TRABAJO FINAL PROGRAMACION ORIENTADA A OBJETOS

JUAN DAVID LONDOÑO

MATEO ZUÑIGA

CARLOS LOITTI

JOSE ALEJANDRO PARRA

DANNY CASTRILLON

JOSEPH PARRA

FCECEP

PROGRAMACION ORIENTADA A OBJETOS

CALI

2023

TRABAJO FINAL

JUAN DAVID LONDOÑO

MATEO ZUÑIGA

CARLOS LOITTI

JOSE ALEJANDRO PARRA

DANNY CASTRILLON

JOSEPH PARRA

TRABAJO EXPOSITIVO

GERMAN ANDRES MORALES LEON

FCECEP

PROGRAMACION ORIENTADA A OBJETOS

CALI

2023

TRABAJO FINAL

[**MODIFICADORES DE ACCESO** 4](#_Toc136193599)

[**PANELES Y FECHAS** 4](#_Toc136193600)

[**CONTROLADORES** 7](#_Toc136193601)

[**TIPOS DE FORMULARIOS** 9](#_Toc136193602)

[**HERENCIAS** 12](#_Toc136193603)

[**EVENTOS EN JAVA** 14](#_Toc136193604)

[**EVENTOS EN POO** 17](#_Toc136193605)

[**IDENTIFICADORES CON SUS TIPOS Y CARACTERÍSTICAS EN JAVA** 18](#_Toc136193606)

[**IDENTIFICADORES CON SUS TIPOS Y CARACTERÍSTICAS EN POO** 19](#_Toc136193607)

[**VARIABLES CON SUS TIPOS Y CARACTERÍSTICAS:** 20](#_Toc136193608)

[**ROLES EN JAVA:** 22](#_Toc136193609)

[**HERENCIAS EN JAVA:** 23](#_Toc136193610)

[**ATRIBUTOS** 28](#_Toc136193611)

[**VARIABLES DE INSTANCIA** 29](#_Toc136193612)

[**VARIABLES DE CLASE** 29](#_Toc136193613)

[**CONTROLES SWING** 30](#_Toc136193614)

[**CUADROS DE DIALOGO** 34](#_Toc136193615)

[**JASPER REPORT** 35](#_Toc136193616)

# **MODIFICADORES DE ACCESO**

Los modificadores de acceso son utilizados en la programación orientada a objetos para controlar el nivel de acceso de las clases, métodos y variables dentro de un programa. Estos modificadores definen quién puede acceder a los elementos y desde dónde se puede acceder. Los modificadores de acceso más comunes son los siguientes:

1. Public: Indica que la clase, método o variable es accesible desde cualquier parte del programa, ya sea dentro de la misma clase, dentro de otras clases del mismo paquete o en clases de paquetes diferentes.
2. Private: Indica que la clase, método o variable solo es accesible dentro de la misma clase. No se puede acceder a él desde otras clases, incluso si están en el mismo paquete.
3. Protected: Indica que la clase, método o variable es accesible dentro de la misma clase, dentro de otras clases del mismo paquete y también en subclases, ya sean del mismo paquete o de paquetes diferentes.
4. Default (sin especificar un modificador): Si no se especifica ningún modificador, se utiliza el modificador por defecto. La clase, método o variable con modificador por defecto es accesible dentro de la misma clase y dentro de otras clases del mismo paquete, pero no es accesible desde clases de paquetes diferentes.

Estos modificadores de acceso permiten controlar la visibilidad y el nivel de encapsulamiento en la programación orientada a objetos. Al definir los modificadores de acceso adecuados, puedes establecer qué elementos de una clase son accesibles desde otras partes del programa y cuáles están ocultos y solo pueden ser utilizados internamente por la propia clase. Esto ayuda a mantener la seguridad, la integridad y la estructura de tu código.

# **PANELES Y FECHAS**

En la programación orientada a objetos, los paneles y las fechas son elementos que se pueden representar como objetos y manipular utilizando conceptos y técnicas de la POO. Aquí tienes una descripción de cómo se pueden utilizar en este contexto:

1. Paneles: Un panel es un componente de interfaz gráfica de usuario (GUI) que se utiliza para organizar y contener otros componentes, como botones, etiquetas, campos de texto, etc. En la programación orientada a objetos, un panel se puede representar como un objeto de una clase específica, como por ejemplo, la clase JPanel en Java.

Los paneles se utilizan para crear interfaces de usuario más complejas al agrupar y organizar los componentes visuales de manera lógica. Se pueden agregar, eliminar o modificar componentes dentro del panel, y se pueden aplicar propiedades y comportamientos específicos a través de métodos y atributos definidos en la clase del panel.

En Java, los paneles son componentes de la interfaz gráfica de usuario (GUI) que se utilizan para organizar y contener otros componentes. Para trabajar con paneles en Java, generalmente se utiliza la clase JPanel, que forma parte del paquete javax.swing.

Ejemplo:

import javax.swing.JFrame;

import javax.swing.JPanel;

public class EjemploPanel {

public static void main(String[] args) {

JFrame frame = new JFrame("Ejemplo Panel");

JPanel panel = new JPanel();

// Agregar componentes al panel

// panel.add(componente1);

// panel.add(componente2);

frame.getContentPane().add(panel);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.pack();

frame.setVisible(true);

}

}

1. Fechas: En programación, las fechas se representan como objetos para poder manipularlas y realizar operaciones con ellas de manera más conveniente. En la mayoría de los lenguajes de programación orientados a objetos, existen clases predefinidas para trabajar con fechas, como por ejemplo, la clase Date en Java o la clase DateTime en C#.

Estas clases proporcionan métodos para crear, comparar, formatear y realizar cálculos con fechas. También pueden incluir funcionalidades como el manejo de zonas horarias, el cálculo de diferencia entre fechas, la conversión de formatos, entre otros.

Al utilizar objetos de fecha, puedes realizar operaciones como sumar o restar días, meses o años, obtener el día de la semana, comparar fechas, formatearlas según un patrón específico, entre otras acciones relacionadas con el manejo y procesamiento de fechas en tu programa.

Ejemplo:

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class EjemploFecha {

public static void main(String[] args) {

LocalDate fechaActual = LocalDate.now();

System.out.println("Fecha actual: " + fechaActual);

LocalDate fechaPersonalizada = LocalDate.of(2023, 5, 28);

System.out.println("Fecha personalizada: " + fechaPersonalizada);

DateTimeFormatter formateador = DateTimeFormatter.ofPattern("dd/MM/yyyy");

String fechaFormateada = fechaActual.format(formateador);

System.out.println("Fecha formateada: " + fechaFormateada);

// Operaciones con fechas

LocalDate fechaFutura = fechaActual.plusDays(7);

System.out.println("Fecha futura: " + fechaFutura);

// Comparación de fechas

boolean esAnterior = fechaPersonalizada.isBefore(fechaActual);

System.out.println("¿La fecha personalizada es anterior a la fecha actual? " + esAnterior);

}

}

En resumen, en la programación orientada a objetos, los paneles y las fechas son elementos que se pueden representar y manipular como objetos de clases específicas. Los paneles se utilizan para organizar componentes de interfaz gráfica, mientras que las fechas se utilizan para trabajar con valores de fecha y realizar operaciones relacionadas con ellas.

# **CONTROLADORES**

En la programación orientada a objetos (POO), los controladores son componentes que se utilizan para gestionar y controlar el flujo de ejecución de un programa. Estos controladores son objetos que se encargan de recibir y procesar las interacciones del usuario, así como de coordinar las operaciones entre los diferentes objetos y componentes del sistema.

Los controladores en la POO pueden tomar diferentes formas y nombres dependiendo del contexto y del diseño de la aplicación. Algunos nombres comunes para los controladores son: controladores de eventos, controladores de acción, controladores de flujo, controladores de formulario, entre otros. En esencia, su función principal es recibir las acciones del usuario, procesarlas y tomar las decisiones adecuadas para manejar esas acciones.

Un controlador típicamente se asocia con una interfaz de usuario y puede tener métodos que responden a eventos específicos, como hacer clic en un botón, seleccionar una opción de un menú, introducir texto en un campo, etc. Estos métodos pueden contener la lógica necesaria para realizar las acciones correspondientes, como actualizar datos, mostrar información, invocar métodos de otros objetos, etc.

Ejemplo:

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class MiControlador implements ActionListener {

private MiModelo modelo;

private MiVista vista;

public MiControlador(MiModelo modelo, MiVista vista) {

this.modelo = modelo;

this.vista = vista;

this.vista.agregarListener(this);

}

@Override

public void actionPerformed(ActionEvent e) {

if (e.getSource() == vista.getBoton()) {

// Realizar acciones cuando se hace clic en el botón

String texto = vista.getTexto();

modelo.procesarTexto(texto);

vista.mostrarResultado(modelo.getResultado());

}

}

}

En este ejemplo, el controlador **MiControlador** implementa la interfaz **ActionListener** que permite manejar eventos de acción en la interfaz de usuario. El controlador se asocia con una vista (**MiVista**) y un modelo (**MiModelo**) mediante su constructor.

En Java, los controladores son componentes utilizados para gestionar la lógica de negocio y controlar el flujo de una aplicación. Estos controladores se encargan de recibir las interacciones del usuario, realizar las operaciones correspondientes y coordinar la comunicación entre los diferentes componentes de la aplicación, como modelos y vistas.

En el contexto de desarrollo de aplicaciones web en Java, los controladores suelen ser implementados utilizando el patrón de diseño Modelo-Vista-Controlador (MVC), donde el controlador actúa como intermediario entre la vista y el modelo. En este enfoque, el controlador maneja las solicitudes del usuario, interactúa con el modelo para obtener o actualizar datos y coordina la presentación de la información a través de la vista.

Ejemplo:

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

@Controller

public class MiControlador {

@GetMapping("/saludo")

public String saludar(@RequestParam(name = "nombre", required = false) String nombre, Model model) {

if (nombre != null) {

String mensaje = "¡Hola, " + nombre + "!";

model.addAttribute("mensaje", mensaje);

}

return "saludo";

}

}

En este ejemplo, la clase **MiControlador** está anotada con **@Controller**, lo que indica que es un controlador en el contexto de Spring MVC. El método **saludar** se anota con **@GetMapping("/saludo")**, lo que significa que responde a las solicitudes GET en la ruta "/saludo".

# **TIPOS DE FORMULARIOS**

En la programación orientada a objetos (POO), los formularios son componentes utilizados para recopilar y enviar datos ingresados por el usuario. Los formularios permiten solicitar información y recibir respuestas a través de campos de entrada, botones y otros elementos interactivos.

En el contexto de la programación orientada a objetos, se puede abordar la implementación de formularios utilizando clases y objetos.

Ejemplo:

public class Formulario {

private String nombre;

private int edad;

private String email;

public void setNombre(String nombre) {

this.nombre = nombre;

}

public void setEdad(int edad) {

this.edad = edad;

}

public void setEmail(String email) {

this.email = email;

}

public void enviarFormulario() {

// Lógica para enviar el formulario a un servidor o procesarlo localmente

// ...

}

}

En este ejemplo, se crea una clase **Formulario** que representa el formulario en sí. La clase tiene atributos para almacenar los datos ingresados por el usuario, como nombre, edad y email. También tiene métodos para establecer estos valores a través de los correspondientes setters.

En Java, los formularios se pueden implementar utilizando diferentes tecnologías y frameworks, como JavaFX, Swing, Servlets o JavaServer Faces (JSF).

Ejemplo;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

public class FormularioJavaFX extends Application {

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Formulario");

Label lblNombre = new Label("Nombre:");

TextField txtNombre = new TextField();

Label lblEdad = new Label("Edad:");

TextField txtEdad = new TextField();

Button btnEnviar = new Button("Enviar");

btnEnviar.setOnAction(event -> {

String nombre = txtNombre.getText();

int edad = Integer.parseInt(txtEdad.getText());

// Lógica para procesar los datos ingresados en el formulario

System.out.println("Nombre: " + nombre);

System.out.println("Edad: " + edad);

});

VBox vbox = new VBox(10);

vbox.getChildren().addAll(lblNombre, txtNombre, lblEdad, txtEdad, btnEnviar);

Scene scene = new Scene(vbox, 300, 200);

primaryStage.setScene(scene);

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}

# **HERENCIAS**

La herencia es un concepto fundamental en la programación orientada a objetos (POO) que permite la creación de jerarquías de clases y la reutilización de código. En la herencia, una clase puede heredar atributos y métodos de otra clase, lo que permite establecer relaciones de "es-un" entre las clases.

En la herencia, la clase que hereda se conoce como clase derivada o subclase, y la clase de la cual se heredan los atributos y métodos se conoce como clase base o superclase. La subclase hereda todos los miembros no privados (atributos y métodos) de la superclase y puede agregar nuevos miembros o modificar el comportamiento de los miembros heredados.

La herencia se denota en Java utilizando la palabra clave **extends**.

Ejemplo:

public class Animal {

private String nombre;

public Animal(String nombre) {

this.nombre = nombre;

}

public void comer() {

System.out.println("El animal come.");

}

public void dormir() {

System.out.println("El animal duerme.");

}

}

public class Perro extends Animal {

public Perro(String nombre) {

super(nombre);

}

public void ladrar() {

System.out.println("El perro ladra.");

}

}

public class HerenciaEjemplo {

public static void main(String[] args) {

Perro perro = new Perro("Max");

perro.comer();

perro.dormir();

perro.ladrar();

}

}

La herencia en Java es un mecanismo que permite la creación de jerarquías de clases, donde una clase puede heredar atributos y métodos de otra clase. Esto fomenta la reutilización de código y permite establecer relaciones de "es-un" entre las clases.

En Java, la herencia se implementa utilizando la palabra clave **extends**.

Ejemplo:}

class ClaseBase {

// Atributos y métodos de la clase base

}

class Subclase extends ClaseBase {

// Atributos y métodos adicionales de la subclase

}

En el ejemplo anterior, la clase **Subclase** hereda de la clase **ClaseBase**. La subclase **Subclase** adquiere todos los atributos y métodos de la clase base **ClaseBase** y puede agregar nuevos atributos y métodos propios.

# **EVENTOS EN JAVA**

En Java, los eventos son acciones que ocurren durante la ejecución de un programa y que pueden ser detectadas y manejadas por el código. Los eventos pueden ser generados por el usuario, como hacer clic en un botón, o por el sistema, como recibir datos de una red.

Para trabajar con eventos en Java, se utilizan clases y interfaces proporcionadas por la biblioteca estándar, como **ActionEvent**, **MouseListener** o **ActionListener**. A continuación, te mostraré algunos ejemplos de cómo trabajar con eventos en Java:

1. Eventos de botón:

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

public class EventosBoton extends JFrame implements ActionListener {

private JButton button;

public EventosBoton() {

button = new JButton("Haz clic");

button.addActionListener(this); // Registra este objeto como el oyente del evento

add(button);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

pack();

setVisible(true);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == button) {

System.out.println("¡Se hizo clic en el botón!");

}

}

public static void main(String[] args) {

new EventosBoton();

}

}

1. Eventos de ratón:

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class EventosRaton extends JFrame implements MouseListener {

private JPanel panel;

public EventosRaton() {

panel = new JPanel();

panel.addMouseListener(this); // Registra este objeto como el oyente del evento

add(panel);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

pack();

setVisible(true);

}

@Override

public void mouseClicked(MouseEvent e) {

System.out.println("Se hizo clic en (" + e.getX() + ", " + e.getY() + ")");

}

public static void main(String[] args) {

new EventosRaton();

}

}

Estos son solo ejemplos básicos para ilustrar el uso de eventos en Java. Dependiendo del tipo de aplicación que estés desarrollando, los eventos y sus manejadores pueden variar. Puedes explorar más eventos y cómo manejarlos consultando la documentación oficial de Java.

# **EVENTOS EN POO**

En la programación orientada a objetos (POO), los eventos se manejan utilizando el concepto de "observadores" o "listeners". Un observador es un objeto que está pendiente de un evento específico y actúa en consecuencia cuando se produce dicho evento. A continuación, te mostraré cómo se pueden implementar eventos en la POO utilizando un ejemplo en Java:

Supongamos que tienes una clase llamada Button (botón) y quieres permitir que otros objetos se suscriban a eventos de clic en ese botón. Aquí tienes un ejemplo de cómo podrías implementarlo:

import java.util.ArrayList;

import java.util.List;

public class Button {

private List<ButtonClickListener> listeners = new ArrayList<>();

public void addClickListener(ButtonClickListener listener) {

listeners.add(listener);

}

public void removeClickListener(ButtonClickListener listener) {

listeners.remove(listener);

}

public void click() {

// Realizar acciones de clic en el botón

// Notificar a los listeners del evento de clic

for (ButtonClickListener listener : listeners) {

listener.onClick();

}

}

}

En este ejemplo, la clase Button tiene una lista de objetos ButtonClickListener, que son los observadores interesados en los eventos de clic en el botón. La clase Button proporciona métodos para agregar (addClickListener) y eliminar (removeClickListener) oyentes, así como un método click para simular el evento de clic en el botón.

# **IDENTIFICADORES CON SUS TIPOS Y CARACTERÍSTICAS EN JAVA**

En Java, los identificadores son utilizados para dar nombres a las variables, métodos, clases y otros elementos del código. Los identificadores deben seguir ciertas reglas y convenciones establecidas. A continuación, se presentan los tipos de identificadores más comunes en Java y sus características:

1. Identificadores de variables y parámetros:

Reglas: Los identificadores de variables deben comenzar con una letra minúscula o un guion bajo ("\_"). Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: nombre, \_edad, cantidadProductos

Características: Los identificadores de variables son utilizados para almacenar valores en memoria durante la ejecución del programa. Deben ser descriptivos y seguir las convenciones de nomenclatura camelCase (es decir, las palabras se escriben sin espacios y cada palabra después de la primera comienza con una letra mayúscula).

1. Identificadores de clases e interfaces:

Reglas: Los identificadores de clases e interfaces deben comenzar con una letra mayúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: Persona, MiInterfaz, Empleado

Características: Los identificadores de clases e interfaces se utilizan para definir tipos de objetos y especificar comportamientos y propiedades comunes para los objetos de ese tipo.

1. Identificadores de métodos:

Reglas: Los identificadores de métodos deben comenzar con una letra minúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: calcularPrecio, obtenerNombre, imprimirSaludo

Características: Los identificadores de métodos se utilizan para definir bloques de código que realizan una tarea específica. Los métodos pueden tener parámetros y devolver valores, y se llaman invocándolos desde otras partes del código.

1. Identificadores de constantes:

Reglas: Los identificadores de constantes suelen escribirse completamente en mayúsculas. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: PI, MAXIMO\_VALOR, URL\_BASE

Características: Los identificadores de constantes se utilizan para definir valores que no cambian durante la ejecución del programa. Por convención, se suelen utilizar letras mayúsculas para mejorar su legibilidad y distinguirlos de las variables.

Es importante tener en cuenta que los identificadores en Java son sensibles a mayúsculas y minúsculas, lo que significa que nombre y Nombre se consideran identificadores diferentes. Además, se recomienda seguir las convenciones de nomenclatura y estilo establecidas por las convenciones de codificación de Java (por ejemplo, CamelCase para clases y camelCase para variables y métodos).

Estas son solo algunas categorías comunes de identificadores en Java. También existen identificadores para paquetes, enumeraciones, tipos primitivos, entre otros elementos del lenguaje.

# **IDENTIFICADORES CON SUS TIPOS Y CARACTERÍSTICAS EN POO**

En la programación orientada a objetos (POO), los identificadores se utilizan para dar nombres a las clases, objetos, atributos y métodos. Los identificadores en POO siguen las mismas reglas generales que en otros lenguajes de programación, pero tienen características específicas relacionadas con la estructura y la sintaxis de la POO. A continuación, se presentan los tipos de identificadores más comunes en la programación orientada a objetos y sus características:

1. Identificadores de clases:

Reglas: Los identificadores de clases deben comenzar con una letra mayúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: Persona, Circulo, Empleado

Características: Los identificadores de clases se utilizan para definir tipos de objetos y especificar comportamientos y propiedades comunes para los objetos de ese tipo. Las clases son los bloques fundamentales de construcción de la POO.

1. Identificadores de objetos:

Reglas: Los identificadores de objetos deben comenzar con una letra minúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: persona1, circuloPrincipal, empleadoNuevo

Características: Los identificadores de objetos se utilizan para crear instancias de clases y acceder a sus atributos y métodos. Los objetos son variables que representan entidades específicas dentro de una clase.

1. Identificadores de atributos:

Reglas: Los identificadores de atributos deben comenzar con una letra minúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: nombre, edad, salario

Características: Los identificadores de atributos se utilizan para representar las propiedades o características de un objeto. Los atributos son variables que almacenan datos relacionados con el objeto y definen su estado.

1. Identificadores de métodos:

Reglas: Los identificadores de métodos deben comenzar con una letra minúscula. Pueden contener letras, dígitos y guiones bajos. No pueden tener espacios ni caracteres especiales.

Ejemplo válido: calcularPrecio, obtenerNombre, imprimirSaludo

Características: Los identificadores de métodos se utilizan para definir acciones o comportamientos que un objeto puede realizar. Los métodos representan bloques de código que manipulan los atributos del objeto o realizan operaciones específicas.

Estos son solo algunos ejemplos de identificadores en la programación orientada a objetos. También existen identificadores para paquetes, interfaces, herencia, entre otros conceptos propios de la POO. Es importante seguir las convenciones de nomenclatura y estilo establecidas para el lenguaje de programación que estés utilizando, ya que esto mejora la legibilidad del código y facilita su comprensión por parte de otros programadores.

# **VARIABLES CON SUS TIPOS Y CARACTERÍSTICAS:**

En Java, las variables son contenedores utilizados para almacenar valores en memoria durante la ejecución del programa. Las variables tienen un tipo de datos que determina qué tipo de valores pueden almacenar y las operaciones que se pueden realizar con ellos. A continuación, se presentan los tipos de variables más comunes en Java y sus características:

1. Variables de tipo primitivo:

Tipos: Los tipos primitivos en Java incluyen int, double, boolean, char, byte, short, long y float, entre otros.

Características:

Almacenan valores individuales.

Ocupan una cantidad fija de memoria.

Son copiadas por valor cuando se asignan a otras variables o se pasan como argumentos a métodos.

Tienen un rango y una precisión específicos dependiendo del tipo.

1. Variables de referencia:

Tipos: Los tipos de referencia en Java incluyen clases, interfaces, matrices y enumeraciones.

Características:

Almacenan referencias a objetos en lugar de los objetos en sí.

Ocupan una cantidad de memoria que depende del tamaño del objeto al que hacen referencia.

Permiten acceder a los métodos y atributos del objeto al que hacen referencia.

Se pueden asignar a null para indicar la ausencia de un objeto.

1. Variables locales:

Ubicación: Se declaran dentro de un método, constructor o bloque de código.

Alcance: Su alcance está limitado al bloque de código en el que se declaran.

Características:

Deben ser inicializadas antes de ser utilizadas.

No tienen un valor predeterminado.

Se almacenan en la pila de ejecución y se eliminan automáticamente cuando el bloque de código termina.

1. Variables de instancia:

Ubicación: Se declaran dentro de una clase, pero fuera de cualquier método o bloque de código.

Alcance: Su alcance está limitado a la instancia específica de la clase en la que se declaran.

Características:

Se inicializan automáticamente con valores predeterminados si no se les asigna un valor explícito.

Se almacenan en el heap de memoria y se eliminan cuando el objeto al que pertenecen es recolectado por el recolector de basura.

1. Variables estáticas:

Ubicación: Se declaran como static dentro de una clase.

Alcance: Su alcance está limitado a la clase en la que se declaran.

Características:

Se comparten entre todas las instancias de la clase.

Se inicializan automáticamente con valores predeterminados si no se les asigna un valor explícito.

Se almacenan en el heap de memoria y se eliminan cuando la clase deja de ser utilizada.

Es importante tener en cuenta que, además de los tipos primitivos y de referencia, Java también permite el uso de clases envolventes (wrapper classes) para trabajar con tipos primitivos como objetos.

Estas son solo algunas categorías y características comunes de variables en Java. La elección del tipo de variable adecuado depende del tipo de datos que se desee almacenar y de cómo se utilizará en el programa.

# **ROLES EN JAVA:**

En el contexto de la programación Java, no existe un concepto nativo de "roles" como una entidad específica del lenguaje. Sin embargo, en el desarrollo de aplicaciones y sistemas, se puede implementar el concepto de roles utilizando diferentes enfoques y patrones de diseño. Aquí hay algunas formas comunes en las que se pueden implementar roles en Java:

Herencia de clases:

Puedes utilizar la herencia para definir diferentes clases que representen roles específicos en tu sistema. Cada clase puede heredar de una clase base o de interfaces comunes y proporcionar implementaciones específicas para las funcionalidades asociadas con ese rol en particular.

Interfaces:

Las interfaces en Java permiten definir un conjunto de métodos que deben ser implementados por cualquier clase que adopte esa interfaz. Puedes crear interfaces que representen diferentes roles en tu sistema y luego implementar esas interfaces en las clases correspondientes. De esta manera, cada clase puede tener múltiples roles al implementar varias interfaces.

Patrones de diseño:

Algunos patrones de diseño, como el patrón Strategy, el patrón Observer o el patrón Visitor, pueden ayudar a definir y gestionar roles en una aplicación Java. Estos patrones proporcionan estructuras y mecanismos para asignar roles específicos a objetos y permitir su interacción y comportamiento según el rol asignado.

Anotaciones personalizadas:

Java también permite definir anotaciones personalizadas, que son metadatos que se pueden agregar a clases, métodos o campos. Puedes crear anotaciones personalizadas para marcar clases o métodos con roles específicos y luego utilizar la reflexión para acceder a esa información y realizar acciones basadas en los roles.

# **HERENCIAS EN JAVA:**

La herencia es un concepto fundamental en la programación orientada a objetos (POO) y también está presente en Java. La herencia permite la creación de nuevas clases basadas en clases existentes, lo que promueve la reutilización de código y la organización de las relaciones entre objetos. En Java, se puede lograr la herencia mediante la utilización de la palabra clave extends. A continuación, se explican los aspectos clave de la herencia en Java:

1. Clase base (superclase):

* También conocida como clase padre o clase superior.
* Es la clase existente de la cual se derivan nuevas clases.
* Puede contener atributos, métodos y constructores que serán heredados por las subclases.

1. Clase derivada (subclase):

* También conocida como clase hija o clase inferior.
* Se crea mediante la extensión de una clase base existente utilizando la palabra clave extends.
* Hereda todos los miembros (atributos y métodos) no privados de la clase base.
* Puede agregar nuevos miembros y también puede anular (sobrescribir) los métodos heredados de la clase base.

1. Relación "es-un":

* La herencia en Java establece una relación "es-un" entre la clase base y la clase derivada.
* La clase derivada se considera un tipo más específico de la clase base.
* Acceso a los miembros heredados:

Los miembros heredados (atributos y métodos no privados) de la clase base se pueden acceder directamente desde la clase derivada utilizando el operador de acceso punto (.).

En caso de que existan miembros con el mismo nombre en la clase base y la clase derivada, se puede utilizar la palabra clave super para acceder a los miembros de la clase base.

1. Herencia múltiple:

* En Java, una clase solo puede heredar de una única clase base (herencia simple).
* Sin embargo, una clase puede implementar múltiples interfaces, lo que proporciona una forma de lograr ciertos aspectos de la herencia múltiple.

La herencia en Java permite crear jerarquías de clases que reflejan las relaciones entre los objetos del dominio del problema. Proporciona una forma de compartir y reutilizar el código común a través de la estructura de clases. Sin embargo, es importante tener en cuenta que un uso excesivo de la herencia puede llevar a una estructura de clases compleja y poco mantenible. Es fundamental comprender y planificar cuidadosamente la jerarquía de clases y el diseño de la herencia para obtener un código limpio y mantenible.

Variables con sus tipos y características:

En Java, las variables son contenedores utilizados para almacenar valores en memoria durante la ejecución del programa. Las variables tienen un tipo de datos que determina qué tipo de valores pueden almacenar y las operaciones que se pueden realizar con ellos. A continuación, se presentan los tipos de variables más comunes en Java y sus características:

1. Variables de tipo primitivo:

Tipos: Los tipos primitivos en Java incluyen int, double, boolean, char, byte, short, long y float, entre otros.

Características:

Almacenan valores individuales.

Ocupan una cantidad fija de memoria.

Son copiadas por valor cuando se asignan a otras variables o se pasan como argumentos a métodos.

Tienen un rango y una precisión específicos dependiendo del tipo.

2. Variables de referencia:

Tipos: Los tipos de referencia en Java incluyen clases, interfaces, matrices y enumeraciones.

Características:

Almacenan referencias a objetos en lugar de los objetos en sí.

Ocupan una cantidad de memoria que depende del tamaño del objeto al que hacen referencia.

Permiten acceder a los métodos y atributos del objeto al que hacen referencia.

Se pueden asignar a null para indicar la ausencia de un objeto.

3. Variables locales:

Ubicación: Se declaran dentro de un método, constructor o bloque de código.

Alcance: Su alcance está limitado al bloque de código en el que se declaran.

Características:

Deben ser inicializadas antes de ser utilizadas.

No tienen un valor predeterminado.

Se almacenan en la pila de ejecución y se eliminan automáticamente cuando el bloque de código termina.

4. Variables de instancia:

Ubicación: Se declaran dentro de una clase, pero fuera de cualquier método o bloque de código.

Alcance: Su alcance está limitado a la instancia específica de la clase en la que se declaran.

Características:

Se inicializan automáticamente con valores predeterminados si no se les asigna un valor explícito.

Se almacenan en el heap de memoria y se eliminan cuando el objeto al que pertenecen es recolectado por el recolector de basura.

5. Variables estáticas:

Ubicación: Se declaran como static dentro de una clase.

Alcance: Su alcance está limitado a la clase en la que se declaran.

Características:

Se comparten entre todas las instancias de la clase.

Se inicializan automáticamente con valores predeterminados si no se les asigna un valor explícito.

Se almacenan en el heap de memoria y se eliminan cuando la clase deja de ser utilizada.

Es importante tener en cuenta que, además de los tipos primitivos y de referencia, Java también permite el uso de clases envolventes (wrapper classes) para trabajar con tipos primitivos como objetos.

Estas son solo algunas categorías y características comunes de variables en Java. La elección del tipo de variable adecuado depende del tipo de datos que se desee almacenar y de cómo se utilizará en el programa.

Roles en Java:

En el contexto de la programación Java, no existe un concepto nativo de "roles" como una entidad específica del lenguaje. Sin embargo, en el desarrollo de aplicaciones y sistemas, se puede implementar el concepto de roles utilizando diferentes enfoques y patrones de diseño. Aquí hay algunas formas comunes en las que se pueden implementar roles en Java:

Herencia de clases:

Puedes utilizar la herencia para definir diferentes clases que representen roles específicos en tu sistema. Cada clase puede heredar de una clase base o de interfaces comunes y proporcionar implementaciones específicas para las funcionalidades asociadas con ese rol en particular.

Interfaces:

Las interfaces en Java permiten definir un conjunto de métodos que deben ser implementados por cualquier clase que adopte esa interfaz. Puedes crear interfaces que representen diferentes roles en tu sistema y luego implementar esas interfaces en las clases correspondientes. De esta manera, cada clase puede tener múltiples roles al implementar varias interfaces.

Patrones de diseño:

Algunos patrones de diseño, como el patrón Strategy, el patrón Observer o el patrón Visitor, pueden ayudar a definir y gestionar roles en una aplicación Java. Estos patrones proporcionan estructuras y mecanismos para asignar roles específicos a objetos y permitir su interacción y comportamiento según el rol asignado.

Anotaciones personalizadas:

Java también permite definir anotaciones personalizadas, que son metadatos que se pueden agregar a clases, métodos o campos. Puedes crear anotaciones personalizadas para marcar clases o métodos con roles específicos y luego utilizar la reflexión para acceder a esa información y realizar acciones basadas en los roles.

Herencias en Java:

La herencia es un concepto fundamental en la programación orientada a objetos (POO) y también está presente en Java. La herencia permite la creación de nuevas clases basadas en clases existentes, lo que promueve la reutilización de código y la organización de las relaciones entre objetos. En Java, se puede lograr la herencia mediante la utilización de la palabra clave extends. A continuación, se explican los aspectos clave de la herencia en Java:

1. Clase base (superclase):

• También conocida como clase padre o clase superior.

• Es la clase existente de la cual se derivan nuevas clases.

• Puede contener atributos, métodos y constructores que serán heredados por las subclases.

2. Clase derivada (subclase):

• También conocida como clase hija o clase inferior.

• Se crea mediante la extensión de una clase base existente utilizando la palabra clave extends.

• Hereda todos los miembros (atributos y métodos) no privados de la clase base.

• Puede agregar nuevos miembros y también puede anular (sobrescribir) los métodos heredados de la clase base.

3. Relación "es-un":

• La herencia en Java establece una relación "es-un" entre la clase base y la clase derivada.

• La clase derivada se considera un tipo más específico de la clase base.

• Acceso a los miembros heredados:

Los miembros heredados (atributos y métodos no privados) de la clase base se pueden acceder directamente desde la clase derivada utilizando el operador de acceso punto (.).

En caso de que existan miembros con el mismo nombre en la clase base y la clase derivada, se puede utilizar la palabra clave super para acceder a los miembros de la clase base.

4. Herencia múltiple:

• En Java, una clase solo puede heredar de una única clase base (herencia simple).

• Sin embargo, una clase puede implementar múltiples interfaces, lo que proporciona una forma de lograr ciertos aspectos de la herencia múltiple.

La herencia en Java permite crear jerarquías de clases que reflejan las relaciones entre los objetos del dominio del problema. Proporciona una forma de compartir y reutilizar el código común a través de la estructura de clases. Sin embargo, es importante tener en cuenta que un uso excesivo de la herencia puede llevar a una estructura de clases compleja y poco mantenible. Es fundamental comprender y planificar cuidadosamente la jerarquía de clases y el diseño de la herencia para obtener un código limpio y mantenible.

# **ATRIBUTOS**

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos se guardan en variables denominadas de instancia, y cada objeto particular puede tener valores distintos para estas variables.

Las variables de instancia también denominados miembros dato, son declaradas en la clase, pero sus valores son fijados y cambiados en el objeto.

Además de las variables de instancia hay variables de clase, las cuales se aplican a la clase y a todas sus instancias. Por ejemplo, el número de ruedas de un automóvil es el mismo cuatro, para todos los automóviles.

TIPOS DE ATRIBUTO

# **VARIABLES DE INSTANCIA**

Cuando se declara el atributo o variable miembro euros en la clase Precio de la siguiente forma:

public class Precio {

// Declaracion de atributos o variables miembro

public double euros;

// Declaracion de metodos . . .

}

Se está declarando el atributo euros como una variable de instancia. En consecuencia, cada vez que se crea una instancia de la clase Precio, se reserva espacio en memoria para una variable de instancia euros. Por ejemplo, el código:

// Creacion de dos instancias de la clase precio

Precio p = new Precio ();

p.pone(56.8);

Precio q = new Precio ();

q.pone(75.6);

genera dos instancias de la clase Precio como muestra la Figura 9.2. En este caso, cada una de las dos instancias, p y q, de la clase Precio tiene una variable de instancia euros propia. Las respectivas llamadas al método ponen para cada instancia (p.pone(56.8) y q.pone(75.6)), permiten asignar un valor a las variables de instancia correspondientes.

# **VARIABLES DE CLASE**

Las variables de clase son atributos diferentes de las variables de instancia. Las variables de clase implican una sola zona de memoria reservada para todas las instancias de la clase, y no una copia por objeto, como sucede con las variables de instancia. Para diferenciarlas de éstas en el código fuente de Java, las variables de clase se distinguen con el modificador static en la declaración del atributo correspondiente. Por defecto (si no se indica la palabra static), el atributo declarado se considera variable de instancia.

Durante la ejecución del programa, el sistema reserva un único espacio en memoria para cada variable estáticas o de clase independientemente del número de instancias creadas de una clase. Esta reserva se produce la primera vez que encuentra dicha clase en el código, de forma que todas las instancias pertenecientes a una clase comparten la misma variable de clase. A diferencias de las variables globales fuera de la POO, las variables de clase garantizan la encapsulación.

Las variables de clase sirven para almacenar características comunes (constantes) a todos los objetos (número de ruedas de una bicicleta) o para almacenar características que dependen de todos los objetos (número total de billetes de lotería). Por ejemplo, la clase CuentaBancaria tiene una variable de instancia, saldo, y una variable de clase, totalCuentas.

public class CuentaBancaria {

// Atributos o variables miembro

public double saldo; // Variable de instancia

public static int totalCuentas=0; // Variable de clase

// Declaraciones de metodos

}

La creación de varias instancias de la clase CuentaBancaria no conlleva la existencia de varias variables totalCuentas. Durante la ejecución de un programa que utilice la clase CuentaBancaria sólo existirá una variable de clase totalCuentas, independientemente del número de instancias de la clase CuentaBancaria que se generen. Es más, no es necesario siquiera que exista una instancia de la clase, para que lo haga la variable de clase. De hecho, se inicializan a false, cero o null (dependiendo de su tipo) antes de que se genere una instancia de la clase.

// Creacion de dos instancias de la clase CuentaBancaria

CuentaBancaria c1 = new CuentaBancaria();

CuentaBancaria c2 = new CuentaBancaria();

# **CONTROLES SWING**

Conjunto de herramientas de tipo control que permiten generar acciones o eventos dentro de una aplicación

Controles

• Etiqueta (jLabel): un área de visualización de una cadena de texto o una imagen, o ambas.

• Botón (jButton): un botón "oprimir"

• Botón de 2 posiciones (jToggleButton): un botón con dos estados.

• Casilla de activaciónn (jCheckBox): un elemento puede ser selecccionado o deseleccionado. Por convención, cualquier jCheckBox en un grupo puede ser seleccionado.

• Botón de opción (jRadioButton): un elemento puede ser selecccionado o deseleccionado. Usado con un objeto buttonGroup para crar un grupo de botones en el cual un botón puede ser seleccionado a la vez.

• Grupo de botones (buttonGroup): esta clase se utiliza para crear un conjunto de botones en que sólo se podrá seleccionar uno.

• Lista desplegable (jComboBox): un componente que combina un botón o campo editable y una lista desplegable.

• Lista (jList): un componente que permite seleccionar uno o más objetos de una lista.

• Campo de texto (jTextField): un componente ligero que permite editar una línea idividual de texto.

• Área de texto (jTextArea): un área de varias líneas que muestra textos sin adornos.

• Barra de desplazamiento (jScrollBar): un componente que permite ajustar el contenido del área visible de otro componente.

• Deslizador (jSlider): un componente que permite seleccionar gráficamente un valor al deslizar un tirador dentro de un intervalo limitado.

• Barra de progreso (jProgressBar): informa habitualmente del progreso de algún trabajo, mostrando un porcentaje de realización y quizás una visualización textual de este porcentaje.

• Cuadro formateado (jFormattedTextField): un componente que permite editar un valor formateado (línea de texto), así como recuperar un objeto particular una vez que se ha editado el texto.

• Cuadro de contraseña (jPasswordField): un componenteque permite editar una línea individual de texto donde se indica que algo se ha tecleado, pero no muestra los caracteres originales.

• Spinner (jSpinner): cuadro de entrada de una línea individual que permite seleccionar un número o un valor de una secuencia ordenada.

• Separador (jSeparator): un componente de utilidad diversa para realizar líneas divisorias

• Panel de texto (jTextPane): un componente de texto que se puede marcar con agtributos que se representarán gráficamente.

• Panel editor (jEditorPane): un componente de texto para editar diferentes contenidos

• Árbol (jTree): un control que muestra un conjunto de datos jerárquicos como un boceto.

• Tabla (jTable): un componente utilizado para mostrar y editar las habituales tablas bidimensionales con celdas.

Entrada y Salida de Datos con jTextField()

La Entrada y Salida de datos desde controles visuales se basa en datos tipo cadena, lo que hace necesario su conversión al tipo de datos que se requiera en la programación.

• getText() - Entrada: Permite el ingreso de datos a través de un control visual de Java jTextField

Ejemplo: Para dato1, dato2 y resultados de tipo double

dato1=Double.valueOf(jTextField1.getText());

dato2=Double.valueOf(jTextField2.getText());

resultado = dato1+dato2;

• setText() - Salida: Permite la salida de datos a través de un control visual de Java jTextField

Ejemplo: Para resultado tipo double y cadenaSalida tipo String

cadenaSalida= Double.toString(resultado);

jTextField3.setText(cadenaSalida);

Etiqueta (jLabel) para Textos / Imágenes

• Para textos: incluye textos al diseño que pueden ser formateados en tamaño, estilo, color, etc...

• Para imágenes fijas: incluye imágenes al diseño insertando arcuivos de cualquier formato como jpg, png, etc.. Los archivos pueden ser cargardos desde el proyecto en uno de sus paquetes o externos desde cualquier ubicación, utilizando la opcion icon en Propiedades del jLabel.

• Para imágenes por asignación: carga diferentes imágenes sobre un mismo jLabel por asignación de istintos archivos según la programación, que permite ir cambiado una imágen en el diseño.

Utilizando la librería: import javax.swing.ImageIcon;

jLabel.setIcon(new ImageIcon(getClass().getResource("imagen1.png")));

Sin la librería:

jLabel.setIcon(new javax.swing.ImageIcon(getClass().getResource("imagen1.png")));

• Para cargar una imagen de fondo: permite fijar una imagen de fondo a nivel de presentación o para diseñar montajes de controles sobre la imagen.

En la forma de diseño del JFrame, en el navegador, con el botón derecho del mause sobre [JFrame], en Activar gestor de distribución, se selecciona la opción Diseño absoluto.

Luego del diseño de controles se utiliza un JLabel para fijar una imagen en el área del Frame deseado. Si se monta el fondo y luego se carga los controles, se debe ordenar en el navegador asegurándose que el jLaber con la imagen de fondo este siempre al final.

Controles de Selección

• buttonGroup

Se programa la constructora adicionando los jRadioButton al grupo.

Ejemplo: buttonGroup1.add(jRadioButton1); ...

• jRadioButton, jCheckBox

Se programa la selección de un botón de opción.

Ejemplo: if (jRadioButton1.isSelected()) ...

• jComboBox, jList

Se programa la selección de cada uno de los ítem de la caja combinada.

Ejemplos:

Comparando con cada ítem: if (jComboBox1.getSelectedItem().equals("ítem1"))...

Comparando con la posición de cada ítem (0, 1, 2,…): if (jComboBox1.getSelectedIndex()==0)...

Conversión de datos entre Numéricos y Cadena

• valueOf de la Clase String

• parseInt de la Clase Integer

• parseFloat de la Clase Float

• Etc...

Ejemplos:

• De String a Número: int valor = Integer.parseInt(cadena); //Long, Float, Double

• De Número a String: String cadena = Integer.toString(valor); //Long, Float, Double

• De Double a String: String cadena = String.valueOf(valorCualquierTipo);

• De String a Double: double valor = Double.valueOf(cadena);

Inicialización de algunos controles

• jProgressBar1.setValue(0);

• buttonGroup1.clearSelection();

• jRadioButton1.setSelected(false);

• jCheckBox1.setSelected(false);

• jTextField1.setText(String.valueOf(0));

• jTextField1.setText("");

• jTextField1.setText(null);

• jComboBox1.setSelectedIndex(0); // ubica el campo en el primer ítem

• jComboBox1.setSelectedIndex(-1); // se ubica en una posición sin ítem

# **CUADROS DE DIALOGO**

son una opción perfecta para enviar al usuario mensajes de todo tipo: de error, de precaución, de información, etc. Además, permiten mostrar uno o más botones, e incluso solicitar información al usuario en forma de cuadro de texto o combo.

Hay que distinguir los cuadros modales de los no modales, siendo los primeros los que necesitan que el usuario responda antes de continuar el programa, y los segundos los que permanecen en la pantalla ye stán disponibles para ser utilizados aunque permiten que el usuario siga trabajando con el programa.

Para crear cuadros de diálogo utilizamos el objeto JOptionPane, y en función del método a elegir, mostrará un tipo u otro. Veamos los métodos más utilizados:

•

• showMessageDialog: es el más habitual, y despliega un cuadro de diálogo con un único botón (Aceptar). Tiene varias sobrecargas del método, es decir, podemos elegir el mismo método con dos, cuatro o cinco parámetros, que se indican a continuación:

MyIcon icon = new MyIcon();

JOptionPane.showMessageDialog(Componente padre, mensaje, título, tipo de mensaje, icono);

showConfirmDialog: por defecto pide al usuario que confirme entre las opciones Sí y No, pero puede ser configurado para adaptarse a las necesidades del programador. Dispone de cuatro sobrecargas del método donde podemos elegir los siguientes parámetros:

JOptionPane.showConfirmDialog(Componente padre, mensaje, titulo, tipo de seleccion, tipo de mensaje, icono);

showOptionDialog: despliega un cuadro de diálogo diseñado por el programador y tiene los siguientes parámetros:

JOptionPane.showOptionDialog(padre, mensaje, titulo, tipo de seleccion,

# **JASPER REPORT**

Es una potente herramienta en código abierto de generación de informes que permite generar información detallada en formato PDF, HTML, XLS, CSV o XML. Ha sido desarrollada completamente en Java, por lo que puede ser usada en una amplia variedad de aplicaciones Java para generar contenido dinámico.

JasperReports trabaja de forma muy similar a un compilador y un interprete. Se siguen los siguientes pasos:

1. EL desarrollador diseña el reporte codificándolo en XML de acuerdo a las etiquetas y atributos definidos en jasperreports.dtd. En el XML, el desarrollador puede definir todo le formato del informe describiendo la posición del texto, gráficos, dibujos, realizar cálculos, etc,...

2. El archivo XML es compilado mediante el método compileReport(), localizado en la clase net.sf.jasperreports.engine.JasperCompileManager

3. Una vez obtenido el archivo .jasper, se necesitan datos dinámicos que cumplimenten el reporte. Por ello se acude al JRDataSource. Para rellenar el informe, se pueden usar los métodos \*fillReportXXX()\* de la clase net.sf.jasperreports.engine.JasperFillManager. Estos métodos reciben como parámetro el objeto del diseño del informe, o un fichero que representa tal objeto, en un formulario serializado, así como una conexión JDBC a la base de datos en donde se almacenan los datos que rellenarán el informe o un Bean que contenga estos datos.

4. Se obtiene un archivo .print exportable en varios formatos PDF, HTML, XML, XSL, CVS.

Para desarrollar este ejemplo, lo primero que ha de hacerse es el diseño del informe, utilizando para ello la herramienta gráfica iReport. Se trata de un diseño muy sencillo en el que va a haber sólo dos campos de texto: uno que mostrará el título del informe y otro que simplemente contendrá un nombre.

El funcionamiento es bien sencillo. El usuario ha de introducir dos valores: la tarea a ejecutar y la ubicación del fichero a manipular. Si se sigue el proceso completo, es decir, si se parte del diseño de iReport sin compilar, cuya extensión es .jrxml, se debe ejecutar la aplicación con los siguientes argumentos:

a. Usar el jrxml para generar el diseño compilado jasper, mediante el método compileReportToFile:

compile E:/eclipse/workspace/JasperConfluence/src/prueba3

b. Usar el jasper para generar el fichero jrprint, mediante el método fillReportToFile. En este caso, se utilizan tres argumentos: la ruta del fichero jasper, un map con los parámetros con los que se rellenará el informe (el título y un nombre) y un datasource vacío (JREmptyDataSource), diseñado para que JasperReports prescinda de utilizar el datasource que todo informe de iReport obliga a tener. En este caso, prueba3.jrxml disponía de una conexión Hsql.

fill E:/eclipse/workspace/JasperConfluence/src/prueba3

c. Usar el jrprint para generar el fichero final, en este caso PDF, mediante la función exportReportToPdfFile. Además, se envían dos parámetros: el título del informe (en la variable que hemos denominado ReportTitle) y un nombre (en la variable que hemos llamado Name).

pdf E:/eclipse/workspace/JasperConfluence/src/prueba3

d. Opcionalmente, si se desea enviar el informe a la impresora, también emplea el jrprint. Usa el método printReport y, como es obvio, no se genera ningún otro fichero.

print E:/eclipse/workspace/JasperConfluence/src/prueba3

package com.cassiopea.ejemplo;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import java.io.File;

import java.util.HashMap;

import java.util.Map;

import net.sf.jasperreports.engine.JREmptyDataSource;

import net.sf.jasperreports.engine.JRException;

import net.sf.jasperreports.engine.JRExporterParameter;

import net.sf.jasperreports.engine.JasperCompileManager;

import net.sf.jasperreports.engine.JasperExportManager;

import net.sf.jasperreports.engine.JasperFillManager;

import net.sf.jasperreports.engine.JasperPrint;

import net.sf.jasperreports.engine.JasperPrintManager;

import net.sf.jasperreports.engine.export.JExcelApiExporter;

import net.sf.jasperreports.engine.export.JRCsvExporter;

import net.sf.jasperreports.engine.export.JRXlsExporter;

import net.sf.jasperreports.engine.export.JRXlsExporterParameter;

import net.sf.jasperreports.engine.util.JRLoader;

/\*\*

\* Clase en la que se ejecuta cada uno de los pasos de la generación de informes

\*

\* por separado pasándole como primer argumento la acción a realizar:

\*

\* compile | fill | print | pdf | xml | xmlembed | html | xls | csv

\*

\* y como segundo argumento el path absoluto del fichero del informe

\*

\*/

public class EjemploPorPasos

{

private static final Log logger = LogFactory.getLog(EjemploPorPasos.class);

private static final String TASK\_COMPILE = "compile";

private static final String TASK\_FILL = "fill";

private static final String TASK\_PRINT = "print";

private static final String TASK\_PDF = "pdf";

private static final String TASK\_XML = "xml";

private static final String TASK\_XML\_EMBED = "xmlembed";

private static final String TASK\_HTML = "html";

private static final String TASK\_XLS = "xls";

private static final String TASK\_XLS2 = "xls2";

private static final String TASK\_CSV = "csv";

public static void main(String[] args){

String fileName = null;

String taskName = null;

if(args.length != 2)

{

usage();

return;

}

taskName=args[0].toLowerCase();

fileName=args[1];

logger.info("main(String[]) - taskName: " + taskName);

logger.info("main(String[]) - fileName: " + fileName);

System.exit(genera(taskName,fileName));

}

public static int genera(String taskName,String fileName)

{

try

{

long start = System.currentTimeMillis();

//Compila el fichero del diseño del informe, creado con iReport, pasándole de .jrxml a .jasper

if (TASK\_COMPILE.equals(taskName)){

JasperCompileManager.compileReportToFile(fileName + ".jrxml");

logger.info("genera(String, String) - Compile time : " + (System.currentTimeMillis() - start));

//Rellena el informe con dos parámetros: su título y un nombre. No utiliza ningún datasource

//Genera un fichero .jrprint, necesario para la exportación del informe a los diferentes formatos

}else if (TASK\_FILL.equals(taskName)){

Map param = new HashMap();

param.put("ReportTitle", "Mi Primer Informe");

param.put("Name", "mundo");

JasperFillManager.fillReportToFile(fileName + ".jasper", param, new JREmptyDataSource());

logger.info("genera(String, String) - Filling time : " + (System.currentTimeMillis() - start));

}else if (TASK\_PRINT.equals(taskName)){

JasperPrintManager.printReport(fileName + ".jrprint", true);

logger.info("genera(String, String) - Printing time : " + (System.currentTimeMillis() - start));

}else if (TASK\_PDF.equals(taskName)){

JasperExportManager.exportReportToPdfFile(fileName + ".jrprint");

logger.info("genera(String, String) - PDF creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_XML.equals(taskName)){

JasperExportManager.exportReportToXmlFile(fileName + ".jrprint", false);

logger.info("genera(String, String) - XML creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_XML\_EMBED.equals(taskName)){

JasperExportManager.exportReportToXmlFile(fileName + ".jrprint", true);

logger.info("genera(String, String) - XML creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_HTML.equals(taskName)){

JasperExportManager.exportReportToHtmlFile(fileName + ".jrprint");

logger.info("genera(String, String) - HTML creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_XLS.equals(taskName)){ // uso de POI

File sourceFile = new File(fileName + ".jrprint");

JasperPrint jasperPrint = (JasperPrint)JRLoader.loadObject(sourceFile);

File destFile = new File(sourceFile.getParent(), jasperPrint.getName() + ".xls");

JRXlsExporter exporter = new JRXlsExporter();

exporter.setParameter(JRExporterParameter.JASPER\_PRINT, jasperPrint);

exporter.setParameter(JRExporterParameter.OUTPUT\_FILE\_NAME, destFile.toString());

exporter.setParameter(JRXlsExporterParameter.IS\_ONE\_PAGE\_PER\_SHEET, Boolean.TRUE);

exporter.exportReport();

logger.info("genera(String, String) - XLS creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_XLS2.equals(taskName)){ // uso de jxl

File sourceFile = new File(fileName + ".jrprint");

JasperPrint jasperPrint = (JasperPrint)JRLoader.loadObject(sourceFile);

File destFile = new File(sourceFile.getParent(), jasperPrint.getName() + "2.xls");

JExcelApiExporter jExcelApiExporter = new JExcelApiExporter();

jExcelApiExporter.setParameter(JRExporterParameter.JASPER\_PRINT, jasperPrint);

jExcelApiExporter.setParameter(JRExporterParameter.OUTPUT\_FILE\_NAME, destFile.toString());

jExcelApiExporter.exportReport();

logger.info("genera(String, String) - XLS2 creation time : " + (System.currentTimeMillis() - start));

}else if (TASK\_CSV.equals(taskName)){

File sourceFile = new File(fileName + ".jrprint");

JasperPrint jasperPrint = (JasperPrint)JRLoader.loadObject(sourceFile);

File destFile = new File(sourceFile.getParent(), jasperPrint.getName() + ".csv");

JRCsvExporter exporter = new JRCsvExporter();

exporter.setParameter(JRExporterParameter.JASPER\_PRINT, jasperPrint);

exporter.setParameter(JRExporterParameter.OUTPUT\_FILE\_NAME, destFile.toString());

exporter.exportReport();

logger.info("genera(String, String) - CSV creation time : " + (System.currentTimeMillis() - start));

}else{

usage();

}

}catch (JRException e){

logger.error("genera(String, String)", e);

return 1;

}catch (Exception e){

logger.error("genera(String, String)", e);

return 1;

}

return 0;

}

private static void usage(){

logger.info("usage() - JasperApp usage:");

logger.info("usage() - tjava JasperApp task file");

logger.info("usage() - tTasks : compile | fill | print | pdf | xml | xmlembed | html | xls | csv ");

}

}