuando se abre el menú de pausa se utilizó Thread.wait() y Thread.notifyAll(). Además de esto se utilizó una variable para poder pausar desde otro thread.

```

private void pauseThread () throws InterruptedException{

    synchronized (lock)
    {
        if (Menu.isPause())
            lock.wait();
    }
}

@Override
public void mouseClicked(MouseEvent e) {
    synchronized(lock){
        menu.removeAll();
        menu.repaint();
        lock.notifyAll();
        pause = false;
    }
}

```

Manejo de Colisiones

Para poder manejar colisiones entre los personajes y objetos se creó otra clase en la cual se crearía un objeto rectángulo para manejar intersecciones. Para los objetos del mapa se crean estos rectángulos desde el comienzo y se guardan en una lista arr. Esto es checado por plyMovement y Enemy para verificar si hay colisión.

```

static{
    pj = new Collision(new Rectangle(250, 250, 240, 580));
    logs = new Collision(new Rectangle(40, 125, 170, 150));
    bag = new Collision(new Rectangle(620+100, 100, 100, 150));
    fruit = new Collision(new Rectangle(815+50, 100, 350, 150));
    grain = new Collision(new Rectangle(1200,25,460,200));
    store = new Collision(new Rectangle(1570,185,90,160));

    arr.add(logs);
    arr.add(bag);
    arr.add(fruit);
    arr.add(grain);
    arr.add(store);
}

```

Para agregar los enemigos se agregó una lista nueva (arre).

```

public int addE(int x, int y, Enemy e) {
    Collision Enemy = new Collision(new Rectangle(250, 250, 96, 96), contador);
    Enemy.setR(x, y);
    Enemy.setE(e);
    contador++;
    arre.add(Enemy);
    return contador;
}

```

Como el jugador solo colisiona con los objetos del mapa solo debe checar la lista arr.

```

public boolean checkIfClear(int x, int y) {
    boolean flag = true;

    pj.setR(PlySettings.getPosX() + x, PlySettings.getPosY() + y);

    for (int i = 0; i < arr.size(); i++) {
        if (pj.isInterjecting(arr.get(i))) {
            flag = false;
        }
    }

    px = pj.getTemp().x;
    py = pj.getTemp().y;

    if (px < 40 || px > 1740 || py < 0 || py > 840) {
        flag = false;
    }

    return flag;
}

```

Los enemigos deben de checar en cambio la colisiones entre ellos. Es entonces que se checa la lista arre, que contiene la posición de todos los enemigos.

```

public Object checkIfClearEa(int indexM, int a, int b) {
    Rectangle EneInt = new Rectangle();
    boolean flag = true;
    Collision enemy2 = arr.get(indexM);
    enemy2.setAhead(a, b);

    for (int i = 0; i < arr.size(); i++) {
        if (indexM == i)
            continue;
        if (arr.get(i) == null) continue;
        if (enemy2.isInterjectingE(arr.get(i))) {
            flag = false;

            if (enemy2.getTemp().getLocation().distance(arr.get(i).getTemp().getLocation()) < enemy2.getTemp().getLocation().distance(EneInt.getLocation()))
                EneInt = arr.get(i).getTemp();
        }
    }

    px = (int) enemy2.getTemp().getX();
    py = (int) enemy2.getTemp().getX();

    if (flag) {
        return true;
    } else {
        return EneInt;
    }
}

```

Manejo de ataques

Para los ataques se utiliza un método parecido a las colisiones. Exceptuando que se modifica la posición del rectángulo dependiendo en qué dirección va el enemigo. Para el enemigo son solo dos posiciones y para el jugador cinco (la quinta es el ataque en área).

```
public void attack(int max, int sk,int inc) throws InterruptedException{
    int posX = PlySettings.getPosX();
    int posY = PlySettings.getPosY();
    boolean mover=false;
    int cantidad = 30;
    int pos = inc;

    for (int i = 0; i < max; i++) {

        shift = PlySettings.isShift();
        Thread.sleep(50);

        if(i==max-1){
            break;
        }
        if(i == (max - 3) && shift && sk+max+11 < 43){
            int limit = (inc==2)? 8: 6;
            attack(limit, sk+max, inc+ 1);
            break;
        }
        Skin= i+11+sk;
        if(inc == 0&&i ==1)
            areaAtaque(lastDir);
        if((inc>0&&i == 1 && max == 6)){
            cantidad = 30;
            mover = true;

            if(pos ==4)
                areaAtaqueU();
            else
                areaAtaque(lastDir);
        }else if(i==2 && max ==8){
            cantidad = 90;
            mover = true;
            areaAtaque(lastDir);
        }else{
            mover = false;
        }

        switch(lastDir){
```

```

public boolean atkPj(int lastDir) {
    boolean flag = false;
    Rectangle atk;
    switch(lastDir) {
        case 0:
            atk = new Rectangle((int)(pj.getTemp().getX()-(pj.getTemp().getWidth()/2)),
                (int)pj.getTemp().getY(),(int)pj.getTemp().getWidth(),
                (int)pj.getTemp().getHeight());
            break;
        case 1:
            atk = new Rectangle((int)(pj.getTemp().getX()),
                (int)(pj.getTemp().getY()-(pj.getTemp().getHeight()/2)),
                (int)pj.getTemp().getWidth(),(int)pj.getTemp().getHeight());
            break;
        case 2:
            atk = new Rectangle((int)(pj.getTemp().getX())+(pj.getTemp().getWidth()/2),
                (int)pj.getTemp().getY(),(int)pj.getTemp().getWidth(),
                (int)pj.getTemp().getHeight());
            break;
        case 3:
            atk = new Rectangle((int)(pj.getTemp().getX()),
                (int)(pj.getTemp().getY())+(pj.getTemp().getWidth()/2),
                (int)pj.getTemp().getWidth(),(int)pj.getTemp().getHeight());
            break;
        case 4:
            atk = new Rectangle((int)(pj.getTemp().getX()-(pj.getTemp().getWidth()/2)),
                (int)(pj.getTemp().getY()-(pj.getTemp().getHeight()/2)),
                (int)(pj.getTemp().getWidth())+(pj.getTemp().getWidth()/2)),
                (int)(pj.getTemp().getHeight())+(pj.getTemp().getWidth()/2));
            break;
        default:
            atk = null;
            break;
    }

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

```

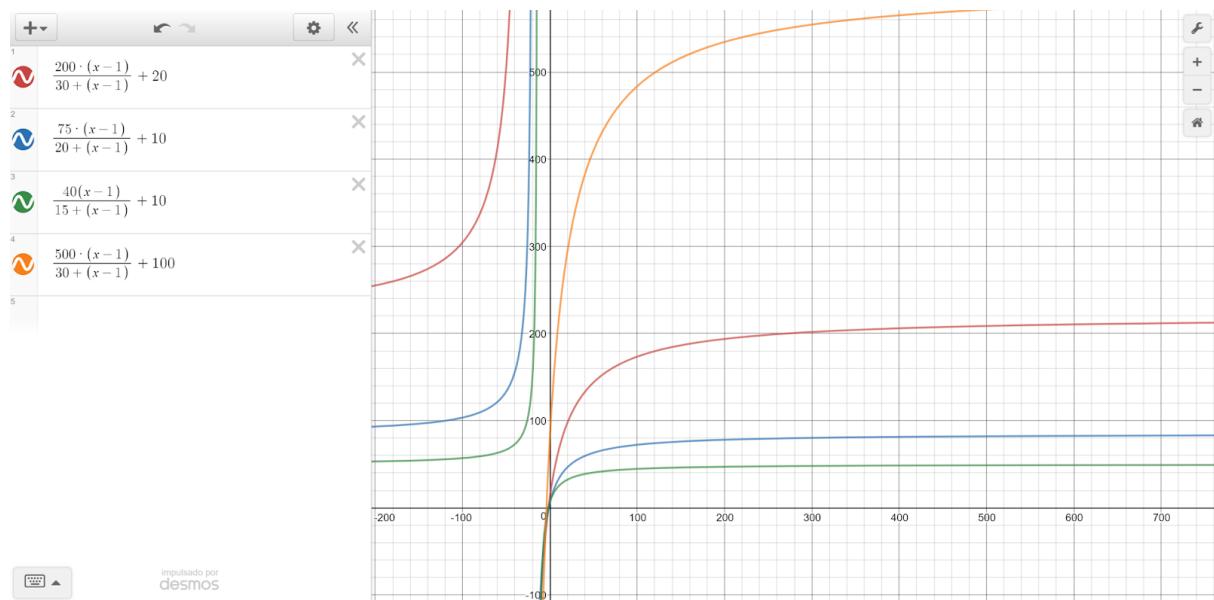
```

            for (int i = 0; i < arre.size(); i++) {
                try{
                    if(atk.intersects(arre.get(i).getTemp()))
                        arre.get(i).damage(((200*Level.getLevel())/(30+Level.getLevel()))+20);
                        flag = true;
                }catch(Exception e){
                    continue;
                }
            }
            return flag;
        }
    }

```

Manejo de nivel y escala de poder

Para poder manejar el incremento de niveles y dar al jugador un sentimiento de progresión era necesario implementar un sistema de niveles. Pero para evitar una progresión lineal y falta de sentimiento de logro por parte del jugador fue necesario pensar fórmulas no lineales. Para esto simplemente límites en las ecuaciones y las fui modificando para que existiera una curva relativamente suave de crecimiento de poder.



La clasificación es la siguiente:

- Roja: Ataque del jugador
- Azul: Vida restaurada por nivel del jugador
- Verde: Ataque del enemigo
- Naranja: Vida del enemigo

Para implementarlo cree una simple clase nivel.

```

public class Level extends Thread {
    static int level = 0;

    public void lvlup(){
        level++;
        Settings.getUI().lvlup(level);
    }

    public static int getLevel() {
        return level;
    }

    public void run(){
        List<Collision> arre = Collision.getArrE();
        while(true){

            if((arre.size()+1)%6==5){
                lvlup();
                System.out.println("LVL UP");
                new Enemy(Settings.getCANVAS_2());
            }
        }
    }
}

```

Esta clase sería la encargada de ir variando (x-1). De esta manera simplemente implemente las fórmulas en los distintos puntos y cree un método el cual curará al personaje cuando subiera de nivel.

```

public void lvlup(int i){
    if(ene==null){
        hpLeft += (int) ((75*i)/(20+i))+10;
        if(hpLeft>100) hpLeft = 100;
    }
}

```

Manejo de vidas y muertes

El manejo de vidas y muertes con respecto a los enemigos es llevado a cabo por una clase que se encarga de manejar la vida tanto de los enemigos como del jugador. Además de esto se encarga de modificar el método paintComponent(Graphics g) para poder agregarlo a la ventana del juego.

```

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setColor(Color.RED);
    g2.fillRect(5,5,hpLeft,height);
    g2.setColor(Color.black);
    g2.setStroke(new BasicStroke(5,
                                BasicStroke.CAP_ROUND,
                                BasicStroke.JOIN_ROUND));
    g2.drawRect(5,5,total,height);
}

}

```

Este método tiene distintos constructores dependiendo que se necesitaba, utilizando así el polimorfismo que nos permite java. Si es la instancia que se encargará de manejar la vida del personaje no se utilizan parámetros algunos. Si es para la vida de algún enemigo el parámetro es la clase enemigo.

```

/*
public class DrawRect extends JLabel {

    private int total = 200;
    private int hpLeft = total;
    private int height = 25;
    Enemy ene = null;
    Window wnd = new Window();

    DrawRect() {
    }

    DrawRect(Enemy e){
        height = 10;
        total = (((500*(Level.getLevel()))/(30+(Level.getLevel())))+100);
        hpLeft = total;
        ene = e;
    }
}

```

Para modificar la vida de los personajes se llama al método Damage

```
        }

    public void Damage(int cantidad) {
        hpLeft -= cantidad;
        if(hpLeft<=0)
            if(ene!=null)
                ene.muerto();
            else
                System.exit(0);

    }
}
```

Aquí diferencia entre sí es una instancia del enemigo o del jugador definiendo si la variable ene es null. Cuando el personaje murió se termina el programa. En cambio si el enemigo muerto se llama al método muerto() de la clase enemigo.

```
public void muerto() {

    canvas.remove(enemy);
    col.delete(index);
    Settings.getCANVAS_3().remove(ES.getDamageRef());
    vivo = false;
    ES.muerto();
    Limi.muerto();
    EnMa.muerto();

}
```

Todos los métodos *muerto* llamados dentro de este método modifica la variable vivo a false, la cual es una variable que mantiene los ciclos de los threads funcionando. También busca remover la gráfica del enemigo de la pantalla.