Manual técnico

PISCIFACTORÍA

Álex Varela Cascallar
ACCESO A DATOS

Contenido

Introducción

Este manual tiene como objetivo proporcionar información detallada sobre el funcionamiento, la configuración y el uso de Piscifactoría, un sistema basado en Java. Está diseñado para ayudar a desarrolladores, administradores de sistemas y cualquier persona interesada en comprender y trabajar con el código fuente de este proyecto.

Objetivo del manual

- Brindar una visión general de la estructura y el propósito del proyecto.
- Explicar cómo utilizar las diversas funciones y componentes del código.
- Proporcionar consejos y pautas para la personalización y adaptación del proyecto.
- Ayudar a los usuarios a comprender y solucionar problemas comunes.

El manual se dividirá en las distintas clases que tendrá el programa, explicando los distintos métodos que cada una contiene.

Clase Monedas

La clase Monedas está diseñada para gestionar la cantidad de monedas en el monedero, permitiendo realizar operaciones como compras, ventas y agregar monedas a la cantidad existente.

Funciones y métodos

public static Monedas getInstancia()

Descripción: Esta función es un método estático que se utiliza para obtener la instancia única del monedero. Si la instancia aún no existe, se crea una instancia inicial con una cantidad predeterminada de monedas (en este caso, 100).

Uso: Permite acceder a la instancia del monedero para realizar operaciones.

```java
/**
 * Obtiene la instancia única del monedero, creándola si no existe.
 *
 * @return La instancia del monedero.
 */
public static Monedas getInstancia() {
    if (instance == null) {
        instance = new Monedas(cantidad:100);
    }
    return instance;
}
```

public boolean comprobarPosible(int precio)

Descripción: Este método comprueba si es posible realizar una compra con la cantidad de monedas disponible en el monedero. Retorna `true` si es posible y `false` en caso contrario.

Uso: Se utiliza antes de realizar una compra para verificar si hay suficientes monedas para la transacción.

```java
/**
 * Comprueba si es posible realizar una compra con la cantidad especificada.
 *
 * @param precio El precio de la compra.
 * @return true si es posible realizar la compra, false en caso contrario.
 */
public boolean comprobarPosible(int precio) {
    if (cantidad >= precio) {
        return true;
    } else {
        return false;
    }
}
```

public void compra(int precio)

Descripción: Este método permite realizar una compra, disminuyendo la cantidad de monedas en el monedero por el valor del precio de la compra.

Uso: Se utiliza para actualizar la cantidad de monedas después de una compra exitosa.

```
/**
 * Realiza una compra, disminuyendo la cantidad de monedas.
 *
 * @param precio El precio de la compra.
 */
public void compra(int precio){
    this.cantidad-=precio;
}
```

public void venta(int cantidad)

Descripción: Este método permite realizar una venta, aumentando la cantidad de monedas en el monedero por el valor del precio de la venta.

Uso: Se utiliza para aumentar la cantidad de monedas después de una venta exitosa.

```
/**
 * Realiza una venta, aumentando la cantidad de monedas.
 *
 * @param precio El precio de la compra.
 */
public void venta(int precio){
    this.cantidad+=precio;
}
```

public void agregarMonedos(int monedas)

Descripción: Este método permite aumentar la cantidad de monedas en el monedero al agregar una cantidad especificada.

Uso: Útil para aumentar la cantidad de monedas en el monedero.

```
/***
 * Método para aumentar las monedas
 * @param monedas Cantidad a aumentar
 */
public void agregarMonedos(int monedas){
    this.cantidad+=monedas;
}
```

Para utilizar esta clase siempre deberemos recurrir al método getInstancia(), ya que se trata de un singleton.

Clase Stats

La clase Stats extiende la clase Estadísticas y proporciona una instancia única para el seguimiento de estadísticas. Esta clase se utiliza para crear y gestionar estadísticas dentro de una aplicación, lo que facilita el seguimiento y análisis de datos relevantes. A continuación, se describen las funciones y métodos clave de la clase Stats.

Funciones y métodos

public static Stats getInstancia(String[] nombres)

Descripción: Este método se utiliza para obtener la instancia única de Stats. Si la instancia no existe, se crea una nueva instancia de Stats con los nombres de los peces especificados en forma de un array de cadenas.

Uso: Al proporcionar un array de nombres de peces, se crea o se recupera una instancia de Stats que se utilizará para llevar un registro de esas estadísticas específicas. Este método solo se utiliza la primera vez que se llama a Stats, para poder crear la instancia del contructor de Estadísticas.

```java
/**
 * Obtiene la instancia única de Stats. Si no existe, crea una nueva instancia.
 *
 * @param nombres Un array de cadenas que representa los nombres de las estadísticas a seguir.
 * @return La instancia única de Stats.
 */
public static Stats getInstancia(String[] nombres) {
    if (instance == null) {
        instance = new Stats(nombres);
    }
    return instance;
}
```

public static Stats getInstancia()

Descripción: Este método permite obtener la instancia única de Stats sin especificar nombres de estadísticas. Si la instancia no existe, devuelve null.

Uso: Llamada general a la instancia creada del método Stats.

```java
/**
 * Obtiene la instancia única de Stats. Si no existe, devuelve null.
 *
 * @return La instancia única de Stats o null si no existe una instancia previamente creada.
 */
public static Stats getInstancia(){
    return instance;
}
```

Clase AlmacenCentral

La clase AlmacenCentral representa un almacén central y se encarga de gestionar su capacidad y mejoras. Esta clase incluye métodos para aumentar la capacidad del almacén central y realizar mejoras mediante el gasto de monedas. A continuación, se detallan las funciones y métodos clave de la clase AlmacenCentral.

Funciones y métodos

public static AlmacenCentral getInstance()

Descripción: Este método estático se utiliza para obtener la instancia única del AlmacenCentral. Si la instancia aún no existe, se crea una nueva instancia de AlmacenCentral.

Uso: Permite acceder a la instancia del almacén central para gestionar su capacidad y mejoras.

```java
/**
 * Método para obtener la instancia única del Almacén Central.
 *
 * @return La instancia del Almacén Central.
 */
public static AlmacenCentral getInstance() {
    if(instance==null){
        instance=new AlmacenCentral();
    }
    return instance;
}
```

public void aumentarCapacidad(int cantidad)

Descripción: Este método permite aumentar la capacidad máxima del almacén central en una cantidad determinada.

Uso: Se utiliza para mejorar el almacén central al aumentar su capacidad máxima.

```java
/**
 * Aumenta la capacidad máxima del almacén en una cantidad determinada.
 *
 * @param cantidad La cantidad en la que se aumentará la capacidad máxima.
 */
public void aumentarCapacidad(int cantidad){
    this.capacidadMax+=cantidad;
}
```

public void upgrade()

Descripción: Este método permite realizar una mejora en el almacén central, aumentando su capacidad máxima. Para llevar a cabo la mejora, se requiere gastar 100 monedas.

Uso: Se utiliza para realizar mejoras en el almacén central, lo que aumenta su capacidad máxima y se paga con monedas.

```java
/**
 * Realiza una mejora en el almacén central, aumentando su capacidad máxima.
 * Se requiere el gasto de monedas para realizar esta mejora.
 */
public void upgrade(){
    if(Monedas.getInstancia().comprobarPosible(precio:100)){
        Monedas.getInstancia().compra(precio:100);
        this.aumentarCapacidad(cantidad:50);
    }else{
        System.out.println(x:"No tienes monedas suficientes");
    }
}
```

Clase Pez

La clase abstracta Pez representa un pez en una simulación y define sus atributos y comportamientos. Esta clase sirve como base para la creación de diferentes tipos de peces con características específicas. A continuación, se describen las funciones y métodos clave de la clase Pez.

Funciones y métodos

public void showStatus()

Descripción: Este método muestra el estado actual del pez. El cuerpo del método está vacío en la clase abstracta y debe ser implementado en las clases concretas que extiendan **Pez**.

Uso: Se utiliza para mostrar información sobre el estado del pez en la simulación.

```java
public void showStatus() {
    System.out.println("----------" + this.datos.getNombre() + "----------");
    System.out.println("Edad: " + this.edad + "días");
    if (this.sexo) {
        System.out.println(x:"Sexo: M");
    } else {
        System.out.println(x:"Sexo: H");
    }
    if (this.vivo) {
        System.out.println(x:"Vivo: Si");
    } else {
        System.out.println(x:"Vivo: No");
    }
    if (this.maduro) {
        System.out.println(x:"Adulto: Si");
    } else {
        System.out.println(x:"Adulto: No");
    }
    if (this.ciclo == 0) {
        System.out.println(x:"Fértil: Si");
    } else {
        System.out.println(x:"Fértil: No");
    }
    if(this.alimentado){
        System.out.println(x:"Alimentado: Si");
    }else{
        System.out.println(x:"Alimentado: No");
    }
}
```

NOTA: La captura corresponde a un método de un pez específico

public int comer(int comida)

Descripción: Este método verifica si el pez pudo comer en base a la cantidad de comida proporcionada. Retorna la cantidad de comida consumida, siendo 3 si el pez no pudo comer.

Uso: Se utiliza para simular la alimentación del pez y controlar la cantidad de comida consumida.

```java
/**
 * Verifica si el pez pudo comer en base a la cantidad de comida proporcionada.
 *
 * @param comida Cantidad de comida disponible.
 * @return Cantidad de comida consumida, en caso de ser 3 el pez no comió.
 */
public int comer(int comida) {
    if (comida != 0) {
        return 1;
    } else {
        return 3;
    }
}
```

public int grow(int comida, boolean comido)

Descripción: Este método controla el crecimiento del pez en función de la comida proporcionada. Retorna la cantidad de alimento consumido.

Uso: Se utiliza para simular el crecimiento del pez en función de la alimentación y las condiciones.

```java
/**
 * Controla el crecimiento del pez en función de la comida proporcionada.
 *
 * @param comida Cantidad de comida disponible.
 * @param comido Indica si el pez ha sido alimentado por un pez muerto.
 * @return Cantidad de alimento consumido.
 */
public int grow(int comida, boolean comido) {
    if (comido) {
        if (this.vivo == true) {
            this.edad++;
            this.alimentado = true;
            this.comprobarMadurez();
        }
        return 0;
    } else {
        int com = this.comer(comida);
        if (com == 3) {
            this.alimentado = false;
            this.morision();
        }
        if (this.vivo == true) {
            this.edad++;
            this.comprobarMadurez();
            if (com != 3) {
                this.alimentado = true;
                return com;
            }
        }
        return 0;
    }
}
```

public boolean eliminarPez()

Descripción: Este método verifica si el pez muerto debe ser eliminado. Retorna `true` si el pez debe ser eliminado, `false` en caso contrario.

Uso: Se utiliza para determinar si un pez muerto debe ser eliminado de la simulación.

```java
/**
 * Verifica si el pez muerto debe ser eliminado.
 *
 * @return true si el pez debe ser eliminado, false en caso contrario.
 */
public boolean eliminarPez() {
    Random comer = new Random();
    if (comer.nextBoolean()) {
        return true;
    } else {
        return false;
    }
}
```

public void morision()

Descripción: Este método comprueba la muerte del pez en base a un 50% de probabilidades.

Uso: Se utiliza para simular la muerte del pez en función de un factor aleatorio.

```java
/**
 * Comprueba la muerte del pez en base a un 50% de probabilidades.
 */
public void morision() {
    Random muerte = new Random();
    if (muerte.nextBoolean()) {
        this.setVivo(vivo:false);
    }
}
```

public boolean reproduccion()

Descripción: Este método verifica si el pez es apto para la reproducción y retorna true si es apto, false en caso contrario.

Uso: Se utiliza para determinar si el pez cumple con las condiciones necesarias para la reproducción.

```java
/**
 * Verifica si el pez es apto para la reproducción.
 *
 * @return true si el pez es apto para reproducirse, false en caso contrario.
 */
public boolean reproduccion() {
    if (this.maduro && this.edad % this.datos.getCiclo() == 0) {
        if (!this.sexo) {
            this.ciclo = this.datos.getCiclo();
            return true;
        } else {
            return false;
        }

    } else {
        this.ciclo--;
        return false;
    }
}
```

public void comprobarMadurez()

Descripción: Este método comprueba la madurez del pez en función de su edad y actualiza su estado de madurez.

Uso: Se utiliza para determinar si el pez ha alcanzado la madurez y actualizar su estado en consecuencia.

```java
public void comprobarMadurez() {
    if (this.edad >= this.datos.getMadurez()) {
        this.setMaduro(fertil:true);

    } else {
        this.setMaduro(fertil:false);
    }
}
```

public boolean isOptimo()

Descripción: Este método verifica si el pez está en su estado óptimo para la venta.

Uso: Se utiliza para determinar si el pez ha alcanzado un estado óptimo y está listo para ser vendido.

```java
/**
 * Verifica si el pez está en su estado óptimo para la venta.
 *
 * @return true si el pez está en su estado óptimo, false en caso contrario.
 */
public boolean isOptimo() {
    if (this.edad == this.datos.getOptimo()) {
        return true;
    } else {
        return false;
    }
}
```

Clase tanque

La clase Tanque representa un tanque de peces en una piscifactoría y permite gestionar los peces en el tanque, realizando operaciones como alimentar, reproducir y vender. El tanque tiene una capacidad limitada y almacena una colección de peces. A continuación, se describen las funciones y métodos clave de la clase Tanque.

Funciones y métodos

public void showStatus()

Descripción: Este método muestra el estado del tanque, incluyendo la ocupación, cantidad de peces vivos, alimentados, adultos y género.

Uso: Se utiliza para mostrar información detallada sobre el estado del tanque.

```java
/**
 * Muestra el estado del tanque, incluyendo la ocupación, cantidad de peces vivos, alimentados, adultos y género.
 */
public void showStatus() {
    System.out.println("Ocupación: " + this.peces.size() + "/" + this.capacidad + " ("
            + this.porcentaje(this.peces.size(), this.capacidad) + "%)");
    System.out.println("Peces vivos: " + this.vivos() + "/" + this.peces.size() + " ("
            + this.porcentaje(this.vivos(), this.peces.size()) + "%)");
    System.out.println("Peces alimentados: " + this.alimentados() + "/" + this.vivos() + " ("
            + this.porcentaje(this.alimentados(), this.vivos()) + "%)");
    System.out.println("Peces adultos: " + this.adultos() + "/" + this.vivos() + " ("
            + this.porcentaje(this.adultos(), this.vivos()) + "%)");
    System.out.println("Hembras/Machos: " + this.hembras() + "/" + this.machos());
}
```

public void showFishStatus()

Descripción: Este método muestra el estado de cada pez en el tanque.

Uso: Se utiliza para mostrar información detallada sobre el estado de cada pez en el tanque.

```java
/**
 * Muestra el estado de cada pez en el tanque.
 */
public void showFishStatus() {
    for (Pez pez : peces) {
        pez.showStatus();
    }
}
```

public boolean hasDead()

Descripción: Este método comprueba si hay peces muertos en el tanque y los registra.

Uso: Se utiliza para verificar si hay peces muertos y llevar un registro de ellos.

```java
/**
 * Comprueba si hay peces muertos en el tanque y los registra.
 *
 * @return true si hay peces muertos, false en caso contrario.
 */
public boolean hasDead() {
    if (muertos != null) {
        this.muertos.removeAll(muertos);
    }
    for (int i = 0; i < peces.size(); i++) {
        if (!peces.get(i).isVivo()) {
            this.muertos.add(i);
        }
    }
    if (this.muertos != null) {
        if (muertos.size() != 0) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

public int nuevoDiaComer(int comida)

Descripción: Este método realiza el proceso de crecimiento y alimentación de los peces en el tanque, teniendo en cuenta la cantidad de comida disponible.

Uso: Se utiliza para simular el crecimiento y la alimentación de los peces en el tanque.

```java
/**
 * Realiza el proceso de crecimiento y alimentación de los peces en el tanque.
 *
 * @param comida La cantidad de comida disponible para los peces.
 * @return La cantidad de comida consumida.
 */
public int nuevoDiaComer(int comida) {
    int resto = comida;
    int cadaveres = 0;

    if (this.hasDead()) {
        cadaveres = this.muertos.size();
    }
    for (Pez pez : peces) {
        if (pez.isVivo()) {
            if (cadaveres != 0) {
                if (pez.eliminarPez()) {
                    cadaveres--;
                }
                pez.grow(resto, comido:true);
            } else {
                resto -= pez.grow(resto, comido:false);
            }
        }
    }
    if (this.muertos.size() != 0) {
        for (int i = muertos.size() - 1; i >= cadaveres; i--) {
            this.peces.remove((int) this.muertos.get(i));
        }
    }
    return resto;
}
```

public void nuevoDiaReproduccion()

Descripción: Este método simula la reproducción entre los peces en el tanque.

Uso: Se utiliza para gestionar el proceso de reproducción de los peces.

```java
/**
 * Realiza la reproducción entre peces en el tanque.
 */
public void nuevoDiaReproduccion() {
    int capacidadDisponible = this.capacidad - this.peces.size();
    List<Pez> nuevosPeces = new ArrayList<>();

    for (Pez pez : peces) {
        if (pez.isVivo() && capacidadDisponible > 0) {
            if (pez.isMaduro() && pez.reproduccion()) {
                int huevos = pez.getDatos().getHuevos();

                if (huevos <= capacidadDisponible) {
                    for (int i = 0; i < huevos; i++) {
                        Pez nuevoPez = this.crearNuevaInstancia(pez.getClass());
                        nuevosPeces.add(nuevoPez);
                        Stats.getInstancia().registrarNacimiento(nuevoPez.getDatos().getNombre());
                        capacidadDisponible--;
                    }
                } else {
                    for (int i = 0; i < capacidadDisponible; i++) {
                        Pez nPez = this.crearNuevaInstancia(pez.getClass());
                        nuevosPeces.add(nPez);
                        Stats.getInstancia().registrarNacimiento(nPez.getDatos().getNombre());
                        capacidadDisponible--;
                    }
                }
            }
        }
    }
    peces.addAll(nuevosPeces);
}
```

public void limpiarTanque()

Descripción: Este método elimina los peces muertos del tanque.

Uso: Se utiliza para limpiar el tanque y eliminar los peces muertos.

```java
/**
 * Elimina los peces muertos del tanque.
 */
public void limpiarTanque() {
    if (this.hasDead()) {
        Iterator<Integer> iterator = muertos.iterator();
        while (iterator.hasNext()) {
            int muerto = iterator.next();
            this.peces.remove((int) muerto);
            iterator.remove();
        }
    }
}
```

public void vaciarTanque()

Descripción: Este método vacía completamente el tanque, eliminando todos los peces.

Uso: Se utiliza para vaciar por completo el tanque de peces.

```java
/**
 * Vacia completamente el tanque, eliminando todos los peces.
 */
public void vaciarTanque() {
    this.peces.removeAll(peces);
}
```

public int machos()

Descripción: Este método devuelve la cantidad de peces machos vivos en el tanque.

Uso: Se utiliza para contar y obtener la cantidad de peces machos.

```java
/**
 * Devuelve la cantidad de peces machos en el tanque.
 *
 * @return La cantidad de peces machos vivos.
 */
public int machos() {
    int machos = 0;
    for (Pez pez : peces) {
        if (pez.isSexo() && pez.isVivo()) {
            machos++;
        }
    }
    return machos;
}
```

public int hembras()

Descripción: Este método devuelve la cantidad de peces hembras vivas en el tanque.

Uso: Se utiliza para contar y obtener la cantidad de peces hembras.

```java
/**
 * Devuelve la cantidad de peces hembras en el tanque.
 *
 * @return La cantidad de peces hembras vivas.
 */
public int hembras() {
    int hembras = 0;
    for (Pez pez : peces) {
        if (!pez.isSexo() && pez.isVivo()) {
            hembras++;
        }
    }
    return hembras;
}
```

public boolean sexoNuevoPez()

Descripción: Este método determina si se debe crear un nuevo pez macho o hembra en función de la proporción en el tanque.

Uso: Se utiliza para decidir el sexo del nuevo pez a crear.

```java
/**
 * Determina si se debe crear un nuevo pez macho o hembra en función de la proporción en el tanque.
 *
 * @return true si se debe crear un nuevo pez macho, false si se debe crear un nuevo pez hembra.
 */
public boolean sexoNuevoPez() {
    if (this.machos() == 0 && this.hembras() == 0) {
        return true;
    }
    if (this.machos() < this.hembras()) {
        return true;
    } else {
        return false;
    }
}
```

public Pez crearNuevaInstancia(Class<? extends Pez> tipoDePez)

Descripción: Este método crea una nueva instancia de un pez de un tipo específico.

Uso: Se utiliza para crear una nueva instancia de pez para su posterior adición al tanque.

```java
/**
 * Crea una nueva instancia de un pez.
 *
 * @param tipoDePez La clase del tipo de pez que se desea crear.
 * @return Una nueva instancia de pez del tipo especificado.
 */
public Pez crearNuevaInstancia(Class<? extends Pez> tipoDePez) {
    try {
        Constructor<? extends Pez> constructor = tipoDePez.getDeclaredConstructor(...parameterTypes:boolean.class);
        return constructor.newInstance(this.sexoNuevoPez());
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

public void comprarPez()

Descripción: Este método compra un nuevo pez y lo agrega al tanque. Se utiliza cuando ya hay peces en el tanque.

Uso: Se utiliza para adquirir y añadir un nuevo pez al tanque.

```java
/**
 * Compra un nuevo pez y lo agrega al tanque.
 * Método utilizado cuando ya hay peces en el tanque.
 */
public void comprarPez() {
    Pez npez = this.crearNuevaInstancia(this.peces.get(index:0).getClass());
    if (Monedas.getInstancia().comprobarPosible(npez.getDatos().getCoste())) {
        Monedas.getInstancia().compra(npez.getDatos().getCoste());
        this.peces.add(npez);
    } else {
        System.out.println(x:"No tienes monedas suficientes");
    }
}
```

public void comprarPez(Pez pez)

Descripción: Este método compra un pez específico y lo agrega al tanque.

Uso: Se utiliza para adquirir y añadir un pez específico al tanque.

```java
/**
 * Compra un pez específico y lo agrega al tanque.
 *
 * @param pez El pez que se desea comprar y agregar al tanque.
 */
public void comprarPez(Pez pez) {

    if (Monedas.getInstancia().comprobarPosible(pez.getDatos().getCoste())) {
        Monedas.getInstancia().compra(pez.getDatos().getCoste());
        this.peces.add(pez);
    } else {
        System.out.println(x:"No tienes monedas suficientes");
    }
}
```

public void venderOptimos()

Descripción: Este método vende los peces que tienen propiedades óptimas y registra las ganancias.

Uso: Se utiliza para vender peces óptimos y registrar las ganancias generadas.

```java
/**
 * Vende los peces que tienen propiedades óptimas y registra las ganancias.
 */
public void venderOptimos() {
    Iterator<Pez> iterator = this.peces.iterator();
    this.vendidos = 0;
    this.ganancias = 0;
    while (iterator.hasNext()) {
        Pez pez = iterator.next();
        if (pez.isOptimo() && pez.isVivo()) {
            Monedas.getInstancia().venta(pez.getDatos().getMonedas());
            Stats.getInstancia().registrarVenta(pez.getDatos().getNombre(), pez.getDatos().getMonedas());
            this.vendidos++;
            this.ganancias += pez.getDatos().getMonedas();
            iterator.remove();
        }
    }
}
```

public void venderAdultos()

Descripción: Este método vende los peces adultos en el tanque y registra las ganancias.

Uso: Se utiliza para vender peces adultos y registrar las ganancias generadas.

```java
/**
 * Vende los peces adultos en el tanque y registra las ganancias.
 */
public void venderAdultos() {
    Iterator<Pez> iterator = this.peces.iterator();
    this.vendidos = 0;
    this.ganancias = 0;
    while (iterator.hasNext()) {
        Pez pez = iterator.next();
        if (pez.isMaduro() && pez.isVivo()) {
            Monedas.getInstancia().venta(pez.getDatos().getMonedas());
            Stats.getInstancia().registrarVenta(pez.getDatos().getNombre(), pez.getDatos().getMonedas());
            this.vendidos++;
            this.ganancias += pez.getDatos().getMonedas();
            iterator.remove();
        }
    }
}
```

public int vivos()

Descripción: Este método devuelve la cantidad de peces vivos en el tanque.

Uso: Se utiliza para contar y obtener la cantidad de peces vivos.

```java
/**
 * Devuelve la cantidad de peces vivos en el tanque.
 *
 * @return La cantidad de peces vivos.
 */
public int vivos() {
    int cantidad = 0;
    for (Pez pez : peces) {
        if (pez.isVivo()) {
            cantidad++;
        }
    }
    return cantidad;
}
```

public int alimentados()

Descripción: Este método devuelve la cantidad de peces alimentados en el tanque.

Uso: Se utiliza para contar y obtener la cantidad de peces alimentados.

```java
/**
 * Devuelve la cantidad de peces alimentados en el tanque.
 *
 * @return La cantidad de peces alimentados.
 */
public int alimentados() {
    int cantidad = 0;
    for (Pez pez : peces) {
        if (pez.isAlimentado() && pez.isVivo()) {
            cantidad++;
        }
    }
    return cantidad;
}
```

public int adultos()

Descripción: Este método devuelve la cantidad de peces adultos en el tanque.

Uso: Se utiliza para contar y obtener la cantidad de peces adultos.

```java
/**
 * Devuelve la cantidad de peces adultos en el tanque.
 *
 * @return La cantidad de peces adultos.
 */
public int adultos() {
    int cantidad = 0;
    for (Pez pez : peces) {
        if (pez.isMaduro() && pez.isVivo()) {
            cantidad++;
        }
    }
    return cantidad;
}
```

Clase Piscifactoría

El propósito de la clase Piscifactoría es representar una instalación de cría de peces con múltiples tanques. Esta clase proporciona métodos y atributos para administrar los peces en la piscifactoría, alimentarlos, permitir la reproducción, vender peces y gestionar el almacenamiento de comida.

Funciones y métodos

nuevoDia()

Descripción: Realiza las acciones correspondientes para avanzar un día en la piscifactoría, incluyendo alimentar a los peces, permitir la reproducción y realizar ventas.

```java
/**
 * Realiza las acciones correspondientes para avanzar un día en la
 * piscifactoria, incluyendo alimentar a los peces, permitir la reproducción y
 * realizar ventas.
 */
public void nuevoDia() {
    for (int i = 0; i < this.tanques.size(); i++) {
        if (this.almacen != 0) {
            this.almacen -= this.tanques.get(i).nuevoDiaComer(this.almacen);
            this.tanques.get(i).nuevoDiaReproduccion();
        }
        this.tanques.get(i).venderOptimos();
        System.out.println("Piscifactoria " + this.nombre + ": " + this.tanques.get(i).getVendidos()
                + " peces vendidos por " + this.tanques.get(i).getGanancias() + " monedas");
    }
    this.gananciaDiaria();
}

/**
 * Vende todos los peces adultos en los tanques de la piscifactoria.
 */
public void venderAdultos() {
    for (Tanque<Pez> tanque : tanques) {
        tanque.venderAdultos();
    }
}
```

venderAdultos()

Descripción: Vende todos los peces adultos en los tanques de la piscifactoría.

```java
/**
 * Vende todos los peces adultos en los tanques de la piscifactoria.
 */
public void venderAdultos() {
    for (Tanque<Pez> tanque : tanques) {
        tanque.venderAdultos();
    }
}
```

gananciaDiaria()

Descripción: Calcula la ganancia diaria total de la piscifactoría y la muestra en la consola.

```java
/**
 * Calcula la ganancia diaria total de la piscifactoria y la muestra en la
 * consola.
 */
public void gananciaDiaria() {
    int cantidad = 0;
    int ganancia = 0;
    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.getVendidos();
        ganancia += tanque.getGanancias();
    }
    System.out.println("Piscifactoria " + this.nombre + ": " + cantidad + " peces vendidos por " + ganancia
            + " monedas totales");
}
```

upgradeFood()

Descripción: Realiza una actualización de la capacidad de almacenamiento de comida en función de la ubicación de la piscifactoría. Para ubicaciones en río, la capacidad se incrementa en 25 unidades. Para ubicaciones marítimas, la capacidad se incrementa en 100 unidades.

```java
/**
 * Realiza una actualización de la capacidad de almacenamiento de comida en
 * función de la ubicación de la piscifactoria.
 * Para ubicaciones en río, la capacidad se incrementa en 25 unidades.
 * Para ubicaciones maritimas, la capacidad se incrementa en 100 unidades.
 */
public void upgradeFood() {
    if (this.rio) {
        if (Monedas.getInstancia().comprobarPosible(precio:100)) {
            if (this.almacenMax < 250) {
                Monedas.getInstancia().compra(precio:100);
                this.almacenMax += 25;
            } else {
                System.out.println(x:"No puedes aumentar la capacidad");
            }
        } else {
            System.out.println(x:"No tienes dinero suficiente");
        }
    } else {
        if (Monedas.getInstancia().comprobarPosible(precio:200)) {
            if (this.almacenMax < 1000) {
                Monedas.getInstancia().compra(precio:200);
                this.almacenMax += 100;
            } else {
                System.out.println(x:"No puedes aumentar la capacidad");
            }
        } else {
            System.out.println(x:"No tienes dinero suficiente");
        }
    }
}
```

comprarTanque()

Descripción: Permite comprar un nuevo tanque de peces en la piscifactoría.

```java
/**
 * Permite comprar un nuevo tanque de peces en la piscifactoria.
 */
public void comprarTanque() {
    if (this.rio) {
        if (Monedas.getInstancia().comprobarPosible(150 * this.tanques.size())) {
            if (this.tanques.size() < 10) {
                Monedas.getInstancia().compra(150 * this.tanques.size());
                this.tanques.add(new Tanque<Pez>(capacidad:25));
            } else {
                System.out.println(x:"No es posible comprar un nuevo tanque, llegaste al máximo");
            }
        } else {
            System.out.println(x:"No tienes dinero suciciente");
        }
    } else {
        if (Monedas.getInstancia().comprobarPosible(600 * this.tanques.size())) {
            if (this.tanques.size() < 10) {
                Monedas.getInstancia().compra(600 * this.tanques.size());
                this.tanques.add(new Tanque<Pez>(capacidad:100));
            } else {
                System.out.println(x:"No es posible comprar un nuevo tanque, llegaste al máximo");
            }
        } else {
            System.out.println(x:"No tienes dinero suciciente");
        }
    }
}
```

showStatus()

Descripción: Muestra el estado actual de la piscifactoría en la consola, incluyendo información sobre la ocupación, peces vivos, peces alimentados, peces adultos, hembras/machos y el almacenamiento de comida.

```java
/**
 * Muestra el estado actual de la piscifactoria en la consola, incluyendo
 * información sobre la ocupación,
 * peces vivos, peces alimentados, peces adultos, hembras/machos y el
 * almacenamiento de comida.
 */
public void showStatus() {
    System.out.println("==========" + this.nombre + "==========");
    System.out.println("Tanques: " + this.tanques.size());
    System.out.println("Ocupacion: " + this.pecesTotales() + "/" + this.capacidadTotal() + " ("
            + this.porcentaje(this.pecesTotales(), this.capacidadTotal()) + "%)");
    System.out.println("Peces vivos: " + this.vivosTotales() + "/" + this.pecesTotales() + " ("
            + this.porcentaje(this.vivosTotales(), this.pecesTotales()) + "%)");
    System.out.println("Peces alimentados: " + this.alimentadosTotales() + "/" + this.vivosTotales() + " ("
            + this.porcentaje(this.alimentadosTotales(), this.vivosTotales()) + "%)");
    System.out.println("Peces adultos: " + this.adultosTotales() + "/" + this.vivosTotales() + " ("
            + this.porcentaje(this.adultosTotales(), this.vivosTotales()) + "%)");
    System.out.println("Hembras/Machos: " + this.hembrasTotales() + "/" + this.machosTotales());
    System.out.println("Almacen de comida actual: " + this.almacen + "/" + this.almacenMax + " ("
            + this.porcentaje(this.almacen, this.almacenMax) + "%)");
}
```

nuevoPez()

Descripción: Permite al jugador elegir un tanque para agregar un nuevo pez, ya sea comprando un pez o eligiendo uno de los peces disponibles.

```java
/**
 * Permite al jugador elegir un tanque para agregar un nuevo pez, ya sea
 * comprando un pez o eligiendo uno de los peces disponibles.
 */
public void nuevoPez() {
    Console c = System.console();
    int opcion = 0;
    int pez = 0;
    boolean salida = false;
    try {
        do {
            this.listTanks();
            opcion = Integer.parseInt(c.readLine());
            if (opcion < 0 || opcion > this.tanques.size()) {
                System.out.println(x:"Opción no válida, introduce uno de los valores mostrados");
            } else {
                if (this.tanques.get(opcion).getPeces().size() != 0) {
                    try {
                        this.tanques.get(opcion).comprarPez();
                        salida = true;
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                } else {
                    do {
                        this.opcionPez();
                        try {
                            pez = Integer.parseInt(c.readLine());
                            System.out.println(pez);
                            if (pez > 0 && pez < 8) {
                                this.añadirPez(opcion, pez);
                                salida = true;
                            } else {
                                System.out.println(x:"Opción no válida, introduce una de las opciones mostradas");
                            }
                        } catch (NumberFormatException e) {
                            System.out.println(x:"Opción no valida, introduce una de las opciones mostradas");
                        }
                    } while (pez < 1 || pez > 7);
                }
            }
        } while (!salida);
    } catch (NumberFormatException e) {
        System.out.println(x:"Opción no válida");
    }
}
```

listTanks()

Descripción: Muestra en la consola la lista de tanques disponibles en la piscifactoría.

```java
/**
 * Muestra en la consola la lista de tanques disponibles en la piscifactoria.
 */
public void listTanks() {
    for (int i = 0; i < this.tanques.size(); i++) {
        if (this.tanques.get(i).getPeces().size() == 0) {
            System.out.println(i + ". Tanque vacío");
        } else {
            System.out.println(i + ". Pez:" + this.tanques.get(i).getPeces().get(index:0).getDatos().getNombre());
        }
    }
}
```

opcionPez()

Descripción: Muestra en la consola las opciones disponibles para agregar un nuevo pez en función del tipo de piscifactoría.

```java
/**
 * Muestra en la consola las opciones disponibles para agregar un nuevo pez en
 * funcion del tipo de piscifactoría.
 */
public void opcionPez() {
    if (this.rio) {
        System.out.println(x:"******Peces******");
        System.out.println(x:"1.Pejerrey");
        System.out.println(x:"2.Lucio del norte");
        System.out.println(x:"3.Salmón chinook");
        System.out.println(x:"4.Perca europea");
        System.out.println(x:"5.Carpa");
        System.out.println(x:"6.Dorada");
        System.out.println(x:"7.Trucha arcoiris");
    } else {
        System.out.println(x:"******Peces******");
        System.out.println(x:"1.Róbalo");
        System.out.println(x:"2.Caballa");
        System.out.println(x:"3.Lenguado europeo");
        System.out.println(x:"4.Sargo");
        System.out.println(x:"5.Besugo");
        System.out.println(x:"6.Dorada");
        System.out.println(x:"7.Trucha arcoiris");
    }
}
```

añadirPez(int tanque, int pez)

Descripción: Agrega un nuevo pez al tanque especificado en función de la elección del jugador.

```java
/**
 * Agrega un nuevo pez al tanque especificado en función de la elección del
 * jugador.
 *
 * @param tanque El índice del tanque al que se agregará el nuevo pez.
 * @param pez    El índice del tipo de pez a agregar.
 */
public void añadirPez(int tanque, int pez) {
    if (this.rio) {
        if (this.tanques.get(tanque).getPeces().size() < this.tanques.get(tanque).getCapacidad()) {
            switch (pez) {
                case 1:
                    this.tanques.get(tanque).comprarPez(new Pejerrey(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 2:
                    this.tanques.get(tanque).comprarPez(new LucioDelNorte(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 3:
                    this.tanques.get(tanque).comprarPez(new SalmonChinook(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 4:
                    this.tanques.get(tanque).comprarPez(new PercaEuropea(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 5:
                    this.tanques.get(tanque).comprarPez(new Carpa(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 6:
                    this.tanques.get(tanque).comprarPez(new Dorada(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 7:
                    this.tanques.get(tanque).comprarPez(
                            new TruchaArcoiris(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                default:
                    break;
            }
        } else {
            System.out.println(x:"El tanque está lleno y no se puede agregar el pez");
        }
```

```java
    } else {
        if (this.tanques.get(tanque).getPeces().size() < this.tanques.get(tanque).getCapacidad()) {
            switch (pez) {
                case 1:
                    this.tanques.get(tanque).comprarPez(new Robalo(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 2:
                    this.tanques.get(tanque).comprarPez(new Caballa(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 3:
                    this.tanques.get(tanque)
                            .comprarPez(new LenguadoEuropeo(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 4:
                    this.tanques.get(tanque).comprarPez(new Sargo(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 5:
                    this.tanques.get(tanque).comprarPez(new Besugo(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 6:
                    this.tanques.get(tanque).comprarPez(new Dorada(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                case 7:
                    this.tanques.get(tanque)
                            .comprarPez(new TruchaArcoiris(this.tanques.get(tanque).sexoNuevoPez()));
                    break;
                default:
                    break;
            }
        } else {
            System.out.println(x:"El tanque está lleno y no se puede agregar el pez");
        }
    }
}
```

vivosTotales()

Descripción: Calcula la cantidad total de peces vivos en la piscifactoría.

```java
/**
 * Calcula la cantidad total de peces vivos en la piscifactoria.
 *
 * @return La cantidad total de peces vivos.
 */
public int vivosTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.vivos();
    }
    return cantidad;
}
```

alimentadosTotales()

Descripción: Calcula la cantidad total de peces alimentados en la piscifactoría.

```java
/**
 * Calcula la cantidad total de peces alimentados en la piscifactoria.
 *
 * @return La cantidad total de peces alimentados.
 */
public int alimentadosTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.alimentados();
    }
    return cantidad;
}
```

adultosTotales()

Descripción: Calcula la cantidad total de peces adultos en la piscifactoría.

```java
/**
 * Calcula la cantidad total de peces adultos en la piscifactoria.
 *
 * @return La cantidad total de peces adultos.
 */
public int adultosTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.adultos();
    }
    return cantidad;
}
```

machosTotales()

Descripción: Calcula la cantidad total de machos en la piscifactoría.

```java
/**
 * Calcula la cantidad total de machos en la piscifactoria.
 *
 * @return La cantidad total de machos.
 */
public int machosTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.machos();
    }
    return cantidad;
}
```

hembrasTotales()

Descripción: Calcula la cantidad total de hembras en la piscifactoría.

```java
/**
 * Calcula la cantidad total de hembras en la piscifactoria.
 *
 * @return La cantidad total de hembras.
 */
public int hembrasTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.hembras();
    }
    return cantidad;
}
```

pecesTotales()

Descripción: Calcula la cantidad total de peces en la piscifactoría.

```java
/**
 * Calcula la cantidad total de peces en la piscifactoria.
 *
 * @return La cantidad total de peces.
 */
public int pecesTotales() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.getPeces().size();
    }
    return cantidad;
}
```

capacidadTotal()

Descripción: Calcula la capacidad total de almacenamiento de peces en la piscifactoría.

```java
/**
 * Calcula la capacidad total de almacenamiento de peces en la piscifactoria.
 *
 * @return La capacidad total de almacenamiento de peces.
 */
public int capacidadTotal() {
    int cantidad = 0;

    for (Tanque<Pez> tanque : tanques) {
        cantidad += tanque.getCapacidad();
    }
    return cantidad;
}
```

limpiarTanques()

Descripción: Limpia los tanques de peces en la piscifactoría, eliminando los peces muertos.

```java
/**
 * Limpia los tanques de peces en la piscifactoria, eliminando los peces
 * muertos.
 */
public void limpiarTanques() {
    for (Tanque<Pez> tanque : tanques) {
        tanque.limpiarTanque();
    }
}
```

vaciarTanques()

Descripción: Vacía los tanques de peces en la piscifactoría.

```java
/**
 * Vacia los tanques de peces en la piscifactoria.
 */
public void vaciarTanques() {
    for (Tanque<Pez> tanque : tanques) {
        tanque.vaciarTanque();
    }
}
```

agregarComida(int cantidad)

Descripción: Permite comprar más comida para el almacén.

```java
/**
 * Método para comprar más comida para el almacén
 * @param cantidad Cantidad de comida a agregar
 */
public void agregarComida(int cantidad){
    int costo;
    if (cantidad <= 25) {
        costo = cantidad;
    } else {
        costo = cantidad - (cantidad / 25) * 5;
    }

    if (Monedas.getInstancia().comprobarPosible(costo)) {
        this.almacen += cantidad;
        Monedas.getInstancia().compra(costo);
        if (this.almacen > this.almacenMax) {
            this.almacen = this.almacenMax;
        }
        System.out.println("Añadida " + cantidad + " de comida");
    } else {
        System.out.println(x:"No tienes suficientes monedas para agregar comida.");
    }
}
```

Clase Simulador

El código se encarga de simular la gestión de piscifactorías, permitiendo al usuario realizar diversas operaciones, como administrar los tanques de peces, alimentarlos, comprar y vender peces, gestionar el almacenamiento de comida, y avanzar en el tiempo. A continuación, se analizan las funciones más relevantes del código.

Funciones y métodos

Init()

La función init se llama al inicio del programa y permite al usuario configurar el nombre de la compañía, crear una piscifactoría inicial y establecer otros elementos del juego, como la moneda y las estadísticas.

```java
/**
 * Inicializa el simulador configurando el nombre de la compañía, creando una
 * piscifactoría inicial y otros elementos del juego.
 */
public void init() {
    System.out.println(x:"Introduce el nombre de la compañía: ");
    String nombre = sc.nextLine();
    String nombreP = nombrePisc();
    Monedas.getInstancia();
    Stats.getInstancia(peces);
    this.setNombreCompa(nombre);
    this.piscifactorias.add(new Piscifactoria(rio:true, nombreP));
}
```

Menús (menu, menuPisc, showGeneralStatus, selecPisc, upgrade, tipoPisc, mar, rio)

Las funciones de menú ayudan a la navegación y la interacción del usuario con el simulador. Los menús muestran opciones y permiten al usuario realizar acciones específicas, como ver el estado general, seleccionar piscifactorías, realizar mejoras, seleccionar el tipo de piscifactoría, etc.

```java
/**
 * Muestra el menú principal del juego.
 */
public void menu() {
    System.out.println(x:"******Menú******");
    System.out.println(x:"1. Estado general");
    System.out.println(x:"2. Estado piscifactoría");
    System.out.println(x:"3. Estado tanques");
    System.out.println(x:"4. Informes");
    System.out.println(x:"5. Ictiopedia");
    System.out.println(x:"6. Pasar día");
    System.out.println(x:"7. Comprar comida");
    System.out.println(x:"8. Comprar peces");
    System.out.println(x:"9. Vender peces");
    System.out.println(x:"10. Limpiar tanques");
    System.out.println(x:"11. Vaciar tanques");
    System.out.println(x:"12. Mejorar");
    System.out.println(x:"13. Pasar varios días");
    System.out.println(x:"14. Salir");
}
```

```java
/**
 * Muestra el menú de selección de piscifactorías y permite al usuario
 * seleccionar una opción.
 *
 * @param salida El número que representa la salida a la que el usuario desea
 *               volver.
 */
public void menuPisc(int salida) {
    int contador = 1;
    System.out.println(x:"Seleccione una opción:");
    System.out.println(x:"---------------------- Piscifactorías ----------------------");
    System.out.println(x:"[Peces vivos / Peces totales / Espacio total]");
    for (Piscifactoria pisc : piscifactorias) {
        System.out.println(contador + ".-" + pisc.getNombre() + " [" + pisc.vivosTotales() + "/"
                + pisc.pecesTotales() + "/" + pisc.capacidadTotal() + "]");
    }
    try {
        int indice = Integer.parseInt(sc.nextLine());
        if (indice == 0) {

        } else if (indice < 1 || indice > this.piscifactorias.size()) {
            System.out.println(x:"Opoción inválida, retornando al menú principal");
        } else {
            if (salida == 2) {
                this.piscifactorias.get(indice - 1).showStatus();
            } else if (salida == 3) {
                this.piscifactorias.get(indice - 1).listTanks();
                int tanque = Integer.parseInt(sc.nextLine());
                if (tanque < 0 || tanque > this.piscifactorias.get(indice - 1).getTanques().size()) {
                    System.out.println(x:"Opción no válida, retornando al menú principal");
                } else {
                    Syste ArrayList<Piscifactoria> piscifactorias tanque + "==============");
                    this.piscifactorias.get(indice - 1).getTanques().get(tanque).showStatus();
                    this.piscifactorias.get(indice - 1).getTanques().get(tanque).showFishStatus();
                }
            }
        }
    } catch (NumberFormatException e) {
        System.out.println(x:"Opción no válida");

    }
}
```

```java
/**
 * Muestra el estado general de las piscifactorías, los días transcurridos y las
 * monedas disponibles. De haber un Almacén central, muestra su estado también.
 */
public void showGeneralStatus() {
    for (Piscifactoria pisc : piscifactorias) {
        pisc.showStatus();
    }
    System.out.println("Dia: " + this.dias);
    System.out.println(Monedas.getInstancia().getCantidad() + " monedas disponibles");
    if (almacenCentral == true) {
        System.out.println("Almacen central: " + AlmacenCentral.getInstance().getCapacidad() + "/"
                + AlmacenCentral.getInstance().getCapacidadMax() + " ("
                + this.piscifactorias.get(index:0).porcentaje(AlmacenCentral.getInstance().getCapacidad(),
                        AlmacenCentral.getInstance().getCapacidadMax())
                + "%)");
    }
}
```

```java
/**
 * Muestra las opciones de selección de piscifactorías disponibles.
 */
public void selecPisc() {
    for (int i = 0; i < this.piscifactorias.size(); i++) {
        System.out.println((i + 1) + ". " + this.piscifactorias.get(i).getNombre());
    }
}
```

```java
/**
 * Muestra el menú de actualización de edificios, permitiendo al usuario comprar
 * o mejorar edificios, o cancelar la operación.
 */
public void upgrade() {
    int opcion = 0;
    System.out.println(x:"******Mejoras******");
    System.out.println(x:"1. Comprar edificios");
    System.out.println(x:"2. Mejorar edificios");
    System.out.println(x:"3. Cancelar");
    try {
        opcion = Integer.parseInt(sc.nextLine());
    } catch (NumberFormatException e) {
        System.out.println(x:"Argumento inválido, retrocediendo al menú principal");
    }
    switch (opcion) {
        case 1:
            System.out.println(x:"******Comprar edificios******");
            System.out.println(x:"1. Piscifactoría");
            if (almacenCentral == false) {
                System.out.println(x:"2. Almacén central");
            }
            opcion = Integer.parseInt(sc.nextLine());
            switch (opcion) {
                case 1:
                    boolean tipo = tipoPisc();
                    nuevaPisc(tipo);
                    break;
                case 2:
                    if (almacenCentral == false) {
                        comprarAlmacen();
                    } else {
                        System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                    }
                    break;
                default:
                    System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                    break;
            }
            break;
```

```
                break;
            case 2:
                if (almacenCentral == false) {
                    comprarAlmacen();
                } else {
                    System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                }
                break;
            default:
                System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                break;
        }
        break;
    case 2:
        System.out.println(x:"******Mejorar edificios******");
        System.out.println(x:"1. Piscifactoría");
        if (almacenCentral == true) {
            System.out.println(x:"2. Almacen central");
        }
        opcion = Integer.parseInt(sc.nextLine());
        switch (opcion) {
            case 1:
                int pisc = 0;
                do {
                    this.selecPisc();
                    try {
                        pisc = Integer.parseInt(sc.nextLine());
                        if (pisc < 1 || pisc > this.piscifactorias.size()) {
                            System.out.println(x:"Índice incorrecto, inserta un valor de los indicados");
                        } else {
                            do {
                                System.out.println(x:"******Mejorar piscifactoría******");
                                System.out.println(x:"1. Comprar tanque");
                                System.out.println(x:"2. Aumentar almacén");
                                System.out.println(x:"3. Cancelar");
                                opcion = Integer.parseInt(sc.nextLine());
                                switch (opcion) {
                                    case 1:
                                        this.piscifactorias.get(pisc - 1).comprarTanque();
                                        break;
                                    case 2:
                                        this.piscifactorias.get(pisc - 1).upgradeFood();
                                        break;
                                    case 3:
                                        opcion = 2;
                                        break;
                                    default:
                                        System.out
                                                .println(x:"Opción no válida, inserta un valor de los indicados");
                                        break;
                                }
                            } while (opcion != 1 || opcion != 2 || opcion != 3);
                        }
                    } catch (NumberFormatException e) {
                        System.out.println(x:"Opción no válida, inserta un valor de los indicados");
```

```
                    } while (pisc < 1 || pisc > this.piscifactorias.size());
                    break;
                case 2:
                    if (almacenCentral == true) {
                        AlmacenCentral.getInstance().upgrade();
                    } else {
                        System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                    }
                    break;
                default:
                    System.out.println(x:"Opción no válida, retrocediendo al menú principal");
                    break;
            }
            break;
        case 3:
            break;
        default:
            System.out.println(x:"Opción no válida, retrocediendo al menú principal");
            break;
    }
}
```

```java
/**
 * Permite al usuario seleccionar el tipo de piscifactoría (río o mar) y
 * devuelve un valor booleano correspondiente.
 *
 * @return true si se selecciona río, false si se selecciona mar.
 */
public boolean tipoPisc() {
    int salida = 0;
    boolean cosa = true;
    do {
        System.out.println(x:"Selecciona el tipo de piscifactoría");
        System.out.println(x:"1.Río");
        System.out.println(x:"2.Mar");
        try {
            salida = Integer.parseInt(sc.nextLine());
        } catch (NumberFormatException e) {
            System.out.println(x:"Argumento inválido");
        }
        if (salida == 1) {
            cosa = true;
        } else if (salida == 2) {
            cosa = false;
        } else {
            System.out.println(x:"Opción no válida, vuelve a introducir tu elección");
        }
    } while (salida < 1 || salida > 2);
    return cosa;
}
```

```java
/**
 * Calcula el número de piscifactorías de tipo "mar".
 *
 * @return Número de piscifactorías de tipo "mar".
 */
public int mar() {
    int numero = 0;
    for (Piscifactoria piscifactoria : piscifactorias) {
        if (!piscifactoria.isRio()) {
            numero++;
        }
    }
    return numero;
}
```

```java
/**
 * Calcula el número de piscifactorías de tipo "río".
 *
 * @return Número de piscifactorías de tipo "río".
 */
public int rio() {
    int numero = 0;
    for (Piscifactoria piscifactoria : piscifactorias) {
        if (piscifactoria.isRio()) {
            numero++;
        }
    }
    return numero;
}
```

nuevaPisc()

La función nuevaPisc permite al usuario crear una nueva piscifactoría, sea de tipo río o mar. El costo de la nueva piscifactoría se calcula en función del tipo de piscifactoría y la cantidad existente de cada tipo. También, verifica si el usuario tiene suficientes monedas para realizar la compra.

```java
/**
 * Crea una nueva piscifactoría, ya sea de tipo río o mar, dependiendo del valor
 * de la variable "rio".
 * Si no hay piscifactorías de tipo mar, se asume un costo de 500 monedas.
 *
 * @param rio true si la piscifactoría es de tipo río, false si es de tipo mar.
 */
public void nuevaPisc(boolean rio) {
    if (rio) {
        if (Monedas.getInstancia().comprobarPosible(this.rio() * 500)) {
            Monedas.getInstancia().compra(this.rio() * 500);
            String nompreP = nombrePisc();
            this.piscifactorias.add(new Piscifactoria(rio, nompreP));
        } else {
            System.out.println(x:"No tienes suficientes monedas");
        }
    } else {
        if (this.mar() == 0) {
            if (Monedas.getInstancia().comprobarPosible(1 * 500)) {
                Monedas.getInstancia().compra(this.mar() * 500);
                String nompreP = nombrePisc();
                this.piscifactorias.add(new Piscifactoria(rio, nompreP));
            } else {
                System.out.println(x:"No tienes suficientes monedas");
            }
        } else {
            if (Monedas.getInstancia().comprobarPosible(this.mar() * 500)) {
                Monedas.getInstancia().compra(this.mar() * 500);
                String nompreP = nombrePisc();
                this.piscifactorias.add(new Piscifactoria(rio, nompreP));
            } else {
                System.out.println(x:"No tienes suficientes monedas");
            }
        }
    }
}
```

nombrePisc()

La función nombrePisc solicita al usuario que ingrese el nombre de una piscifactoría y devuelve ese nombre como una cadena de texto.

```java
/**
 * Solicita al usuario introducir el nombre de una piscifactoría y devuelve la
 * entrada del usuario como una cadena de texto.
 *
 * @return El nombre de la piscifactoría ingresado por el usuario.
 */
public String nombrePisc() {
    System.out.println(x:"Introduce el nombre de la piscifactoría");
    return sc.nextLine();
}
```

comprarAlmacen()

La función comprarAlmacen permite al usuario comprar un almacén central si aún no ha sido adquirido. Comprueba si hay suficientes monedas y, si es posible, realiza la compra. Una vez comprado, activa el almacén central.

```java
/**
 * Permite al usuario comprar un almacén central si aún no ha sido adquirido.
 * Comprueba si hay suficientes monedas y realiza la compra si es posible.
 * Si la compra se realiza con éxito, se activa el almacén central y se
 * establece el indicador "almacenCentral" en true.
 */
public void comprarAlmacen() {
    if (Monedas.getInstancia().comprobarPosible(precio:2000)) {
        Monedas.getInstancia().compra(precio:2000);
        AlmacenCentral.getInstance();
        this.almacenCentral = true;
    } else {
        System.out.println(x:"No tienes suficientes monedas");
    }
}
```

showIctio()

La función showIctio muestra información detallada sobre las especies de peces disponibles. El usuario puede seleccionar una especie para obtener más detalles.

```java
/**
 * Muestra información detallada sobre las especies de peces disponibles,
 * permitiendo al usuario seleccionar una para obtener más detalles.
 */
public void showIctio() {
    int opcion = 0;
    do {
        for (int i = 0; i < peces.length; i++) {
            System.out.println((i + 1) + ". " + peces[i]);
        }
        try {
            opcion = Integer.parseInt(sc.nextLine());
            if (opcion < 1 || opcion > peces.length) {
                System.out.println(x:"Opción no válida");
            } else {
                switch (opcion) {
                    case 1:
                        Besugo.datos();
                        break;
                    case 2:
                        Pejerrey.datos();
                        break;
                    case 3:
                        Carpa.datos();
                        break;
                    case 4:
                        SalmonChinook.datos();
                        break;
                    case 5:
                        LucioDelNorte.datos();
                        break;
                    case 6:
                        PercaEuropea.datos();
                        break;
                    case 7:
                        Robalo.datos();
                        break;
                    case 8:
                        Caballa.datos();
                        break;
                    case 9:
                        LenguadoEuropeo.datos();
                        break;
                    case 10:
                        Sargo.datos();
                        break;
                    case 11:
                        Dorada.datos();
                        break;
                    case 12:
                        TruchaArcoiris.datos();
                        break;
                    default:
                        break;
                }
            }
        } catch (NumberFormatException e) {
            System.out.println(x:"Opción no válida");
        }
    } while (opcion < 1 || opcion > peces.length);
}
```

venderPeces()

La función venderPeces vende peces adultos en todas las piscifactorías y muestra estadísticas relacionadas con la venta de peces.

```java
/**
 * Vende peces adultos en todas las piscifactorías.
 * Luego, muestra las estadísticas relacionadas con la venta de peces.
 */
public void venderPeces() {
    for (Piscifactoria pisc : piscifactorias) {
        pisc.venderAdultos();
    }
    Stats.getInstancia().mostrar();
}
```

añadirPez()

La función añadirPez permite al usuario seleccionar una piscifactoría y agregar un pez a un tanque en esa piscifactoría.

```java
/**
 * Permite al usuario seleccionar una piscifactoría y añadir un pez a un tanque
 * en esa piscifactoría.
 */
public void añadirPez() {
    int pisc = 0;
    boolean salida = false;
    do {
        this.selecPisc();
        try {
            pisc = Integer.parseInt(sc.nextLine());
            if (pisc < 0 || pisc > this.piscifactorias.size()) {
                System.out.println(x:"Índice incorrecto, inserta un valor de los indicados");
            } else {
                this.piscifactorias.get(pisc - 1).nuevoPez();
                salida = true;
            }
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            System.out.println(x:"Argumento inválido, retrocediendo al menú principal");
            salida = true;
        }
    } while (!salida);
}
```

nuevoDia()

La función nuevoDia permite al usuario avanzar un número específico de días en la simulación, realizando todas las acciones necesarias en los peces (comida, crecimiento y reproducción).

```java
/**
 * Permite al usuario avanzar un número específico de días en la simulación.
 *
 * @param dias La cantidad de días que se avanzarán en la simulación.
 */
public void nuevoDia(int dias) {
    for (int i = 0; i < dias; i++) {
        for (Piscifactoria pisc : piscifactorias) {
            pisc.nuevoDia();
        }
        this.dias++;
    }
}
```

addFood()

La función addFood permite al usuario comprar comida para una piscifactoría específica. Ofrece varias opciones de cantidad de comida para comprar y llenar el almacén de comida de esa piscifactoría.

```java
/***
 * Método para comprar comida
 *
 */
public void addFood() {
    this.selecPisc();
    int pisc = Integer.parseInt(sc.nextLine());

    System.out.println(x:"Opciones de comida:");
    System.out.println(x:"1. Añadir 5");
    System.out.println(x:"2. Añadir 10");
    System.out.println(x:"3. Añadir 25");
    System.out.println(x:"4. Llenar");
    System.out.println(x:"5. Salir");
    System.out.print(s:"Elige una opción: ");

    int opcion = Integer.parseInt(sc.nextLine());

    switch (opcion) {
        case 1:
            this.piscifactorias.get(pisc-1).agregarComida(cantidad:5);
            break;
        case 2:
            this.piscifactorias.get(pisc-1).agregarComida(cantidad:10);
            break;
        case 3:
            this.piscifactorias.get(pisc-1).agregarComida(cantidad:25);
            break;
        case 4:
            this.piscifactorias.get(pisc-1).agregarComida(this.piscifactorias.get(pisc-1).getAlmacenMax() - this.piscifactorias.get(pisc-1).getAlmacen());
            break;
        case 5:
            break;
        default:
            System.out.println(x:"Opción no válida.");
    }
}
```

Upgrade()

La función upgrade permite al usuario realizar mejoras en las piscifactorías y el almacén central. Permite comprar nuevos tanques, aumentar la capacidad de almacenamiento de comida y mejorar el almacén central.

El código de esta función ya fue proporcionado en el apartado de Menús.

Clase EscritorHelper

Esta clase se utiliza para la salida a los archivos de log, save y trans.

Funciones y métodos

public void addFirstLines(String partida, String pisci1)

Este método se utiliza para agregar las líneas iniciales al log.

```java
/**
 * Método que agrega las líneas iniciales al archivo de transcripción y
 * log.
 *
 * @param partida Nombre de la partida.
 * @param pisci1  Piscifactoría inicial.
 */
public void addFirstLines(String partida, String pisci1) {
    String primerasLineas = "=====Arranque=====" + "\n" +
            "Iniciando partida " + partida + "." + "\n" +
            "=====Dinero Inicial=====" + "\n" +
            "Dinero: 100 monedas" + "\n" +
            "=====Peces Implementados=====" + "\n" +
            "Río: " + "\n" +
            "Pejerrey" + "\n" +
            "Carpa plateada" + "\n" +
            "Salmón chinook" + "\n" +
            "Tilapia del Nilo" + "\n" +
            "Carpa" + "\n" +
            "Mar:" + "\n" +
            "Róbalo" + "\n" +
            "Caballa" + "\n" +
            "Arenque del Atlántico" + "\n" +
            "Sargo" + "\n" +
            "Besugo" + "\n" +
            "Doble:" + "\n" +
            "Dorada" + "\n" +
            "Trucha Arcoiris" + "\n" +
            "-------------------------" + "\n" +
            "Piscifactoria inicial: " + pisci1 + "\n";
    addTrans(primerasLineas);

    String primeraLog = "Inicio de la simulación " + partida + ".\n";
    String segundaLog = "Piscifactoría inicial: " + pisci1 + ".\n";
    addLogs(primeraLog);
    addLogs(segundaLog);
}
```

public void cerrarStreams()

Método para cerrar los streams abiertos con los archivos.

```java
/**
 * Cierra los streams de log y transcripción.
 */
public void cerrarStreams() {
    try {
        if (streamTransicripcion != null) {
            streamTransicripcion.close();
        }
        if (streamLog != null) {
            streamLog.close();
        }
    } catch (IOException e) {
        escritorHelper.addError(mensaje:"Error al cerrar los streams de log y transcipcion\n");
    }
}
```

public void addTrans(String mensaje)

Método para agregar una transcripción.

```java
/**
 * Agrega una línea al archivo de transcripción.
 *
 * @param mensaje Mensaje a agregar.
 */
public void addTrans(String mensaje) {
    try {
        streamTransicripcion.write(mensaje);
        streamTransicripcion.flush();
    } catch (IOException e) {
        escritorHelper.addError(mensaje:"Error al añadir una linea al archivo de transcripcion \n");
    }
}
```

private String obtenerHora()

Método para la obtención de la hora actual.

```java
/**
 * Obtiene la hora actual formateada.
 *
 * @return Hora actual formateada.
 */
private String obtenerHora() {
    // Obtener la fecha y hora actual
    Date fechaHoraActual = new Date();

    // Definir el formato de fecha y hora deseado
    SimpleDateFormat formato = new SimpleDateFormat(pattern:"[yyyy-MM-dd HH:mm:ss]");

    // Formatear la fecha y hora actual
    return formato.format(fechaHoraActual);
}
```

public void addLogs(String mensaje)

Método para agregar una nueva línea en el archivo de log.

```java
/**
 * Agrega una línea al archivo de log con la hora actual.
 *
 * @param mensaje Mensaje a agregar.
 */
public void addLogs(String mensaje) {
    try {
        streamLog.write(obtenerHora() + " " + mensaje);
        streamLog.flush();
    } catch (IOException e) {
        escritorHelper.addError(mensaje:"Error al añadir una linea al archivo de log\n");
    }
}
```

public void addError(String mensaje)

Método para agregar una nueva línea en el archivo de errores.

```java
/**
 * Agrega una línea al archivo de errores.
 *
 * @param mensaje Mensaje de error a agregar.
 */
public void addError(String mensaje) {
    BufferedWriter streamError = null;
    try {
        File error = new File(pathname:"Logs/0_errors.log");
        streamError = new BufferedWriter(new FileWriter(error));
        streamError.write(obtenerHora() + " " + mensaje);
        streamError.flush();
    } catch (IOException e) {
        escritorHelper.addError(mensaje:"Error al añadir una linea al archivo de errores\n");
    } finally {
        try {
            if (streamError != null) {
                streamError.close();
            }
        } catch (IOException e) {
            escritorHelper.addError(mensaje:"Error al cerrar el stream de errores\n");
        }
    }
}
```