Trabajo Práctico **“GONDOLF”**

Juego en Java con Entorno

***Integrantes:***

*Diego Julián García –* [*diegojuliangarcia15@gmail.com*](mailto:diegojuliangarcia15@gmail.com)

*Diego Facundo Duarte –* [*facundoduarte679@gmail.com*](mailto:facundoduarte679@gmail.com)

***Docentes:***

*Verónica Moyano -*[**vmoyano@campus.ungs.edu.ar**](mailto:vmoyano@campus.ungs.edu.ar)

*Cesar Niveyro -*[**cniveyro@campus.ungs.edu.ar**](mailto:cniveyro@campus.ungs.edu.ar)

Programación I

**Introducción**

El siguiente trabajo practico será presentado dentro del marco de la materia Programación 1, consiste en el desarrollo de un videojuego que gira en torno a “Gondolf”, un mago que se enfrenta a murciélagos utilizando hechizos.

El objetivo principal del proyecto fue la implementación de mecánicas básicas del juego en el lenguaje Java. Se crearon clases y métodos que permitieron la creación y aparición de personajes, movimientos, disparos, interacción con obstáculos, objetos, entre otros.

Además, se prestó atención a la forma de organizar el código, con el fin de mantener de forma correcta las responsabilidades de las clases y sus métodos. El resto del documento se dedicará a explicar las clases utilizadas y, de forma sintética, los puntos más importantes, tales como métodos que permitan funciones de gran utilidad. Los atributos/variables como x, y, diámetro, serán omitidos ya que suelen repetirse en todas las funciones y aunque sean necesarios para el funcionamiento son secundarios en la explicación.

**Descripción**

**Clase Piedra**

Esta clase representa los obstáculos estáticos del escenario. Las piedras tienen posición y tamaño, y sirven como límites que tanto el mago como los enemigos deben esquivar. No se mueven, pero se dibujan cada vez que se actualiza la pantalla.

Variables de instancia:

x, y: representan la posición en la pantalla donde se dibuja la piedra.

imagenPiedra: almacena la imagen que se va a dibujar.

diametro: determina el tamaño (o el área de colisión en las funciones que es precisado).

Los métodos utilizados fueron:

dibujarPiedra(Entorno e): Se encarga de dibujar la piedra en pantalla.

crearPiedras(): Se responsabiliza de crear un arreglo de piedras y establecerle una posición de la pantalla a cada piedra.

Además,

en la clase juego se utilizó su implementación para distribuirlas en forma de arreglo de objetos.

Se creó el arreglo de piedras con un for, estableciendo a cada piedra una posición x e y, luego en la sección tick() se dibujaron como el resto de clases.

Problemas:

Esta clase fue bastante sencilla de implementar y en términos generales no presentó ninguna gran dificultad. Quizás, la función encargada de la aparición de las piedras, inicialmente lo habíamos planteado como un generador aleatorio, pero presentaba algunas fallas como la aparición sobre el jugador o esquinas. Por ello, si bien podría haber sido resuelto, decidimos que era mejor crear un arreglo con la posición de cada piedra, donde las establezcamos simétricamente en seis lugares de la pantalla.

**Clase Mago**

Esta clase es la encargada de darle vida al personaje principal, el mago Gondolf. En ella se reconocen sus colisiones, posición o tamaño.

Variables de instancia:

x, y: Coordenadas del centro mago.

diam: Diametro del mago.

velocidad: Velocidad de movimiento en pixeles por tick.

imagenGondolf: Imagen del personaje extraída por una herramienta para ser dibujada en pantalla.

Los métodos utilizados fueron:

void moverIzquierda(Piedra[] piedras)

void moverDerecha(Piedra[] piedras)

void moverArriba(Piedra[] piedras)

void moverAbajo(Piedra[] piedras)

Mueven al mago en la dirección correspondiente si no colisiona con los bordes del entorno ni con una piedra.  
Utilizan un sistema preventivo de colisiones que primero calcula la nueva posición y valida que sea posible antes de mover al personaje

void dibujarMago(Entorno e)

Dibuja al mago en su posición actual utilizando la imagen cargada.

boolean colisionPiedra(double nuevoX, double nuevoY, Piedra[] piedras)

Verifica si el mago colisionaría con alguna piedra al intentar moverse a una nueva posición. Si es así, devuelve true para evitar el movimiento.

Problemas:

Encontramos un problema en la colisión de piedras, donde más que un error, fue una cuestión de analizar cómo podría ser implementada lógicamente, ya que los radios no eran suficiente. Nos llevó algo de tiempo, pero encontramos que con el uso del teorema de Pitágoras podíamos medir la distancia entre ambos radios y con ello detectar si existía una colisión.

**Clase Murcielago**

Esta clase representa a los enemigos del juego. Los murciélagos se mueven de forma autónoma, cambian de dirección aleatoriamente y pueden desaparecer al salir de los límites. También se encarga de sus colisiones y de dibujarlos en pantalla

Variables de instancia:

double x, y: Coordenadas del centro del murciélago en el entorno.

double diam: Diámetro del murciélago (Lo utilizamos para su tamaño y colisiones).

double velocidad: Double que representa la velocidad a la que el murciélago se desplaza al seguir al mago, se utiliza como una variable para otros métodos.

Image imagenMurcielago: Imagen cargada y escalada del murciélago.

Los métodos utilizados fueron:

void dibujarMurcielago(Entorno e) Dibuja la imagen del murciélago en su posición actual.

boolean colisionaConMago(Mago mago)

Se utiliza para determinar si el murciélago está colisionando con el mago. Usa el teorema de Pitágoras para calcular la distancia entre ambos y compara esa distancia con la suma de sus radios.

void seguir(Mago mago)

Se encarga de hacer que el murciélago se desplace hacia el mago. Calcula la dirección usando la diferencia de posiciones y reescribe el valor para moverse en la dirección correcta con la velocidad ya definida. Si está muy cerca (menos de 1 píxel de distancia), no se mueve para evitar temblores.

crearMurcielagos(Murcielago [] arreglo, int murcielagosCreados)

Este método se encarga de crear de a 10 murciélagos en las posiciones indicadas (donde los puntos indiquen) le asigna una coordenada a cada uno y se ejecuta hasta que se llegue al parámetro de murciélagos creados llegue al tope, en este caso, 50.

Problemas:

Quizás el método que nos presentó más adversidad fue el del movimiento hacia el mago. Si bien ya sabíamos cómo implementarlo, con la posición del mago, la del murciélago y la comparación entre ambos, nuevamente como la clase anterior, la distancia jugaba un papel importante. Por lo que una vez habíamos razonado e implementado la distancia, la función funcionaba correctamente.

**Clase Pantalla**

Esta clase divide visualmente el juego. Se encarga de dibujar el fondo de la zona jugable y la zona de poderes. También muestra las pantallas de derrota y victoria según el estado del juego.

Variables de instancia:

anchoJuego, altoJuego: tamaño del área de juego (600x600 píxeles).

anchoPoderes: ancho del área lateral destinada a los poderes (200 píxeles).

fondoJuego: imagen de fondo de la zona jugable.

fondoPoderes: imagen de fondo del área de poderes.

Los métodos utilizados fueron:

dibujarPantalla(Entorno entorno) Dibuja el fondo de la zona jugable, centrado en (300, 300). Utiliza la función dibujarImagen.

dibujarPoderes(Entorno entorno) Dibuja el fondo del menú lateral derecho de poderes.

dibujarDerrota(Entorno entorno) Muestra la pantalla de derrota, la cual sucede si la vida del mago llego a 0. Fondo negro, texto rojo y blanco con mensajes personalizados.

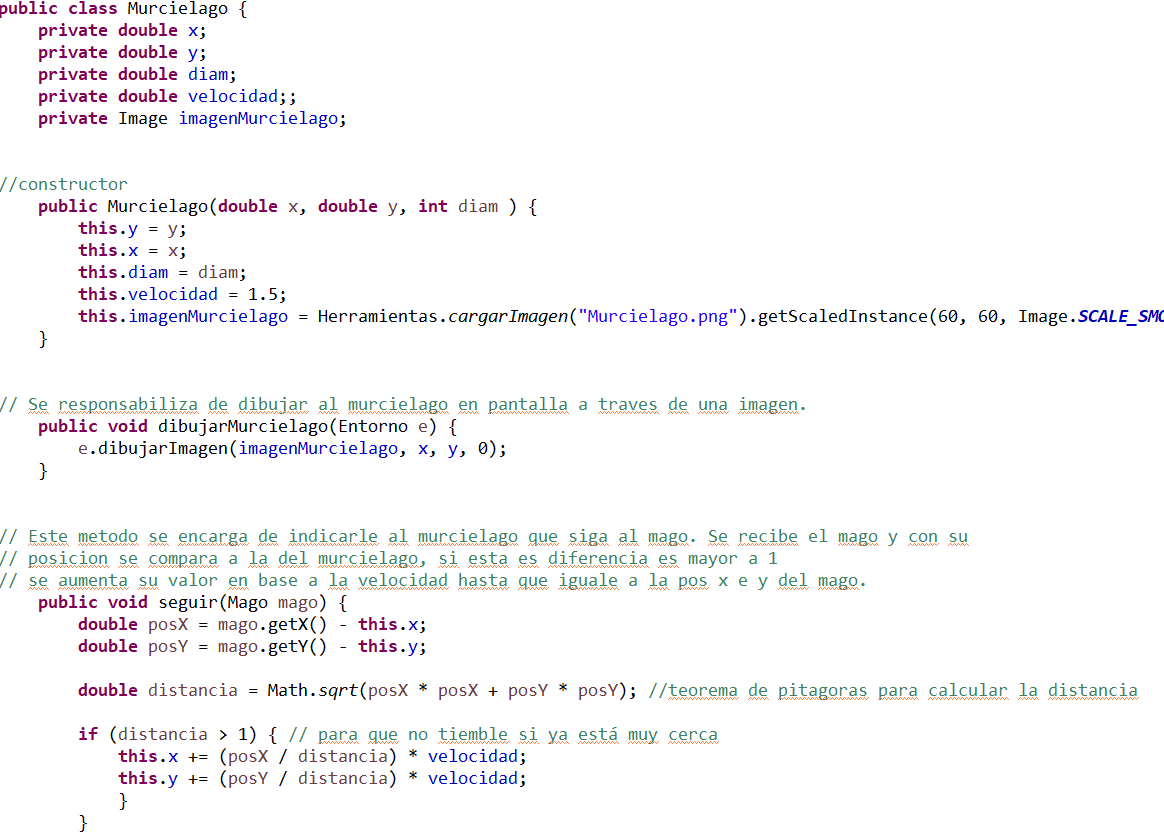
dibujarVictoria(Entorno entorno) Tiene una ejecución y tipado prácticamente igual al de la derrota, se encarga de mostrar la pantalla de victoria, es decir, cuando el mago elimino los 50 murciélagos. Con mismo fondo negro, texto verde y blanco.

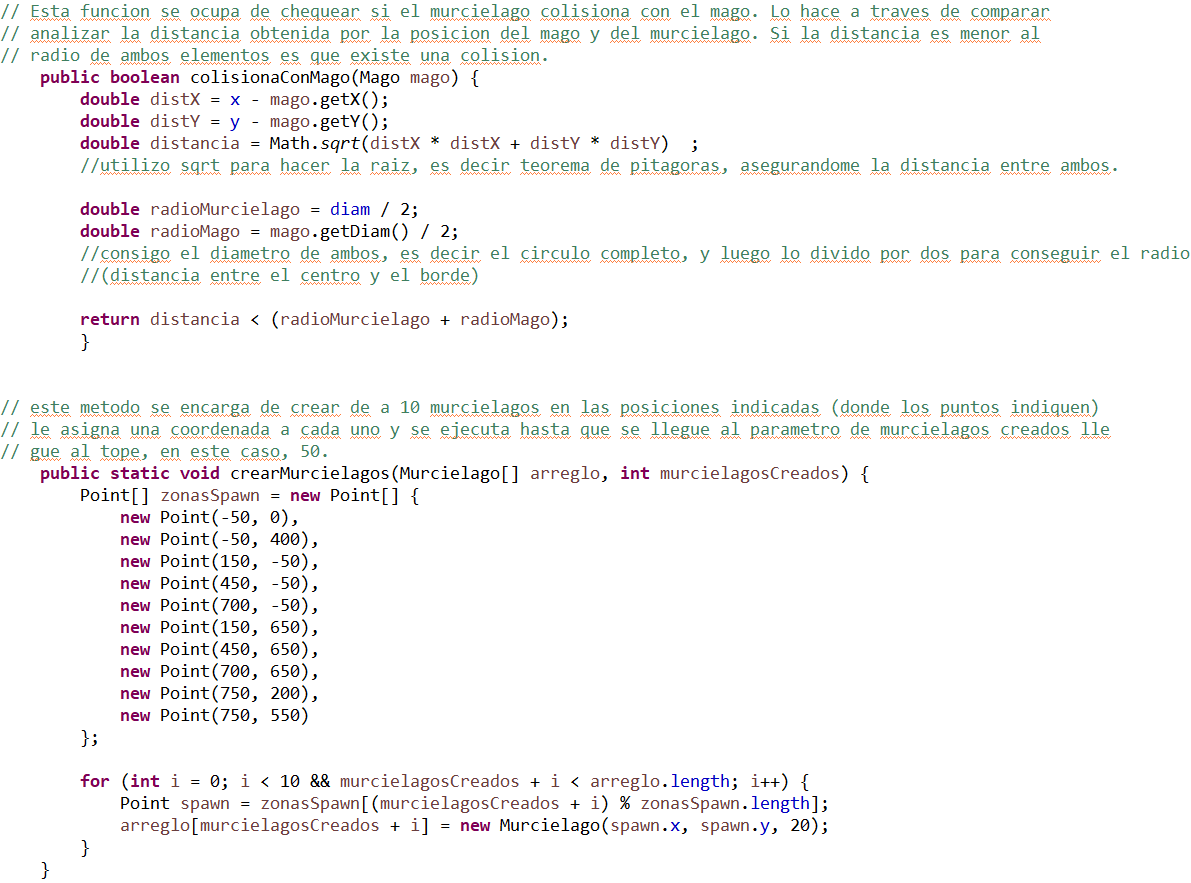
Problemas:

Nos enfrentamos al problema de que las imágenes de fondo de las zonas jugables y de poderes se ajustaban de forma extraña, a pesar de que indicábamos la posición y tamaño de cada una, se intercambiaban de lugar, posición, o incluso dejaban espacios. Nuestra solución fue escalar las imágenes a una resolución común, (600x600), e ir probando constantemente hasta que ambas coincidieran correctamente. Si bien funciono, la implementación de sus medidas puede verse un poco alterada debido a que, si bien las resoluciones eran las mismas, se ajustaban diferente, incluso con las mismas direcciones.

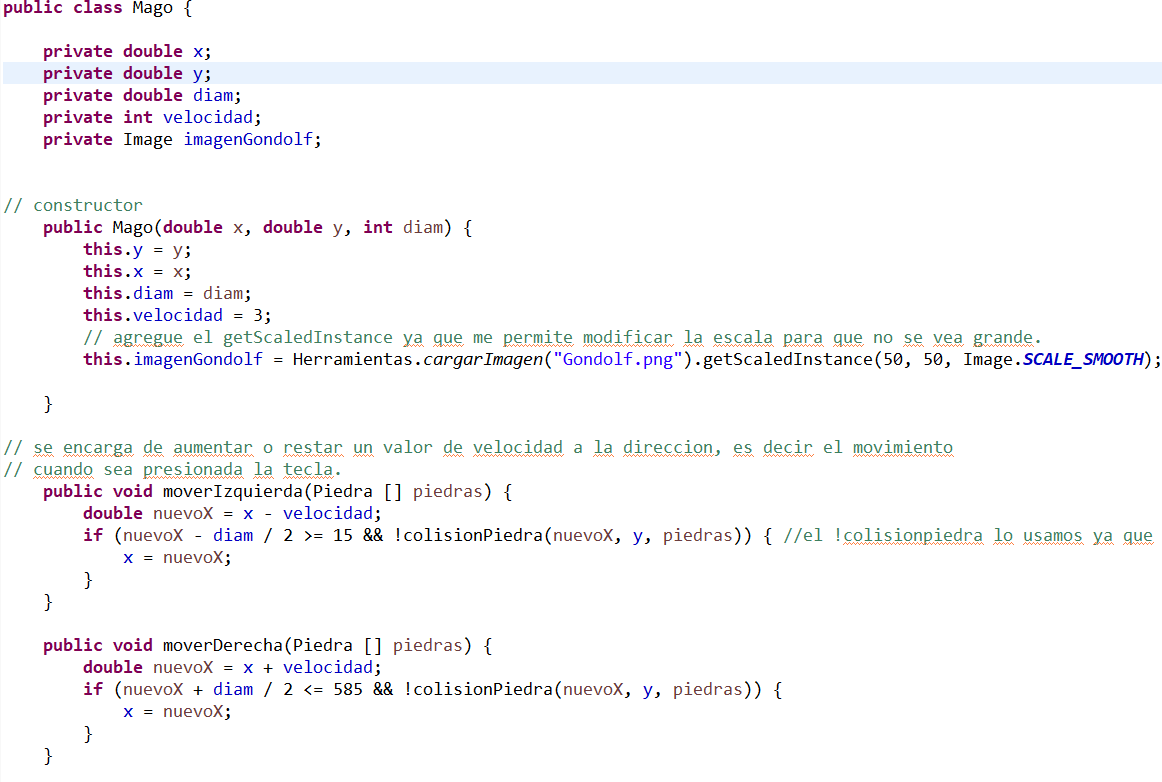
**Implementación**

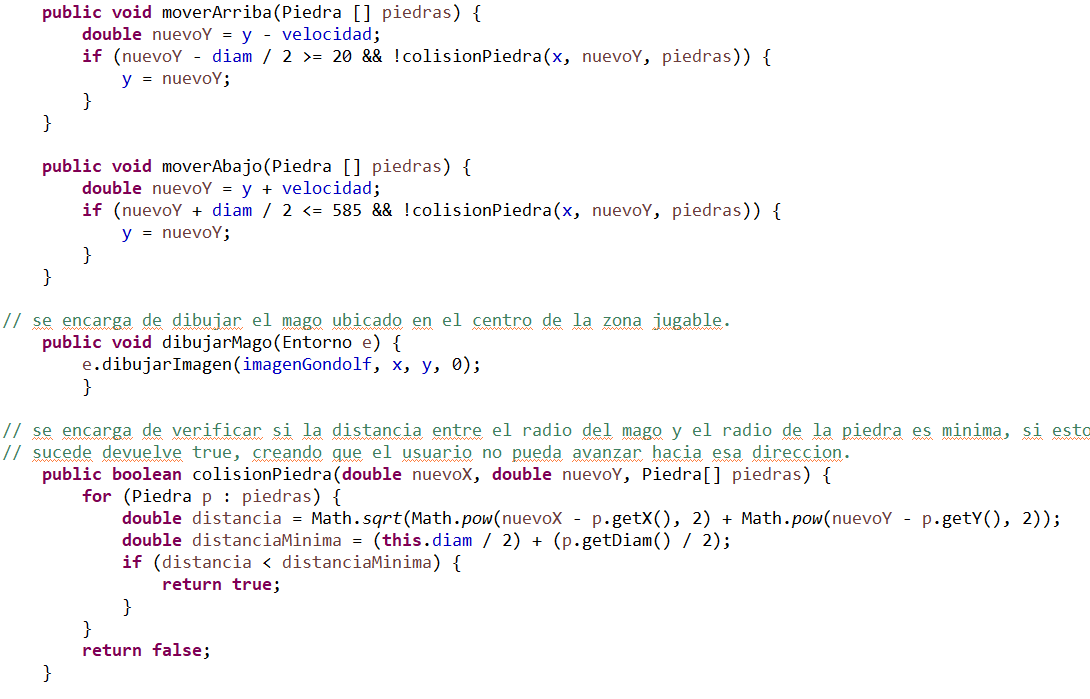
**Clase murciélago**



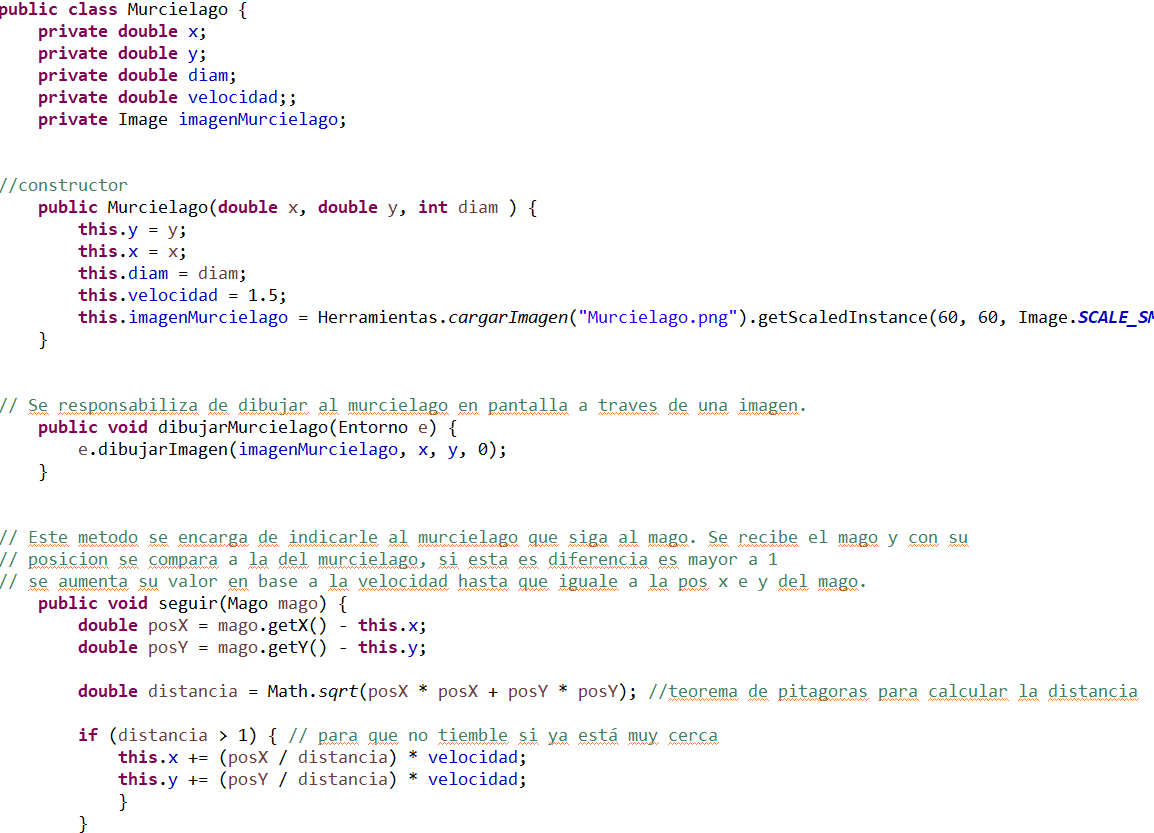


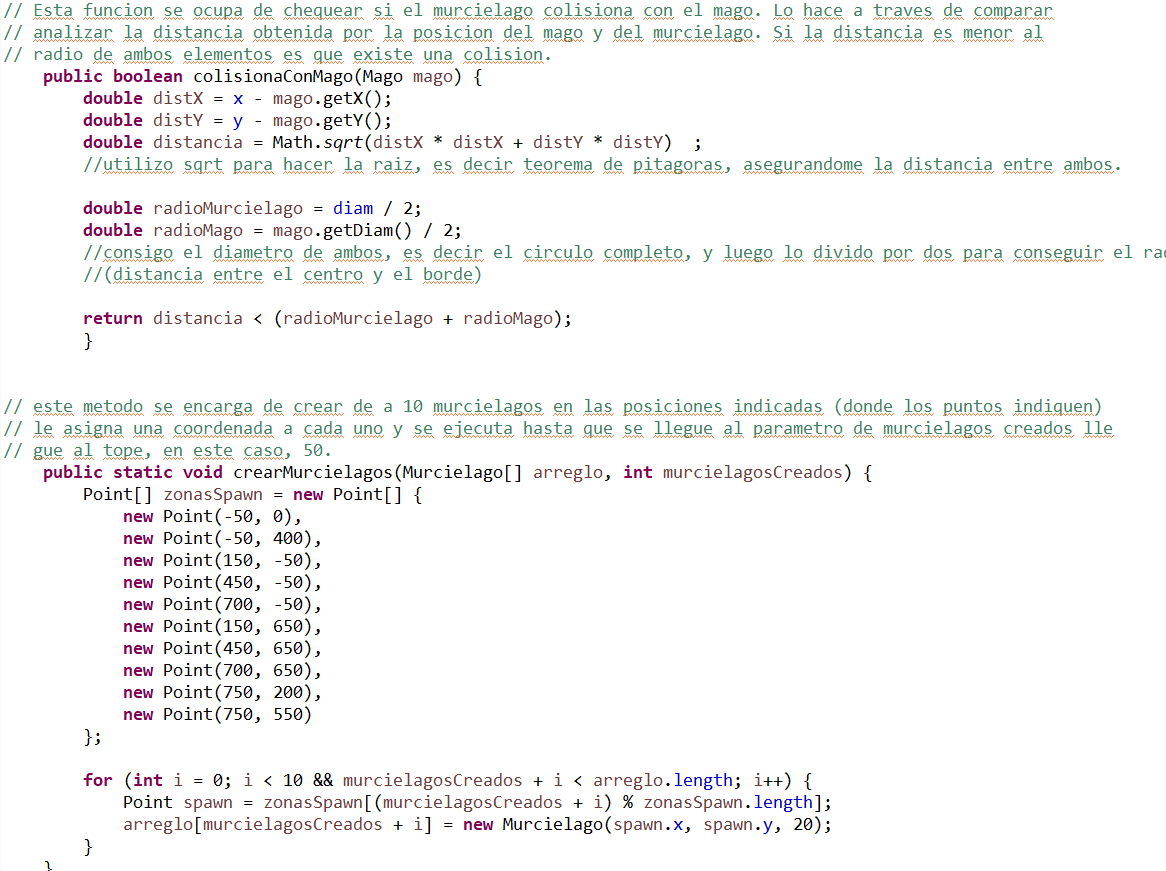
**Clase Mago**

****

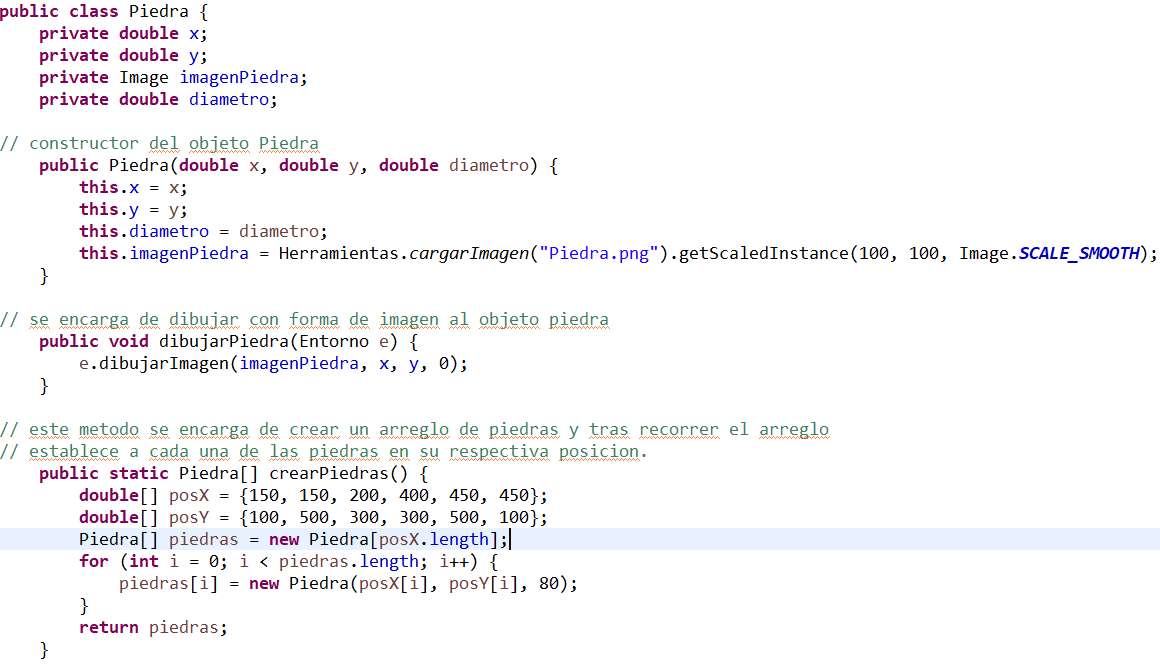
****

**Clase Murciélago**

****

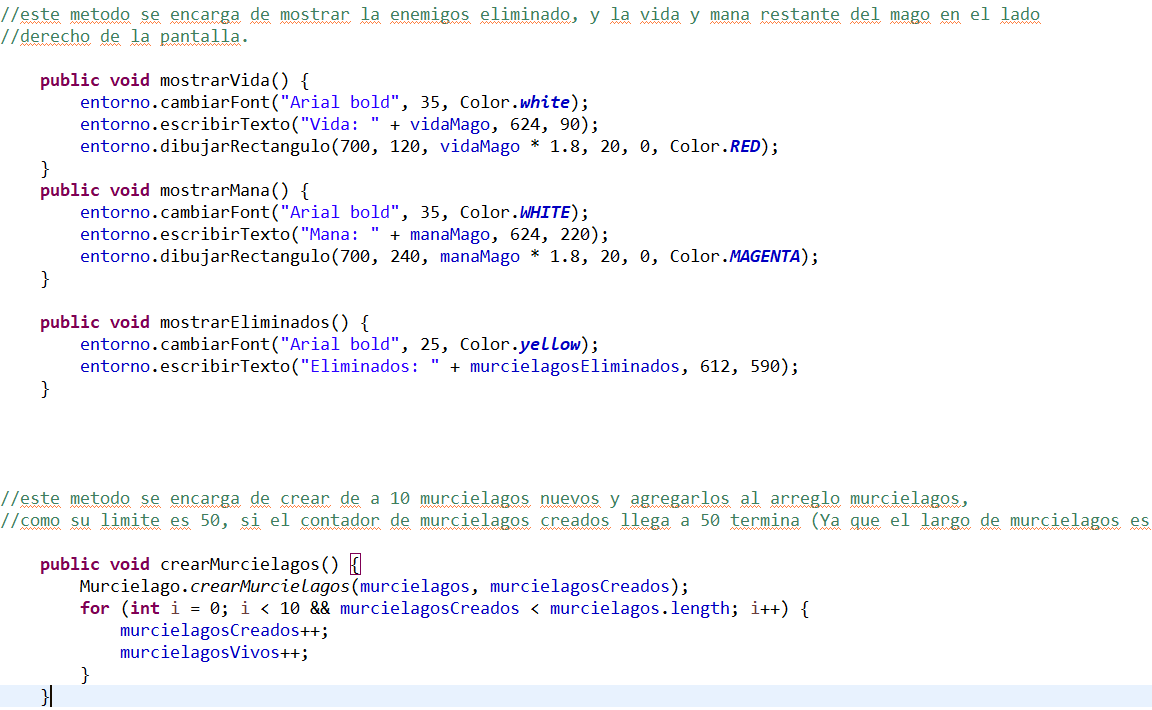


**Clase Piedra**

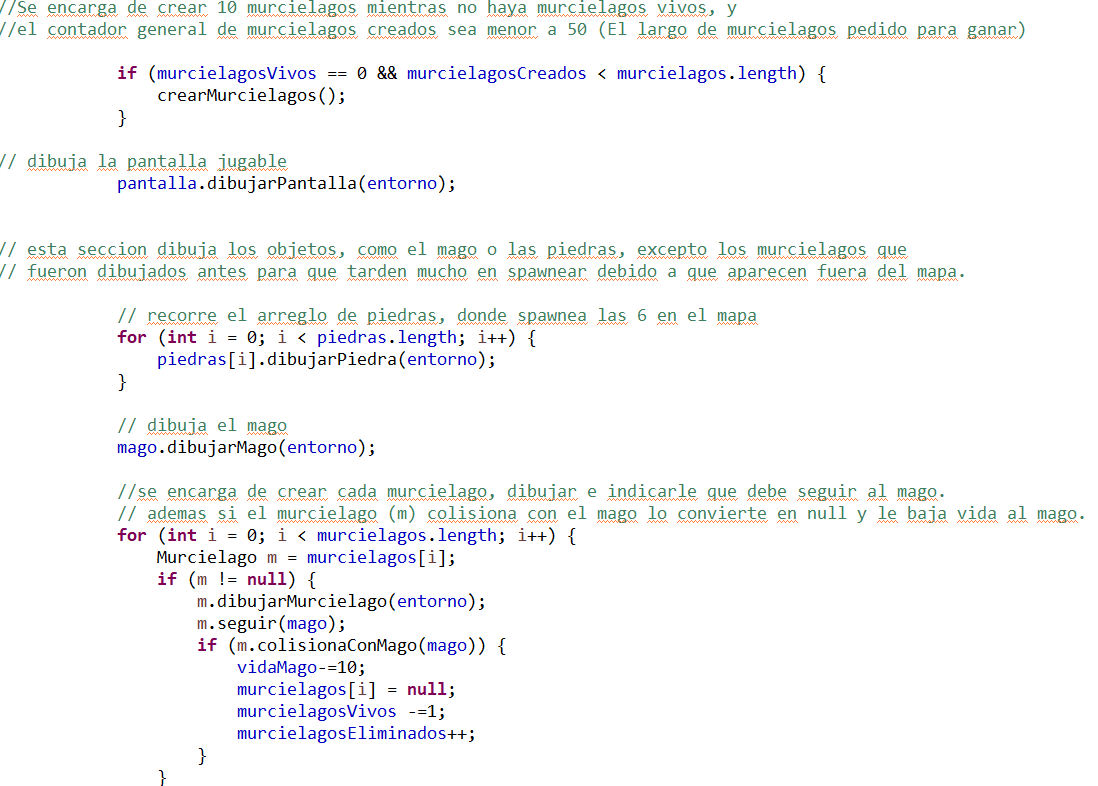
****

**Clase Juego**

Falta esta parte

****

Falta esta parte

****

Falta esta parte



