Contenido

Introducción

En esta bitácora se recopilan y documentan diversos sistemas desarrollados en Kotlin, cada uno diseñado para abordar necesidades específicas en diferentes contextos. A través de estos proyectos, se exploran conceptos fundamentales de programación orientada a objetos, como clases, interfaces y herencia, así como patrones de diseño que optimizan la organización y la reutilización del código.

Los sistemas incluyen:

1. **Gestión de Impuestos Prediales**: Este sistema permite calcular el impuesto total a pagar basado en propiedades, considerando factores como la ubicación, el tamaño y descuentos aplicables según características del contribuyente.
2. **Cálculo de Nómina**: Se implementa un sistema de nómina para trabajadores, donde se consideran diferentes tipos de empleados (fijos, por horas, por comisión) y se calculan sus salarios en función de sus horas trabajadas y otros factores.
3. **Sistema de Reservas**: A través de este sistema, se gestionan reservas individuales y grupales para vuelos, con opciones para aplicar descuentos y calcular costos de manera dinámica, incorporando la flexibilidad de reservas anidadas.

Cada uno de estos sistemas no solo proporciona funcionalidad práctica, sino que también demuestra la capacidad de Kotlin para manejar diferentes estructuras y lógica de programación, creando aplicaciones que son robustas y fáciles de mantener. La integración de estos sistemas en una única bitácora permite una visión holística del aprendizaje y el desarrollo, sirviendo como una referencia valiosa para futuras implementaciones y mejoras.

.

Objetivo de la práctica

El objetivo de esta bitácora es documentar de manera clara y sistemática el desarrollo y la implementación de varios sistemas en Kotlin, enfocados en la gestión de impuestos, nómina de empleados y reservas de servicios. A través de esta recopilación, se busca:

1. **Registrar el Proceso de Aprendizaje**: Capturar el progreso y la evolución en la comprensión de conceptos de programación orientada a objetos, patrones de diseño y mejores prácticas de desarrollo de software.
2. **Facilitar la Revisión y el Análisis**: Proporcionar una referencia detallada que permita analizar el código, la lógica y la estructura de cada sistema, facilitando la identificación de áreas de mejora y optimización.
3. **Promover la Reutilización de Código**: Fomentar la reutilización de componentes y funciones desarrolladas en los distintos sistemas, contribuyendo a la eficiencia y la mantenibilidad del código en futuros proyectos.
4. **Impulsar la Innovación**: Servir como base para la generación de nuevas ideas y la implementación de mejoras en los sistemas existentes, explorando nuevas funcionalidades y adaptaciones según las necesidades del usuario.

A través de este objetivo, la bitácora no solo actúa como un registro del trabajo realizado, sino también como una herramienta para el crecimiento profesional y la consolidación de conocimientos en el ámbito del desarrollo de software.

.

Resultados obtenidos

1. Impuesto Predial

Ejecución de la metodología especificada.

Funcionamiento del Sistema

1. **Entrada de Datos**:
    - El sistema solicita los datos de la persona (nombre, edad y si es madre soltera).
    - Luego, permite agregar información sobre los predios, incluyendo la zona y el tamaño en metros cuadrados.
2. **Cálculo del Impuesto**:
    - Para cada predio, calcula un costo base multiplicando el costo por metro cuadrado de la zona por el tamaño del predio.
    - Luego, aplica un descuento basado en la situación de la persona y el mes de pago.
3. **Descuento Total y Resultado**:
    - El sistema suma el impuesto de cada predio y aplica los descuentos apropiados, mostrando el total a pagar.

Propósito del Sistema

Este sistema podría ser parte de una **aplicación de gestión de impuestos municipales**, donde ciudadanos o autoridades locales pueden calcular los impuestos prediales de forma simplificada. Facilita:

- La captación de información detallada sobre cada predio.
- La aplicación de descuentos basados en políticas de apoyo social.
- El cálculo de un impuesto total ajustado y preciso, según la normativa local.

Código de instrucciones ejecutadas por cada sentencia utilizada.

código clase Persona:

```kotlin
// Persona.kt
data class Persona(val nombre: String, val edad: Int, val esMadreSoltera: Boolean)
```

Línea de Código	Explicación
data class Persona	Define una clase de datos llamada Persona. Las "data classes" en Kotlin se usan para almacenar datos y automáticamente generan métodos útiles como toString, equals, y hashCode.
(val nombre: String,	Declara una propiedad nombre de tipo String para la clase Persona, que representa el nombre de la persona. La palabra clave val indica que es inmutable.
val edad: Int,	Declara una propiedad edad de tipo Int, que representa la edad de la persona. Al ser val, esta propiedad también es inmutable.
val esMadreSoltera: Boolean)	Declara una propiedad esMadreSoltera de tipo Boolean, que indica si la persona es madre soltera (true) o no (false). También es inmutable.

código de la clase Predio:

```kotlin
// Predio.kt
data class Predio(val zona: Zona, val metrosCuadrados: Double)
```

Línea de Código	Explicación
data class Predio	Define una clase de datos llamada Predio. Al ser una "data class", la clase Predio contará automáticamente con métodos como toString, equals, y hashCode.
(val zona: Zona,	Declara una propiedad zona de tipo Zona para la clase Predio, la cual representa la zona en la que se encuentra el predio. Zona puede ser otra clase o enum.
val metrosCuadrados: Double)	Declara una propiedad metrosCuadrados de tipo Double, que representa el tamaño del predio en metros cuadrados. Al ser val, es una propiedad inmutable.

código de la clase Zona:

```kotlin
// Zona.kt
data class Zona(val clave: String, val nombre: String, val costoPorMetro: Double)
```

Línea de Código	Explicación
data class Zona	Define una clase de datos llamada Zona. Al ser una "data class", Zona obtiene métodos generados automáticamente como toString, equals, y hashCode.
(val clave: String,	Declara una propiedad clave de tipo String para la clase Zona, que representa una clave única o código de identificación de la zona. Es inmutable (val).
val nombre: String,	Declara una propiedad nombre de tipo String, que representa el nombre descriptivo de la zona. Al ser val, esta propiedad es inmutable.
val costoPorMetro: Double)	Declara una propiedad costoPorMetro de tipo Double, que representa el costo por metro cuadrado en esa zona específica. Es una propiedad inmutable.

código de la clase ImpuestoPredial:

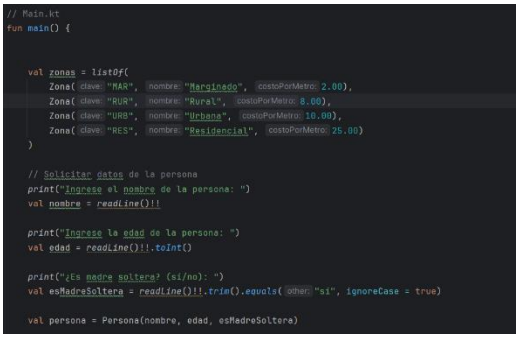
```kotlin
// ImpuestoPredial.kt
    class ImpuestoPredial {
    // Método para calcular el impuesto total
    fun calcularImpuestoTotal(predios: List<Predio>, persona: Persona, mes: Int): Double {
        val total = predios.sumOf { predio ->
            val costoBase = predio.zona.costoPorMetro * predio.metrosCuadrados
            val descuento = calcularDescuento(persona, mes)
            costoBase * (1 - descuento)
        }
        return total
    }

    // Método para determinar el descuento aplicable
    private fun calcularDescuento(persona: Persona, mes: Int): Double {
        val esTemporada = mes == 1 || mes == 2
        val descuentoBase = when {
            persona.edad >= 70 || persona.esMadreSoltera -> if (esTemporada) 0.70 else 0.50
            else -> if (esTemporada) 0.40 else 0.0
        }
        return descuentoBase
    }
}
```

Línea de Código	Explicación
class ImpuestoPredial	Define una clase llamada ImpuestoPredial.

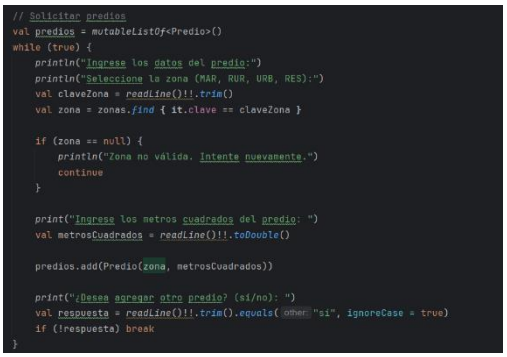
Línea de Código	Explicación
fun calcularImpuestoTotal(predios: List<Predio>, persona: Persona, mes: Int): Double {	Declara un método público calcularImpuestoTotal que recibe una lista de Predio, un objeto Persona y un valor entero mes. Retorna el impuesto total como un Double.
val total = predios.sumOf { predio ->	Calcula el total del impuesto sumando el impuesto de cada predio en la lista. La función sumOf suma los resultados de la operación especificada para cada predio.
val costoBase = predio.zona.costoPorMetro * predio.metrosCuadrados	Calcula el costoBase multiplicando el costoPorMetro de la zona del predio por los metrosCuadrados de dicho predio.
val descuento = calcularDescuento(persona, mes)	Llama al método calcularDescuento para obtener el porcentaje de descuento aplicable según la persona y el mes.
costoBase * (1 - descuento)	Aplica el descuento al costoBase, multiplicándolo por (1 - descuento).
}	Cierra el bloque de la expresión sumOf.
return total	Retorna el total del impuesto después de aplicar los descuentos.
}	Cierra el método calcularImpuestoTotal.
private fun calcularDescuento(persona: Persona, mes: Int): Double {	Declara un método privado calcularDescuento que recibe una Persona y el mes como parámetros y devuelve el descuento aplicable como un Double.
`val esTemporada = mes == 1	
val descuentoBase = when {	Declara descuentoBase y usa una expresión when para definir el porcentaje de descuento aplicable.
`persona.edad >= 70	
else -> if (esTemporada) 0.40 else 0.0	Si la persona no cumple los criterios anteriores, aplica un descuento del 40% en temporada, o ningún descuento (0.0) fuera de temporada.

Línea de Código	Explicación
}	Cierra la expresión when.
return descuentoBase	Retorna el valor de descuentoBase, que es el porcentaje de descuento calculado.
}	Cierra el método calcularDescuento.

código de la clase Main:



```kotlin
// Main.kt
fun main() {


    val zonas = listOf(
        Zona( clave: "MAR", nombre: "Marginado", costoPorMetro: 2.00),
        Zona( clave: "RUR", nombre: "Rural", costoPorMetro: 8.00),
        Zona( clave: "URB", nombre: "Urbana", costoPorMetro: 10.00),
        Zona( clave: "RES", nombre: "Residencial", costoPorMetro: 25.00)
    )

    // Solicitar datos de la persona
    print("Ingrese el nombre de la persona: ")
    val nombre = readLine()!!

    print("Ingrese la edad de la persona: ")
    val edad = readLine()!!.toInt()

    print("¿Es madre soltera? (si/no): ")
    val esMadreSoltera = readLine()!!.trim().equals( other: "si", ignoreCase = true)

    val persona = Persona(nombre, edad, esMadreSoltera)
```



```kotlin
// Solicitar predios
val predios = mutableListOf<Predio>()
while (true) {
    println("Ingrese los datos del predio:")
    println("Seleccione la zona (MAR, RUR, URB, RES):")
    val claveZona = readLine()!!.trim()
    val zona = zonas.find { it.clave == claveZona }

    if (zona == null) {
        println("Zona no válida. Intente nuevamente.")
        continue
    }

    print("Ingrese los metros cuadrados del predio: ")
    val metrosCuadrados = readLine()!!.toDouble()

    predios.add(Predio(zona, metrosCuadrados))

    print("¿Desea agregar otro predio? (si/no): ")
    val respuesta = readLine()!!.trim().equals( other: "si", ignoreCase = true)
    if (!respuesta) break
}
```

```kotlin
    // Solicitar mes de pago
    print("Ingrese el mes de pago (1 para enero, 2 para febrero, etc.): ")
    val mes = readLine()!!.toInt()

    // Calcular el impuesto
    val impuestoPredial = ImpuestoPredial()
    val total = impuestoPredial.calcularImpuestoTotal(predios, persona, mes)
    println("El total del impuesto predial a pagar es: \$${"%.2f".format(total)}")
}
```

Línea de Código	Explicación
fun main() {	Define la función principal main, punto de entrada de la aplicación en Kotlin.
val zonas = listOf(...	Crea una lista inmutable zonas con objetos Zona predefinidos, cada uno con su clave, nombre y costo por metro cuadrado.
Zona("MAR", "Marginado", 2.00), ... Zona("RES", "Residencial", 25.00)	Cada Zona representa una categoría con una clave (MAR, RUR, etc.), nombre y costo por metro cuadrado,

Línea de Código	Explicación
	usado más adelante para calcular impuestos.
print("Ingrese el nombre de la persona: ")	Imprime en pantalla el mensaje "Ingrese el nombre de la persona: ".
val nombre = readLine()!!	Lee el nombre de la persona desde la consola y lo asigna a la variable nombre. !! asegura que el valor no sea null.
print("Ingrese la edad de la persona: ")	Imprime "Ingrese la edad de la persona: " en pantalla para solicitar la edad.
val edad = readLine()!!.toInt()	Lee la entrada de edad desde la consola, la convierte a entero (Int) y la asigna a la variable edad.
print("¿Es madre soltera? (sí/no): ")	Solicita al usuario que indique si la persona es madre soltera.
val esMadreSoltera = readLine()!!.trim().equals("sí", ignoreCase = true)	Lee la entrada y la convierte a Boolean. Si la respuesta es "sí" (ignorando mayúsculas o minúsculas), asigna true a esMadreSoltera; de lo contrario, false.
val persona = Persona(nombre, edad, esMadreSoltera)	Crea una instancia de Persona con los datos ingresados (nombre, edad y esMadreSoltera).
val predios = mutableListOf<Predio>()	Declara una lista mutable predios para almacenar instancias de Predio.
while (true) {	Inicia un ciclo while que se ejecutará hasta que se decida no agregar más predios.
println("Ingrese los datos del predio:")	Imprime "Ingrese los datos del predio:" para solicitar la información de un predio.

Línea de Código	Explicación
println("Seleccione la zona (MAR, RUR, URB, RES):")	Solicita al usuario seleccionar una zona de las opciones disponibles (MAR, RUR, URB, RES).
val claveZona = readLine()!!.trim()	Lee la entrada de zona del usuario, eliminando espacios adicionales, y la asigna a claveZona.
val zona = zonas.find { it.clave == claveZona }	Busca en la lista zonas la zona cuya clave coincida con claveZona. Si no encuentra coincidencias, zona será null.
if (zona == null) { println("Zona no válida. Intente nuevamente."); continue }	Si zona es null, indica que la zona es inválida, muestra un mensaje y vuelve a solicitar los datos del predio.
print("Ingrese los metros cuadrados del predio: ")	Solicita al usuario ingresar el tamaño del predio en metros cuadrados.
val metrosCuadrados = readLine()!!.toDouble()	Lee la entrada del tamaño en metros cuadrados, la convierte a Double, y la asigna a metrosCuadrados.
predios.add(Predio(zona, metrosCuadrados))	Crea un objeto Predio con la zona seleccionada y metrosCuadrados, y lo agrega a la lista predios.
print("¿Desea agregar otro predio? (sí/no): ")	Pregunta al usuario si desea agregar otro predio.
val respuesta = readLine()!!.trim().equals("sí", ignoreCase = true)	Lee la respuesta y, si es "sí", la variable respuesta es true; si no, es false.
if (!respuesta) break	Si la respuesta es false (no desea agregar otro predio), se sale del ciclo while.
print("Ingrese el mes de pago (1 para enero, 2 para febrero, etc.): ")	Solicita al usuario ingresar el número del mes en que desea realizar el pago.
val mes = readLine()!!.toInt()	Lee la entrada del mes, la convierte en entero, y la asigna a la variable mes.

Línea de Código	Explicación
val impuestoPredial = ImpuestoPredial()	Crea una instancia de la clase ImpuestoPredial, que se usará para calcular el impuesto total.
val total = impuestoPredial.calcularImpuestoTotal(predios, persona, mes)	Llama al método calcularImpuestoTotal de ImpuestoPredial, pasando predios, persona, y mes como parámetros, y asigna el resultado a total.
println("El total del impuesto predial a pagar es: \$${"%.2f".format(total)}")	Imprime el total del impuesto predial a pagar, formateado a dos decimales (%.2f).
}	Cierra la función main.

2. Sistema de Gestión de nomina

Ejecución de la metodología especificada.

Propósito del Sistema

Este código complementa la clase Trabajador creando un sistema de gestión de nómina. El propósito principal es permitir la entrada de datos de múltiples trabajadores y calcular sus nóminas semanales basándose en:

- El nombre del trabajador.
- Su categoría, que determina la tarifa por hora.
- Las horas trabajadas y las horas extras.

Cómo Funciona

1. **Entrada de Datos**: Se solicita al usuario que ingrese los datos para varios trabajadores. El usuario puede salir escribiendo "salir".
2. **Cálculo de Nómina**: Para cada trabajador, se crea una instancia de la clase Trabajador, que ya tiene la lógica para calcular la nómina semanal basada en las horas trabajadas y extras.
3. **Salida de Resultados**: Al final, el sistema muestra la nómina semanal de todos los trabajadores ingresados, proporcionando un resumen claro y conciso.

Código de instrucciones ejecutadas por cada sentencia utilizada.

código de la clase Trabajador:

```
class Trabajador(
    val nombre: String,
    val categoria: Int,
    val horasTrabajo: Int,
    val horasExtra: Int

)
{
     val PRECIO_HORA_BASE = 90
    // Función para calcular el precio por hora según la categoría
    private fun calcularPrecioHora(): Double {
        return when (categoria) {
            1 -> PRECIO_HORA_BASE * 1.45
            2 -> PRECIO_HORA_BASE * 1.25
            else -> PRECIO_HORA_BASE.toDouble()
        }
    }

    // Función para calcular el precio de las horas extras
    private fun calcularPrecioHoraExtra(): Double {
        return when {
            horasExtra < 10 -> calcularPrecioHora() * 1.5
            horasExtra in 10 ≤ .. ≤ 20 -> calcularPrecioHora() * 1.4
            else -> calcularPrecioHora() * 1.2
        }
    }
```

```
    // Función para calcular la nómina semanal
    fun calcularNominaSemanal(): Double {
        val sueldoBase = horasTrabajo * calcularPrecioHora()
        val sueldoExtra = horasExtra * calcularPrecioHoraExtra()
        return sueldoBase + sueldoExtra
    }

    // Representación del trabajador
    override fun toString(): String {
        return "Nombre: $nombre, Categoría: $categoria, Horas Trabajo: $horasTrabajo, Horas Extra: $horasExtra, Nómina Semanal: ${calcularNominaSemanal()}"
    }
}
```

Línea de Código	**Explicación**
class Trabajador(...	Define la clase Trabajador con propiedades (nombre, categoria, horasTrabajo, horasExtra) y métodos para calcular el salario de un trabajador.
val nombre: String, val categoria: Int, val horasTrabajo: Int, val horasExtra: Int	Declara las propiedades del trabajador: nombre, categoría laboral, horas trabajadas y horas extra. Estas propiedades se reciben como parámetros en el constructor.
{ val PRECIO_HORA_BASE = 90	Define una constante PRECIO_HORA_BASE, que representa el salario base por hora para los cálculos de nómina.
private fun calcularPrecioHora(): Double {	Define una función privada calcularPrecioHora para calcular el salario por hora según la categoría del trabajador.

Línea de Código	**Explicación**
return when (categoria) { 1 -> PRECIO_HORA_BASE * 1.45 ... else -> PRECIO_HORA_BASE.toDouble() }	Utiliza una expresión when para ajustar el PRECIO_HORA_BASE dependiendo de la categoría: categoría 1 aumenta el salario un 45 %, categoría 2 un 25 %, y para otras categorías se mantiene el salario base.
private fun calcularPrecioHoraExtra(): Double {	Define la función calcularPrecioHoraExtra para calcular la tarifa por hora extra, aplicando distintos incrementos según el número de horas extra.
return when { horasExtra < 10 -> calcularPrecioHora() * 1.5 ... else -> calcularPrecioHora() * 1.2 }	Usa otra expresión when para determinar el salario por hora extra: incrementos de 50 %, 40 % o 20 % según si las horas extra son menores a 10, entre 10 y 20, o más de 20, respectivamente.
fun calcularNominaSemanal(): Double {	Define la función calcularNominaSemanal para calcular el salario semanal del trabajador.
val sueldoBase = horasTrabajo * calcularPrecioHora()	Calcula el salario base multiplicando las horas trabajadas (horasTrabajo) por el salario por hora de la categoría (calcularPrecioHora()).
val sueldoExtra = horasExtra * calcularPrecioHoraExtra()	Calcula el salario de las horas extra multiplicando las horas extra (horasExtra) por el salario por hora extra (calcularPrecioHoraExtra()).
return sueldoBase + sueldoExtra	Devuelve la suma de sueldoBase y sueldoExtra, representando la nómina semanal total del trabajador.
override fun toString(): String {	Sobrescribe la función toString para devolver una representación en texto de los datos del trabajador, incluyendo el nombre, categoría, horas trabajadas, horas extra, y el cálculo de la nómina semanal.
return "Nombre: $nombre, Categoría: $categoria, Horas Trabajo: ...	Retorna una cadena con la información del trabajador y el resultado de calcularNominaSemanal(), proporcionando una vista detallada del trabajador y su salario semanal.
}	Cierra la clase Trabajador.

código de la clase Main:

```
fun main() {
    // Mapa para almacenar los trabajadores
    val trabajadores = mutableMapOf<String, Trabajador>()

    // Pedir datos al usuario
    while (true) {
        println("Ingrese el nombre del trabajador (o 'salir' para terminar): ")
        val nombre = readLine() ?: break
        if (nombre.lowercase() == "salir") break

        println("Ingrese la categoría del trabajador (1, 2, 3): ")
        val categoria = readLine()?.toIntOrNull() ?: continue

        println("Ingrese las horas trabajadas: ")
        val horasTrabajo = readLine()?.toIntOrNull() ?: continue

        println("Ingrese las horas extras trabajadas: ")
        val horasExtra = readLine()?.toIntOrNull() ?: continue

        // Crear un trabajador y agregarlo al mapa
        val trabajador = Trabajador(nombre, categoria, horasTrabajo, horasExtra)
        trabajadores[nombre] = trabajador
    }

    // Mostrar las nóminas de todos los trabajadores
    println("\nNóminas de los trabajadores:")
    trabajadores.forEach { (_, trabajador) ->
        println(trabajador)
    }
}
```

Línea de Código	Explicación
fun main() {	Define la función principal main, que es el punto de entrada de la aplicación.
val trabajadores = mutableMapOf<String, Trabajador>()	Crea un mapa mutable (mutableMapOf) para almacenar objetos de la clase Trabajador, utilizando el nombre del trabajador como clave y el objeto Trabajador como valor.
while (true) {	Inicia un bucle infinito para solicitar datos del usuario hasta que se indique que se desea salir.

Línea de Código	**Explicación**
println("Ingrese el nombre del trabajador (o 'salir' para terminar): ")	Muestra un mensaje en la consola pidiendo al usuario que ingrese el nombre del trabajador.
val nombre = readLine() ?: break	Lee la entrada del usuario y asigna el valor a nombre. Si la entrada es nula (por ejemplo, si ocurre un error al leer), se rompe el bucle.
if (nombre.lowercase() == "salir") break	Comprueba si el nombre ingresado es "salir" (ignorando mayúsculas), y si es así, se rompe el bucle para terminar la entrada de datos.
println("Ingrese la categoría del trabajador (1, 2, 3): ")	Solicita al usuario que ingrese la categoría del trabajador.
val categoria = readLine()?.toIntOrNull() ?: continue	Lee la entrada del usuario, intenta convertirla a un entero. Si la conversión falla (por ejemplo, si el usuario ingresa algo que no es un número), se continúa al siguiente ciclo del bucle, solicitando nuevamente el nombre del trabajador.
println("Ingrese las horas trabajadas: ")	Pide al usuario que ingrese las horas trabajadas por el trabajador.
val horasTrabajo = readLine()?.toIntOrNull() ?: continue	Lee la entrada del usuario y convierte el valor a un entero. Si la conversión falla, se continúa al siguiente ciclo del bucle.
println("Ingrese las horas extras trabajadas: ")	Solicita al usuario que ingrese las horas extras trabajadas.
val horasExtra = readLine()?.toIntOrNull() ?: continue	Lee la entrada y la convierte a un entero. Si no es un número válido, se continúa al siguiente ciclo del bucle.
val trabajador = Trabajador(nombre, categoria, horasTrabajo, horasExtra)	Crea una nueva instancia de Trabajador utilizando los datos ingresados por el usuario y la almacena en la variable trabajador.
trabajadores[nombre] = trabajador	Agrega el nuevo objeto Trabajador al mapa trabajadores, utilizando el nombre como clave.
}	Cierra el bucle while.

Línea de Código	Explicación
println("\nNóminas de los trabajadores:")	Imprime un encabezado para mostrar la nómina de los trabajadores después de que se hayan ingresado todos los datos.
trabajadores.forEach { (_, trabajador) ->	Inicia un bucle para recorrer cada entrada en el mapa trabajadores, donde trabajador representa cada objeto Trabajador en el mapa.
println(trabajador)	Imprime la representación en cadena del objeto Trabajador, que incluye su nombre, categoría, horas trabajadas, horas extra y nómina semanal calculada.
}	Cierra el bucle forEach.
}	Cierra la función main.

3. Sistema de Nomina.

Ejecución de la metodología especificada.

Este sistema está diseñado para gestionar la nómina de empleados de una empresa. Su propósito principal es permitir la creación de diferentes tipos de empleados (fijos, por horas y por comisión), calcular sus salarios de acuerdo a sus características y, finalmente, generar una nómina mensual que muestre los salarios correspondientes a cada empleado.

Funciones del Sistema:

1. **Gestión de Empleados**:
    - Permite al usuario ingresar los datos de varios empleados, incluyendo su nombre, ID, tipo de empleado, salario base y detalles específicos según su tipo (tarifa por hora para empleados por horas, porcentaje de comisión y ventas generadas para empleados por comisión).
2. **Cálculo de Salarios**:
    - El sistema incluye clases específicas para cada tipo de empleado que heredan de una clase base Empleado. Cada clase implementa el método calcularSalario() de manera diferente, asegurando que se calculen los salarios según las reglas correspondientes:
        - **Empleado Fijo**: Recibe un salario base fijo.
        - **Empleado por Horas**: Calcula su salario en función de las horas trabajadas y las horas extras.

- **Empleado por Comisión**: Calcula su salario sumando un porcentaje de sus ventas generadas a su salario base.

3. **Generación de Nómina**:
   - Una vez ingresados los empleados, el sistema puede generar una nómina mensual que calcula el salario total de cada empleado y lo presenta de manera clara y organizada. Esto permite a la empresa conocer el total a pagar a cada empleado y facilita la planificación financiera.
4. **Interactividad**:
   - El sistema es interactivo y solicita información al usuario a través de la consola. Esto permite una fácil personalización y entrada de datos sin necesidad de una interfaz gráfica compleja.
5. **Validaciones**:
   - Incluye validaciones básicas, como convertir entradas de texto a tipos de datos apropiados (enteros y dobles) y manejar entradas nulas, lo que contribuye a la robustez del sistema.

Código de instrucciones ejecutadas por cada sentencia utilizada.

código de la clase Empleado:

```
abstract class Empleado(val nombre: String, val id: Int, val salarioBase: Double) {
    abstract fun calcularSalario(): Double
}
```

Línea de Código	Explicación
abstract class Empleado(val nombre: String, val id: Int, val salarioBase: Double) {	Define una clase abstracta llamada Empleado con tres propiedades: nombre, id (identificador del empleado) y salarioBase. Estas propiedades son parámetros del constructor.
abstract fun calcularSalario(): Double	Declara un método abstracto calcularSalario que debe ser implementado por las subclases de Empleado. Este método retornará un valor de tipo Double, representando el salario calculado.
}	Cierra la definición de la clase Empleado.

código de la clase EmpleadoFijo:

```
class EmpleadoFijo(nombre: String, id: Int, salarioFijo: Double) : Empleado(nombre, id, salarioFijo) {
    override fun calcularSalario(): Double {
        return salarioBase
    }
}
```

Línea de Código	Explicación
class EmpleadoFijo(nombre: String, id: Int, salarioFijo: Double) : Empleado(nombre, id, salarioFijo) {	Define una clase llamada EmpleadoFijo que hereda de la clase abstracta Empleado. Recibe tres parámetros: nombre, id y salarioFijo, que son pasados al constructor de la clase base.
override fun calcularSalario(): Double {	Comienza la implementación del método calcularSalario, que se sobrescribe (override) para definir cómo se calcula el salario para un empleado fijo.
return salarioBase	Retorna el salarioBase del empleado fijo. Dado que el salario fijo no cambia, el método simplemente devuelve este valor.
}	Cierra la definición del método calcularSalario.
}	Cierra la definición de la clase EmpleadoFijo.

código de la clase EmpleadoPorComision:

```
class EmpleadoPorComision(
    nombre: String,
    id: Int,
    salarioBase: Double,
    private val porcentajeComision: Double,
    private val ventasGeneradas: Double
) : Empleado(nombre, id, salarioBase) {

    override fun calcularSalario(): Double {
        return salarioBase + (porcentajeComision / 100) * ventasGeneradas
    }
}
```

Línea de Código	Explicación
class EmpleadoPorComision(	Define una clase llamada EmpleadoPorComision que hereda de la clase abstracta Empleado.
nombre: String,	Declara el parámetro nombre que se pasará al constructor.
id: Int,	Declara el parámetro id que se pasará al constructor.
salarioBase: Double,	Declara el parámetro salarioBase que se pasará al constructor.
private val porcentajeComision: Double,	Declara un parámetro privado porcentajeComision, que representa el porcentaje de comisión que el empleado recibe por sus ventas.
private val ventasGeneradas: Double	Declara un parámetro privado ventasGeneradas, que representa el total de ventas generadas por el empleado.
) : Empleado(nombre, id, salarioBase) {	Llama al constructor de la clase base Empleado, pasando los parámetros nombre, id y salarioBase.
override fun calcularSalario(): Double {	Comienza la implementación del método calcularSalario, que se sobrescribe para definir cómo se calcula el salario de un empleado por comisión.
return salarioBase + (porcentajeComision / 100) * ventasGeneradas	Retorna el salario base más la comisión calculada como un porcentaje de las ventas generadas. Este es el cálculo del salario total del empleado por comisión.
}	Cierra la definición del método calcularSalario.
}	Cierra la definición de la clase EmpleadoPorComision.

código de la clase EmpleadoPorHoras:

```
class EmpleadoPorHoras(
    nombre: String,
    id: Int,
    salarioBase: Double,
    private val tarifaPorHora: Double,
    private val horasTrabajadas: Double,
    private val horasExtras: Double = 0.0, // Horas extras si las hay
    private val tarifaHorasExtras: Double = 1.5 // 50% más por hora extra
) : Empleado(nombre, id, salarioBase) {

    override fun calcularSalario(): Double {
        val salarioHoras = tarifaPorHora * horasTrabajadas
        val salarioHorasExtras = tarifaHorasExtras * tarifaPorHora * horasExtras
        return salarioBase + salarioHoras + salarioHorasExtras
    }
}
```

Línea de Código	Explicación
class EmpleadoPorHoras(	Define una clase llamada EmpleadoPorHoras que hereda de la clase abstracta Empleado.
nombre: String,	Declara el parámetro nombre que se pasará al constructor.
id: Int,	Declara el parámetro id que se pasará al constructor.
salarioBase: Double,	Declara el parámetro salarioBase que se pasará al constructor.
private val tarifaPorHora: Double,	Declara un parámetro privado tarifaPorHora, que representa el pago por cada hora trabajada por el empleado.
private val horasTrabajadas: Double,	Declara un parámetro privado horasTrabajadas, que representa el total de horas trabajadas por el empleado.
private val horasExtras: Double = 0.0,	Declara un parámetro privado horasExtras (con valor predeterminado de 0.0) para manejar horas extras si las hay.

Línea de Código	Explicación
private val tarifaHorasExtras: Double = 1.5	Declara un parámetro privado tarifaHorasExtras, que representa el multiplicador (50% más) que se aplica a la tarifa por hora para calcular el pago de horas extras.
) : Empleado(nombre, id, salarioBase) {	Llama al constructor de la clase base Empleado, pasando los parámetros nombre, id y salarioBase.
override fun calcularSalario(): Double {	Comienza la implementación del método calcularSalario, que se sobrescribe para definir cómo se calcula el salario de un empleado por horas.
val salarioHoras = tarifaPorHora * horasTrabajadas	Calcula el salario correspondiente a las horas trabajadas multiplicando la tarifa por hora por el número de horas trabajadas y lo almacena en la variable salarioHoras.
val salarioHorasExtras = tarifaHorasExtras * tarifaPorHora * horasExtras	Calcula el salario correspondiente a las horas extras multiplicando la tarifa de horas extras, la tarifa por hora y el número de horas extras trabajadas.
return salarioBase + salarioHoras + salarioHorasExtras	Retorna la suma del salario base, el salario por horas trabajadas y el salario por horas extras, dando como resultado el salario total del empleado por horas.
}	Cierra la definición del método calcularSalario.
}	Cierra la definición de la clase EmpleadoPorHoras.

Código de la clase Nomina:

```
class Nomina(private val empleados: List<Empleado>) {
    fun generarNominaMensual(): Map<Int, Double> {
        val nomina = mutableMapOf<Int, Double>()
        for (empleado in empleados) {
            nomina[empleado.id] = empleado.calcularSalario()
        }
        return nomina
    }
}
```

Línea de Código	Explicación
class Nomina(private val empleados: List<Empleado>) {	Define una clase llamada Nomina, que recibe una lista de objetos de tipo Empleado en su constructor. La lista se almacena en la propiedad privada empleados.
fun generarNominaMensual(): Map<Int, Double> {	Declara una función llamada generarNominaMensual que retorna un mapa (Map) donde las claves son enteros (IDs de empleados) y los valores son dobles (salarios calculados).
val nomina = mutableMapOf<Int, Double>()	Crea un mapa mutable llamado nomina para almacenar los salarios de los empleados, donde las claves son los IDs de los empleados y los valores son sus salarios.
for (empleado in empleados) {	Inicia un bucle que itera sobre cada objeto empleado en la lista empleados.
nomina[empleado.id] = empleado.calcularSalario()	Llama al método calcularSalario() del objeto empleado y almacena el resultado en el mapa nomina usando el ID del empleado como clave.
}	Cierra el bloque del bucle for.
return nomina	Retorna el mapa nomina que contiene los IDs de los empleados y sus respectivos salarios calculados.
}	Cierra la definición de la clase Nomina.

código de la clase Main:

```
// Funciones auxiliares y la función main pueden estar en el mismo archivo
fun solicitarDatosEmpleado(): Empleado {
    println("Ingrese el nombre del empleado:")
    val nombre = readLine() ?: ""

    println("Ingrese el ID del empleado:")
    val id = readLine()?.toIntOrNull() ?: 0

    println("Ingrese el tipo de empleado (fijo, porHoras, porComision):")
    val tipo = readLine() ?: ""

    println("Ingrese el salario base del empleado:")
    val salarioBase = readLine()?.toDoubleOrNull() ?: 0.0

    return when (tipo) {
        "fijo" -> EmpleadoFijo(nombre, id, salarioBase)
        "porHoras" -> {
            println("Ingrese la tarifa por hora:")
            val tarifaPorHora = readLine()?.toDoubleOrNull() ?: 0.0
            println("Ingrese las horas trabajadas:")
            val horasTrabajadas = readLine()?.toDoubleOrNull() ?: 0.0
            println("Ingrese las horas extras (opcional, 0 si no aplica):")
            val horasExtras = readLine()?.toDoubleOrNull() ?: 0.0
            EmpleadoPorHoras(nombre, id, salarioBase, tarifaPorHora, horasTrabajadas, horasExtras)
        }
        "porComision" -> {
            println("Ingrese el porcentaje de comisión:")
            val porcentajeComision = readLine()?.toDoubleOrNull() ?: 0.0
            println("Ingrese las ventas generadas:")
            val ventasGeneradas = readLine()?.toDoubleOrNull() ?: 0.0
            EmpleadoPorComision(nombre, id, salarioBase, porcentajeComision, ventasGeneradas)
        }
```

```
        else -> throw IllegalArgumentException("Tipo de empleado no válido")
    }
}

fun main() {
    println("¿Cuántos empleados desea ingresar?")
    val cantidad = readLine()?.toIntOrNull() ?: 0

    val empleados = mutableListOf<Empleado>()

    for (i in 1 ≤ .. ≤ cantidad) {
        println("Ingrese los datos del empleado $i:")
        val empleado = solicitarDatosEmpleado()
        empleados.add(empleado)
    }

    val nomina = Nomina(empleados)
    val resultados = nomina.generarNominaMensual()

    println("Nómina mensual:")
    for ((id, salario) in resultados) {
        println("Empleado ID: $id, Salario: $salario")
    }
}
```

Línea de Código	Explicación
fun solicitarDatosEmpleado(): Empleado {	Declara una función llamada solicitarDatosEmpleado que retorna un objeto de tipo Empleado.
println("Ingrese el nombre del empleado:")	Imprime un mensaje solicitando el nombre del empleado.
val nombre = readLine() ?: ""	Lee la entrada del usuario y asigna el nombre a la variable nombre. Si la entrada es nula, se asigna una cadena vacía.
println("Ingrese el ID del empleado:")	Imprime un mensaje solicitando el ID del empleado.
val id = readLine()?.toIntOrNull() ?: 0	Lee la entrada del usuario y convierte la cadena a un entero. Si la conversión falla, se asigna 0 como ID.
println("Ingrese el tipo de empleado (fijo, porHoras, porComision):")	Imprime un mensaje solicitando el tipo de empleado.
val tipo = readLine() ?: ""	Lee la entrada del usuario y asigna el tipo de empleado a la variable tipo. Si la entrada es nula, se asigna una cadena vacía.
println("Ingrese el salario base del empleado:")	Imprime un mensaje solicitando el salario base del empleado.
val salarioBase = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como salario base.
return when (tipo) {	Inicia un bloque when para determinar el tipo de empleado.
"fijo" -> EmpleadoFijo(nombre, id, salarioBase)	Si el tipo es "fijo", se crea y retorna un objeto EmpleadoFijo con el nombre, ID y salario base.
"porHoras" -> {	Si el tipo es "porHoras", se inicia un bloque para solicitar más datos.
println("Ingrese la tarifa por hora:")	Imprime un mensaje solicitando la tarifa por hora.

Línea de Código	Explicación
val tarifaPorHora = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como tarifa por hora.
println("Ingrese las horas trabajadas:")	Imprime un mensaje solicitando las horas trabajadas.
val horasTrabajadas = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como horas trabajadas.
println("Ingrese las horas extras (opcional, 0 si no aplica):")	Imprime un mensaje solicitando las horas extras, indicando que se puede ingresar 0 si no aplica.
val horasExtras = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como horas extras.
EmpleadoPorHoras(nombre, id, salarioBase, tarifaPorHora, horasTrabajadas, horasExtras)	Se crea y retorna un objeto EmpleadoPorHoras con el nombre, ID, salario base, tarifa por hora, horas trabajadas y horas extras.
}	Cierra el bloque para el tipo "porHoras".
"porComision" -> {	Si el tipo es "porComision", se inicia un bloque para solicitar más datos.
println("Ingrese el porcentaje de comisión:")	Imprime un mensaje solicitando el porcentaje de comisión.
val porcentajeComision = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como porcentaje de comisión.
println("Ingrese las ventas generadas:")	Imprime un mensaje solicitando las ventas generadas.
val ventasGeneradas = readLine()?.toDoubleOrNull() ?: 0.0	Lee la entrada del usuario y convierte la cadena a un doble. Si la conversión falla, se asigna 0.0 como ventas generadas.
EmpleadoPorComision(nombre, id, salarioBase, porcentajeComision, ventasGeneradas)	Se crea y retorna un objeto EmpleadoPorComision con el nombre, ID,

Línea de Código	Explicación
	salario base, porcentaje de comisión y ventas generadas.
}	Cierra el bloque para el tipo "porComision".
else -> throw IllegalArgumentException("Tipo de empleado no válido")	Si el tipo no coincide con ninguno de los anteriores, se lanza una excepción indicando que el tipo de empleado no es válido.
}	Cierra el bloque when.
}	Cierra la función solicitarDatosEmpleado.
fun main() {	Declara la función principal main donde se ejecutará el programa.
println("¿Cuántos empleados desea ingresar?")	Imprime un mensaje solicitando la cantidad de empleados que se desean ingresar.
val cantidad = readLine()?.toIntOrNull() ?: 0	Lee la entrada del usuario y convierte la cadena a un entero. Si la conversión falla, se asigna 0 como cantidad.
val empleados = mutableListOf<Empleado>()	Crea una lista mutable para almacenar los empleados que se ingresarán.
for (i in 1..cantidad) {	Inicia un bucle que itera desde 1 hasta la cantidad de empleados que el usuario quiere ingresar.
println("Ingrese los datos del empleado $i:")	Imprime un mensaje indicando el número del empleado que se está ingresando.
val empleado = solicitarDatosEmpleado()	Llama a la función solicitarDatosEmpleado para obtener un objeto Empleado y lo asigna a la variable empleado.
empleados.add(empleado)	Agrega el objeto empleado a la lista de empleados.
}	Cierra el bucle for.
val nomina = Nomina(empleados)	Crea un objeto Nomina pasando la lista de empleados.

Línea de Código	Explicación
val resultados = nomina.generarNominaMensual()	Llama al método generarNominaMensual del objeto nomina para calcular los salarios de los empleados y almacena el resultado en resultados.
println("Nómina mensual:")	Imprime un mensaje encabezando la sección de resultados de la nómina mensual.
for ((id, salario) in resultados) {	Inicia un bucle que itera sobre cada entrada del mapa resultados donde id es el ID del empleado y salario es su salario.
println("Empleado ID: $id, Salario: $salario")	Imprime el ID del empleado y su salario correspondiente.
}	Cierra el bucle for.
}	Cierra la función main.

4. Sistema de Gestión de Vuelos.

Ejecución de la metodología especificada.

La función del sistema es gestionar reservas de vuelos, permitiendo a los usuarios crear tanto reservas individuales como agrupadas. Aquí hay un desglose detallado de cómo funciona el sistema:

1. **Entrada de Datos:**
    - El sistema solicita al usuario que ingrese información necesaria para las reservas, incluyendo:
        - Descuentos para reservas grupales.
        - Detalles sobre pasajeros, incluyendo nombre, número de vuelo, precio base y tipo de tarifa (económica o ejecutiva).
2. **Reservas Individuales:**
    - Cada reserva individual se crea a partir de la clase ReservaIndividual, que almacena información específica del pasajero y calcula el costo basado en la tarifa seleccionada y el precio base.
3. **Reservas Grupales:**

- Las reservas individuales se pueden agregar a una reserva grupal, que es gestionada por la clase ReservaGrupal. Esta clase permite agregar múltiples reservas individuales y calcular un costo total con un descuento aplicado.

4. **Reservas Grupales Anidadas:**
    - El sistema permite anidar reservas grupales dentro de otras reservas grupales. Esto significa que un grupo puede hacer reservas adicionales, cada una con sus propios descuentos y reservas individuales.
5. **Cálculo del Costo Total:**
    - La clase ReservaGrupal calcula el costo total de todas las reservas individuales y grupales anidadas, aplicando el descuento correspondiente a la reserva grupal principal.
6. **Salida de Datos:**
    - Al final, el sistema imprime los detalles de todas las reservas (tanto individuales como grupales) y muestra el costo total de la reserva grupal.

El sistema permite una gestión flexible de reservas de vuelos, donde se pueden crear, combinar y calcular costos de forma estructurada. Facilita a los usuarios la entrada de datos y les proporciona un resumen claro de los detalles de sus reservas y el costo total a pagar. Esta funcionalidad es especialmente útil para grupos que desean hacer reservas colectivas y beneficiarse de descuentos.

Código de instrucciones ejecutadas por cada sentencia utilizada.

código de la clase Reserva:

```
// Interfaz Reserva
interface Reserva {
    fun calcularCosto(): Double
}
```

Código	Explicación
interface Reserva {	Declara una interfaz llamada Reserva. Las interfaces son contratos que las clases pueden implementar.
fun calcularCosto(): Double	Define un método abstracto calcularCosto, que debe ser implementado por cualquier clase que implemente la interfaz Reserva. Este método retorna un valor de tipo Double, que representa el costo de la reserva.

código de la clase ReservaGrupal:

```
class ReservaGrupal(private val descuento: Double) : Reserva {
    private val reservas: MutableList<Reserva> = mutableListOf()

    fun agregarReserva(reserva: Reserva) {
        reservas.add(reserva)
    }

    override fun calcularCosto(): Double {
        val costoTotal = reservas.sumOf { it.calcularCosto() }
        return costoTotal * (1 - descuento)
    }

    fun detalles(): String {
        return reservas.joinToString( separator: "\n") {
            if (it is ReservaIndividual) it.detalles() else ""
        }
    }
}
```

Código	**Explicación**
class ReservaGrupal(private val descuento: Double) : Reserva {	Declara una clase ReservaGrupal que implementa la interfaz Reserva. Recibe un parámetro descuento de tipo Double en su constructor.
private val reservas: MutableList<Reserva> = mutableListOf()	Declara una propiedad privada reservas, que es una lista mutable de objetos que implementan la interfaz Reserva. Inicializa esta lista como vacía.
fun agregarReserva(reserva: Reserva) {	Declara una función pública agregarReserva que toma un parámetro reserva de tipo Reserva.
reservas.add(reserva)	Añade la reserva proporcionada a la lista reservas.
override fun calcularCosto(): Double {	Inicia la implementación del método calcularCosto de la interfaz Reserva, que debe calcular el costo total de las reservas en grupo.
val costoTotal = reservas.sumOf { it.calcularCosto() }	Calcula el costo total llamando al método calcularCosto de cada reserva en la lista reservas y sumando los resultados.
return costoTotal * (1 - descuento)	Retorna el costo total aplicando el descuento especificado.
fun detalles(): String {	Declara una función pública detalles que retorna una cadena de texto.
return reservas.joinToString("\n") {	Utiliza el método joinToString para concatenar los detalles de las reservas en la lista reservas, separando cada detalle por un salto de línea.
if (it is ReservaIndividual) it.detalles() else ""	Para cada reserva, verifica si es de tipo ReservaIndividual. Si es así, llama al método detalles() de esa reserva; de lo contrario, retorna una cadena vacía.

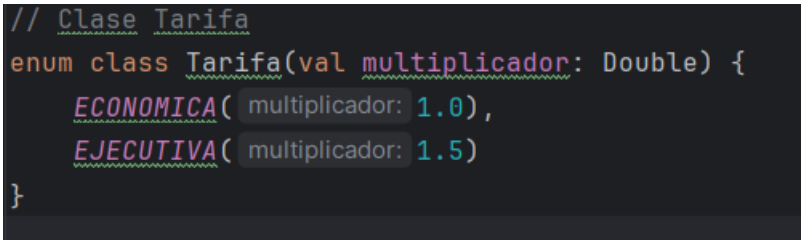
código de la clase ReservaIndividual:

```
class ReservaIndividual(
    private val tarifa: Tarifa,
    private val precioBase: Double,
    private val nombrePasajero: String,
    private val numeroVuelo: String
) : Reserva {
    override fun calcularCosto(): Double {
        return precioBase * tarifa.multiplicador
    }

    fun detalles(): String {
        return "Pasajero: $nombrePasajero, Vuelo: $numeroVuelo, Costo: ${calcularCosto()}"
    }
}
```

Código	Explicación
class ReservaIndividual(	Declara una clase ReservaIndividual que implementa la interfaz Reserva.
private val tarifa: Tarifa,	Declara una propiedad privada tarifa de tipo Tarifa, que se espera que contenga información sobre las tarifas aplicables a la reserva individual.
private val precioBase: Double,	Declara una propiedad privada precioBase de tipo Double, que representa el costo base de la reserva individual.
private val nombrePasajero: String,	Declara una propiedad privada nombrePasajero de tipo String, que almacena el nombre del pasajero que realiza la reserva.
private val numeroVuelo: String	Declara una propiedad privada numeroVuelo de tipo String, que almacena el número de vuelo asociado a la reserva.
) : Reserva {	Cierra la declaración de constructor y especifica que la clase implementa la interfaz Reserva.
override fun calcularCosto(): Double {	Inicia la implementación del método calcularCosto de la interfaz Reserva, que calcula el costo de la reserva individual.

Código	Explicación
return precioBase * tarifa.multiplicador	Retorna el costo total de la reserva multiplicando el precioBase por un valor multiplicador de la propiedad tarifa.
fun detalles(): String {	Declara una función pública detalles que retorna una cadena de texto con información sobre la reserva individual.
return "Pasajero: $nombrePasajero, Vuelo: $numeroVuelo, Costo: ${calcularCosto()}"	Retorna una cadena formateada que incluye el nombre del pasajero, el número de vuelo y el costo calculado de la reserva.

código de la clase Tarifa:



Código	Explicación
enum class Tarifa(	Declara una enumeración llamada Tarifa. Las enumeraciones en Kotlin son un tipo especial de clase que representa un grupo de constantes.
val multiplicador: Double)	Declara una propiedad multiplicador de tipo Double, que se usará para definir el multiplicador asociado a cada tarifa.
ECONOMICA(1.0),	Define un valor de enumeración ECONOMICA con un multiplicador de 1.0, representando la tarifa económica.
EJECUTIVA(1.5)	Define un valor de enumeración EJECUTIVA con un multiplicador de 1.5, representando la tarifa ejecutiva.
}	Cierra la declaración de la enumeración.

código de la clase Main:

```
fun leerTarifa(): Tarifa {
    println("Ingrese la tarifa (1 para Económica, 2 para Ejecutiva):")
    return when (readLine()?.trim()?.toInt()) {
        1 -> Tarifa.ECONOMICA
        2 -> Tarifa.EJECUTIVA
        else -> {
            println("Opción inválida. Usando tarifa económica por defecto.")
            Tarifa.ECONOMICA
        }
    }
}

fun leerDescuento(): Double {
    println("Ingrese el descuento (0.0 a 1.0):")
    return readLine()?.trim()?.toDoubleOrNull() ?: 0.0
}

fun leerDatosReservaIndividual(): ReservaIndividual {
    println("Ingrese el nombre del pasajero:")
    val nombrePasajero = readLine()?.trim() ?: "Desconocido"

    println("Ingrese el número de vuelo:")
    val numeroVuelo = readLine()?.trim() ?: "Desconocido"

    println("Ingrese el precio base para la reserva individual:")
    val precioBase = readLine()?.trim()?.toDoubleOrNull() ?: 0.0

    val tarifa = leerTarifa()
    return ReservaIndividual(tarifa, precioBase, nombrePasajero, numeroVuelo)
}
```

```
fun main() {
    println("Ingrese el descuento para la reserva grupal (0.0 a 1.0):")
    val descuentoGrupal = leerDescuento()
    val reservaGrupal = ReservaGrupal(descuentoGrupal)

    println("Ingrese el número de reservas individuales:")
    val numReservas = readLine()?.trim()?.toIntOrNull() ?: 0

    repeat(numReservas) { index ->
        val reservaIndividual = leerDatosReservaIndividual()
        reservaGrupal.agregarReserva(reservaIndividual)
    }

    println("Ingrese el número de reservas grupales anidadas (0 si no hay):")
    val numReservasGrupales = readLine()?.trim()?.toIntOrNull() ?: 0

    repeat(numReservasGrupales) { index ->
        println("Ingrese el descuento para la reserva grupal anidada ${index + 1}:")
        val descuentoAnidada = leerDescuento()

        val reservaGrupalAnidada = ReservaGrupal(descuentoAnidada)

        println("Ingrese el número de reservas individuales en la reserva grupal anidada ${index + 1}:")
        val numReservasAnidadas = readLine()?.trim()?.toIntOrNull() ?: 0

        repeat(numReservasAnidadas) { innerIndex ->
            val reservaIndividualAnidada = leerDatosReservaIndividual()
            reservaGrupalAnidada.agregarReserva(reservaIndividualAnidada)
        }
```

```
        reservaGrupal.agregarReserva(reservaGrupalAnidada)
    }

    println("Detalles de la reserva grupal:")
    println(reservaGrupal.detalles())
    println("Costo total de la reserva grupal: ${reservaGrupal.calcularCosto()}")
}
```

Código	Explicación
fun leerTarifa(): Tarifa {	Declara una función llamada leerTarifa que devuelve un valor de tipo Tarifa.
println("Ingrese la tarifa (1 para Económica, 2 para Ejecutiva):")	Muestra un mensaje al usuario para que ingrese el tipo de tarifa que desea (económica o ejecutiva).
return when (readLine()?.trim()?.toInt()) {	Usa una expresión when para evaluar el valor ingresado por el usuario y convertirlo a un número entero.

Código	Explicación
1 -> Tarifa.ECONOMICA	Si el usuario ingresa 1, devuelve el valor Tarifa.ECONOMICA.
2 -> Tarifa.EJECUTIVA	Si el usuario ingresa 2, devuelve el valor Tarifa.EJECUTIVA.
else -> {	Si el valor ingresado no es 1 ni 2, se ejecuta la opción else.
println("Opción inválida. Usando tarifa económica por defecto.")	Imprime un mensaje indicando que la opción ingresada es inválida.
Tarifa.ECONOMICA	Devuelve Tarifa.ECONOMICA como opción por defecto.
}	Cierra la función leerTarifa.
fun leerDescuento(): Double {	Declara una función llamada leerDescuento que devuelve un valor de tipo Double.
println("Ingrese el descuento (0.0 a 1.0):")	Solicita al usuario que ingrese el descuento en un rango de 0.0 a 1.0.
return readLine()?.trim()?.toDoubleOrNull() ?: 0.0	Devuelve el valor ingresado como Double, o 0.0 si el usuario no ingresa un valor válido.
fun leerDatosReservaIndividual(): ReservaIndividual {	Declara una función que devuelve un objeto de tipo ReservaIndividual.
println("Ingrese el nombre del pasajero:")	Solicita al usuario que ingrese el nombre del pasajero.

Código	Explicación
val nombrePasajero = readLine()?.trim() ?: "Desconocido"	Lee el nombre ingresado y, si es nulo, establece un valor predeterminado de "Desconocido".
println("Ingrese el número de vuelo:")	Solicita al usuario que ingrese el número de vuelo.
val numeroVuelo = readLine()?.trim() ?: "Desconocido"	Lee el número de vuelo y establece "Desconocido" como valor predeterminado si es nulo.
println("Ingrese el precio base para la reserva individual:")	Solicita al usuario que ingrese el precio base para la reserva individual.
val precioBase = readLine()?.trim()?.toDoubleOrNull() ?: 0.0	Lee el precio base ingresado y, si es nulo, establece 0.0 como valor predeterminado.
val tarifa = leerTarifa()	Llama a la función leerTarifa para obtener el tipo de tarifa seleccionada por el usuario.
return ReservaIndividual(tarifa, precioBase, nombrePasajero, numeroVuelo)	Crea y devuelve un nuevo objeto ReservaIndividual con los datos ingresados.
fun main() {	Declara la función principal main.
println("Ingrese el descuento para la reserva grupal (0.0 a 1.0):")	Solicita al usuario que ingrese el descuento para la reserva grupal.
val descuentoGrupal = leerDescuento()	Llama a la función leerDescuento para

Código	Explicación
	obtener el descuento ingresado por el usuario.
val reservaGrupal = ReservaGrupal(descuentoGrupal)	Crea una nueva instancia de ReservaGrupal con el descuento ingresado.
println("Ingrese el número de reservas individuales:")	Solicita al usuario que ingrese el número de reservas individuales que desea crear.
val numReservas = readLine()?.trim()?.toIntOrNull() ?: 0	Lee el número de reservas individuales y establece 0 como valor predeterminado si no es un número válido.
repeat(numReservas) { index ->	Inicia un bucle que se repetirá numReservas veces para solicitar datos sobre las reservas individuales.
val reservaIndividual = leerDatosReservaIndividual()	Llama a la función leerDatosReservaIndividual para crear un objeto ReservaIndividual.
reservaGrupal.agregarReserva(reservaIndividual)	Agrega la reserva individual creada a la reserva grupal.
println("Ingrese el número de reservas grupales anidadas (0 si no hay):")	Solicita al usuario que ingrese el número de reservas grupales anidadas.
val numReservasGrupales = readLine()?.trim()?.toIntOrNull() ?: 0	Lee el número de reservas grupales anidadas y establece 0 como valor predeterminado si no es un número válido.

Código	Explicación
repeat(numReservasGrupales) { index ->	Inicia un bucle para solicitar datos sobre reservas grupales anidadas, repitiéndose numReservasGrupales veces.
println("Ingrese el descuento para la reserva grupal anidada ${index + 1}:")	Solicita el descuento para cada reserva grupal anidada.
val descuentoAnidada = leerDescuento()	Llama a la función leerDescuento para obtener el descuento de la reserva grupal anidada.
val reservaGrupalAnidada = ReservaGrupal(descuentoAnidada)	Crea una nueva instancia de ReservaGrupal para la reserva anidada con el descuento correspondiente.
println("Ingrese el número de reservas individuales en la reserva grupal anidada ${index + 1}:")	Solicita al usuario que ingrese el número de reservas individuales para la reserva grupal anidada.
val numReservasAnidadas = readLine()?.trim()?.toIntOrNull() ?: 0	Lee el número de reservas individuales anidadas y establece 0 como valor predeterminado si no es un número válido.
repeat(numReservasAnidadas) { innerIndex ->	Inicia un bucle para solicitar datos sobre las reservas individuales anidadas, repitiéndose numReservasAnidadas veces.
val reservaIndividualAnidada = leerDatosReservaIndividual()	Llama a la función leerDatosReservaIndividu

Código	Explicación
	al para crear una reserva individual anidada.
reservaGrupalAnidada.agregarReserva(reservaIndividualAnidada)	Agrega la reserva individual anidada a la reserva grupal anidada.
reservaGrupal.agregarReserva(reservaGrupalAnidada)	Agrega la reserva grupal anidada a la reserva grupal principal.
println("Detalles de la reserva grupal:")	Imprime un mensaje indicando que se mostrarán los detalles de la reserva grupal.
println(reservaGrupal.detalles())	Llama a la función detalles de la reserva grupal y muestra los detalles de todas las reservas individuales y grupales.
println("Costo total de la reserva grupal: ${reservaGrupal.calcularCosto()}")	Calcula y muestra el costo total de la reserva grupal, considerando las reservas individuales y el descuento aplicado.

Conclusión

La realización de esta bitácora ha permitido consolidar el conocimiento adquirido a lo largo del desarrollo de diversos sistemas en Kotlin. Cada proyecto, desde la gestión de impuestos hasta la administración de nómina y reservas de servicios, ha aportado valiosas lecciones sobre programación orientada a objetos, diseño eficiente y aplicación de patrones de diseño.

A través de la documentación sistemática de cada sistema, se ha podido observar la evolución en la comprensión de conceptos clave, así como la mejora en la implementación de soluciones más efectivas y mantenibles. La posibilidad de analizar el código y su lógica ha permitido identificar áreas de mejora, optimizar procesos y fomentar la reutilización de componentes.

Además, la bitácora no solo ha servido como un registro del trabajo realizado, sino que también ha actuado como un recurso de referencia para futuros proyectos. Este proceso de documentación y reflexión es esencial para el crecimiento profesional, pues permite enfrentar nuevos desafíos con una base sólida de conocimientos y experiencias previas.

En conclusión, la integración de estos sistemas y su registro en la bitácora contribuyen no solo al desarrollo de habilidades técnicas, sino también a una mentalidad crítica y analítica necesaria para enfrentar el cambiante mundo del desarrollo de software. A medida que se continúan explorando nuevas tecnologías y metodologías, esta bitácora se convierte en una herramienta fundamental para el aprendizaje continuo y la innovación en futuros proyectos.