BITACORA DE PRUEBAS

APPLICACIONES MOVILES II

24 DE SEPTIEMBRE DE 2021

DOCENTE : AMBROSIO CARDOSO JIMENEZ
ALUMNA: LIZBETH VERONICA CASAS PEREZ
INGENIERIA EN INFORMATICA 9 SEMESTRE GRUPO A

Objetivos de la práctica:

- Aplicar los principios SOLID en cada ejercicio
- Identificar la problemática de cada problema
- Aplicar un código limpio
- Definir las clases y sus respectivos métodos

Ejercicio uno:

problemática: se desea diseñar una aplicación que permita calcular el importe total que una persona debe pagar por el impuesto predial, considerando que una persona puede tener varios predios.

En este caso decidí tomar el principio de **responsabilidad Única y Principio Open/Closed e inversión de dependencias**

Este principio establece que cada clase debe tener una única responsabilidad y esta responsabilidad debe estar definida y ser concreta. Todos los métodos deben estar alineados con la finalidad de la clase.

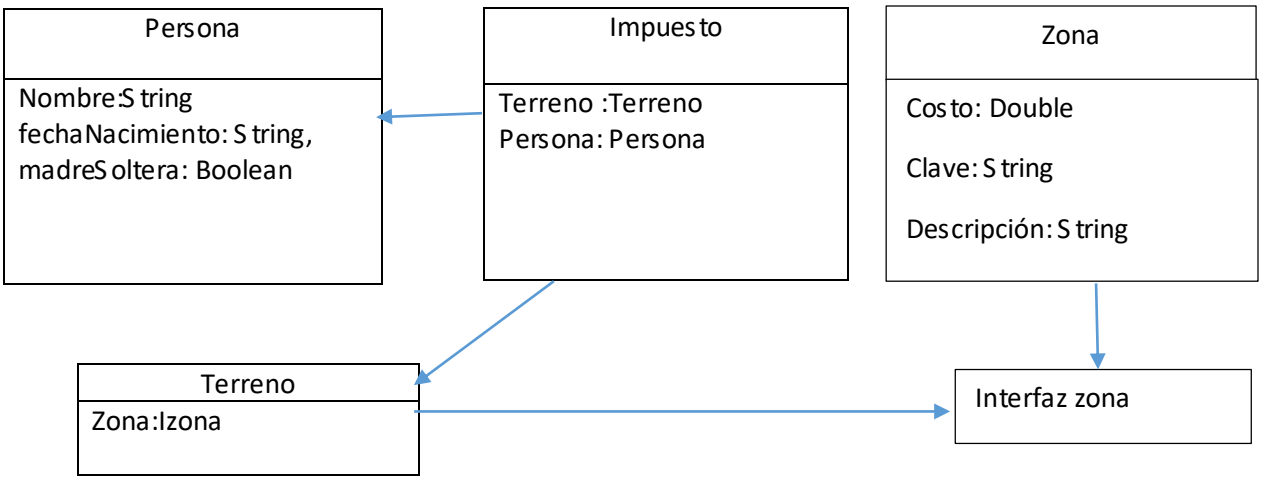
"Una clase debe tener solo una razón para cambiar" – Uncle Bob

Sin embargo, nunca debe modificar clases, interfaces y otras unidades de código que ya existan ya que puede provocar un comportamiento inesperado. Si agrega una nueva característica extendiendo su código en lugar de modificarlo, reduce el riesgo de falla tanto como sea posible.

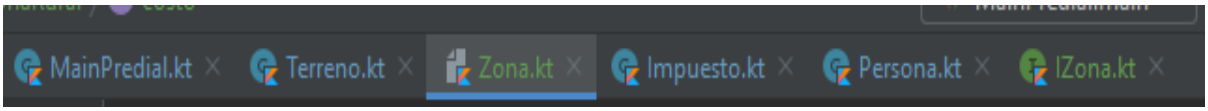
Inversión de dependencias: Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones (p.ej., interfaces).

Las abstracciones no deberían depender de los detalles. Los detalles (implementaciones concretas) deben depender de abstracciones.

En la problemática nos menciona varios puntos clave para las clases, que se necesita una **Persona** para ingresar sus datos y poder agregar los datos del predio**, Predio** necesita los datos de la persona para calcular el costo y el impuesto, **Terreno** llevaría los datos y características como tamaño, **Zona** tiene el cátalo de zonas donde contiene el costo, zona y la clave. A continuación, mostrare las clases a implementar:



1- A continuación, estas serán las clases a realizar en la creación del programa:



2- Iniciamos con la clase persona, es la clase que nos dará acceso a los datos para poder ingresar información descriptiva sobre el cálculo del predial, también nos ayudará a calcular la edad para saber si se le aplicara un descuento si es una persona mayor de edad o en caso de que sea una madre soltera, creamos un data class el cual tendrá los parámetros nombre, fechaNacimiento y madresoltera

```
data class Persona(val nombre: String, var fechaNacimiento: LocalDate, val madreSoltera: Boolean) {

    fun calcularEdad() = LocalDate.now().year - fechaNacimiento.year
}
```

3- se crea un atributo tipo boolean para ingresar si es verdadero o falso que es madre soltera, un LocalDate que es una variable de fecha y el nombre será de tipo String

```
data class Persona(val nombre: String, var fechaNacimiento: LocalDate, val madreSoltera: Boolean)
```

NOTA: Una data class es una clase que contiene solamente atributos que quedemos guardar en él.

4- Se crea una función en la clase persona, que contendrá el cálculo de la edad, para ello ocupamos una variable llamada LocalDate.now que se refiere a la

fecha actual en años, menos la variable fechaNacimiento que se creó en los parámetros de la clase en años

```
fun calcularEdad() = LocalDate.now().year - fechaNacimiento.year
```

5- Primero creamos una interfaz llamada IZona la cual tendrá los datos que se utilizaran para realizar el catálogo de zonas, para recordar, las interfaces te permiten definir tipos cuyos comportamientos pueden ser compartidos por varias clases que no están relacionadas, con el fin de crear instancias se adopten a un dominio específico.

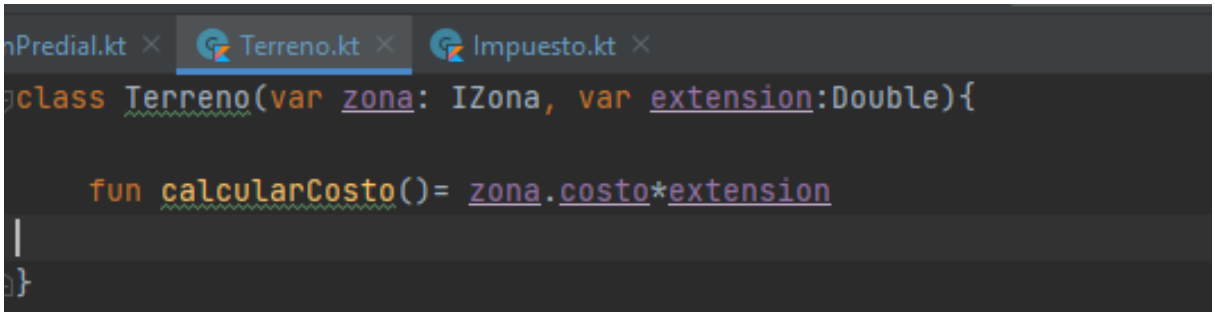
```
interface IZona{
    var clave:String
    var descripcion:String
    var costo:Double
```

6- Ahora creamos la clase zona, la cual obtendrá los datos del catálogo, haciendo referencia a la interfaz que se acabó de crear.

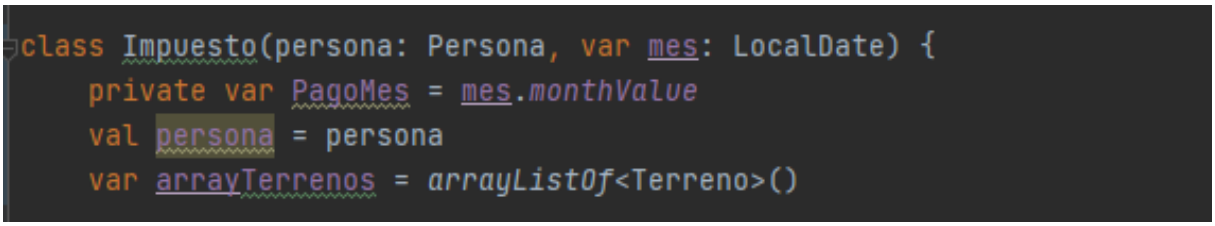
```
class zonaRural() : IZona {
    override var clave: String = "RUR"
    override var descripcion = "Rural"
    override var costo=8.0
}

class zonaUrbana() : IZona {
    override var clave: String = "URB"
    override var descripcion = "Urbana"
    override var costo=10.0
}

class zonaMarginada() : IZona {
    override var clave: String = "MAR"
    override var descripcion = "Marginada"
    override var costo=2.0
}

class zonaResidencial() : IZona {
    override var clave: String = "RES"
    override var descripcion = "Residencial"
    override var costo=25.0
}
```

Creamos varias clases de ella, cada una se clasifico por la zona, costo y su descripción.

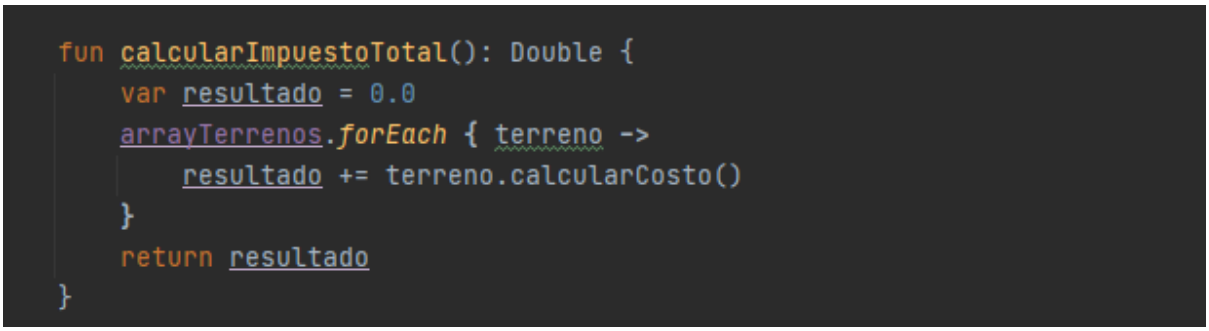
7- Creamos una clase llamada Terreno, esta clase manda a llamar a la zona:Izona para poder realizar el cálculo del costo el cual se multiplica el costo de la zona por su extensió



8- Una vez creada las clases mencionadas anteriormente, vamos a crear la clase llamada Impuesto, esta clase le pedirá prestado los datos obtenidos de la clase persona y los datos de la clase Terreno para realizar su operación, creamos una función llamada agregarTerreno



9- Creamos la función, calcularImpuestoTotal, primero creamos una variable resultado de tipo double, utilizamos el arreglo de terreno Iterando su array, a resultado se concatena con terreno de calcular el costo y obtenemos la variable resultado



10-Y por último creamos una función llamada calcularTotal se encargará de calcular los impuestos a pagar y los descuentos en caso de que se cumpla la condición

- creamos una variable de tipo LocalDate. Now que obtiene la fecha actual y creamos una variable llamada descuento de tipo double
- inicializamos el If, si el mes es menor o igual a dos, mes 1 es enero y mes 2 es febrero
- si persona del método calcularEdad es menor o igual a 70 o persona de madreSoltera es igual a verdadero, entonces aplicara un descuento del 70%, eso quiere decir que, cuando la persona ingrese su edad y sea mayor a 70 años o si la persona ingresa que es una madre soltera, entonces les aplicaran un descuento.
- Entonces si la persona no es mayor de edad o madre soltera, pero realiza su pago dentro de los dos meses indicados, tendrá un 40% de descuento
- Entonces si persona del método calcularEdad es mayor o igual a 70 o persona es madreSoltera y realiza su pago en los siguientes meses, tendrá un descuento del 50%

Dentro del mismo método creamos una variable total que manda a llamar al método calcularImpuestoTotal, entonces retornamos ese total * el descuento aplicado en esta función llamada calcularTotal.

```
fun calcularTotal(): Double {
    var descuento = 0.0
    if (PagoMes <= 2) {
        if (persona.calcularEdad() >= 70 || persona.madreSoltera.equals(true)) {
            descuento = 0.70

        } else {
            descuento = 0.40
        }
    } else if (persona.calcularEdad() >= 70 || persona.madreSoltera.equals(true)) {
        descuento = 0.50
    }
    var total = calcularImpuestoTotal()
    return (total - (total * descuento))
}
```

11-A continuación, creamos una clase main donde se realizarán las pruebas, para ver si el programa funciona correctamente:

- El propietario nació en el año 1991, el pago en el mes de junio, es madre soltera y vive en una zona residencial y el costo es de 800 pesos, pero también tiene otra propiedad urbana de 600 entonces:

```
        val propietario =
            Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1991,  month: 2,  dayOfMonth: 11), madreSoltera = true)

        val impuesto = Impuesto(propietario)
        impuesto.calcularTotal()
        impuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
        impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
        println("Total sin descuento:" + impuesto.calcularImpuestoTotal())
        println("Total :" + impuesto.calcularTotal())
    }
}
```

```
Tests passed: 1 of 1 test – 121 ms
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:26000.0
Total :13000.0

Process finished with exit code 0
```

- Si es una persona que no es madre soltera ni mayor de edad y pago en los meses de agosto entonces, no tiene descuento porque no aplica en los primeros meses:

```
        val propietario =
            Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1992,  month: 2,  dayOfMonth: 11), madreSoltera = false)

        val impuesto = Impuesto(propietario)
        impuesto.calcularTotal()
        impuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
        // impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
        println("Total sin descuento:" + impuesto.calcularImpuestoTotal())
        println("Total :" + impuesto.calcularTotal())
    }
}
```

```
Tests passed: 1 of 1 test – 239 ms
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:20000.0
Total :20000.0

Process finished with exit code 0
```

Si la persona tiene 79 años y pago en el mes de febrero entonces:

```
val propietario =
    Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1942,  month: 2,  dayOfMonth: 11), madreSoltera = fal
//val calcularMes = Impuesto( )


val impuesto = Impuesto(propietario,mes = LocalDate.of( year: 2021, month: 1, dayOfMonth: 9))
impuesto.calcularTotal()
impuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
// impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
println("Total sin descuento:" + impuesto.calcularImpuestoTotal())
println("Total :" + impuesto.calcularTotal())
```

Nos daría un descuento del 70%

```
C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:20000.0
Total :6000.0

Process finished with exit code 0
```

Ahora si eres madre soltera y pagas dentro del mes de enero:

```
al propietario =
   Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1998,  month: 2,  dayOfMonth: 11), madreSoltera = true)
/val calcularMes = Impuesto( )


al impuesto = Impuesto(propietario,mes = LocalDate.of( year: 2021, month: 1, dayOfMonth: 9))
mpuesto.calcularTotal()
mpuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
 impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
rintln("Total sin descuento:" + impuesto.calcularImpuestoTotal())
rintln("Total :" + impuesto.calcularTotal())
```

```
Tests passed: 1 of 1 test – 103 ms
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:20000.0
Total :6000.0
```

En cambio, si no res mayor de edad y no eres madre soltera y pagas en los meses de enero y febrero, tendrás un descuento del 40%

```
val propietario =
    Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1998,  month: 2,  dayOfMonth: 11), madreSoltera = false
//val calcularMes = Impuesto( )


val impuesto = Impuesto(propietario,mes = LocalDate.of( year: 2021, month: 1, dayOfMonth: 9))
impuesto.calcularTotal()
impuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
// impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
println("Total sin descuento:" + impuesto.calcularImpuestoTotal())
println("Total :" + impuesto.calcularTotal())
```

Tests passed: 1 of 1 test – 121 ms

```
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:20000.0
Total :12000.0

Process finished with exit code 0
```

Ahora si la persona tiene dos predios que pagar, pero de igual manera los paga el mes de enero, recibirá su descuento del 40%

```
    val propietario =
        Persona( nombre: "Pedro", fechaNacimiento = LocalDate.of( year: 1998,  month: 2,  dayOfMonth: 11), madreSolte
    //val calcularMes = Impuesto( )


    val impuesto = Impuesto(propietario,mes = LocalDate.of( year: 2021, month: 2, dayOfMonth: 9))
    impuesto.calcularTotal()
    impuesto.agregarTerreno(Terreno(zonaResidencial(), extension = 800.0))
    impuesto.agregarTerreno(Terreno(zonaUrbana(), extension = 600.0))
    println("Total sin descuento:" + impuesto.calcularImpuestoTotal())
    println("Total :" + impuesto.calcularTotal())
}
```

Tests passed: 1 of 1 test – 111 ms

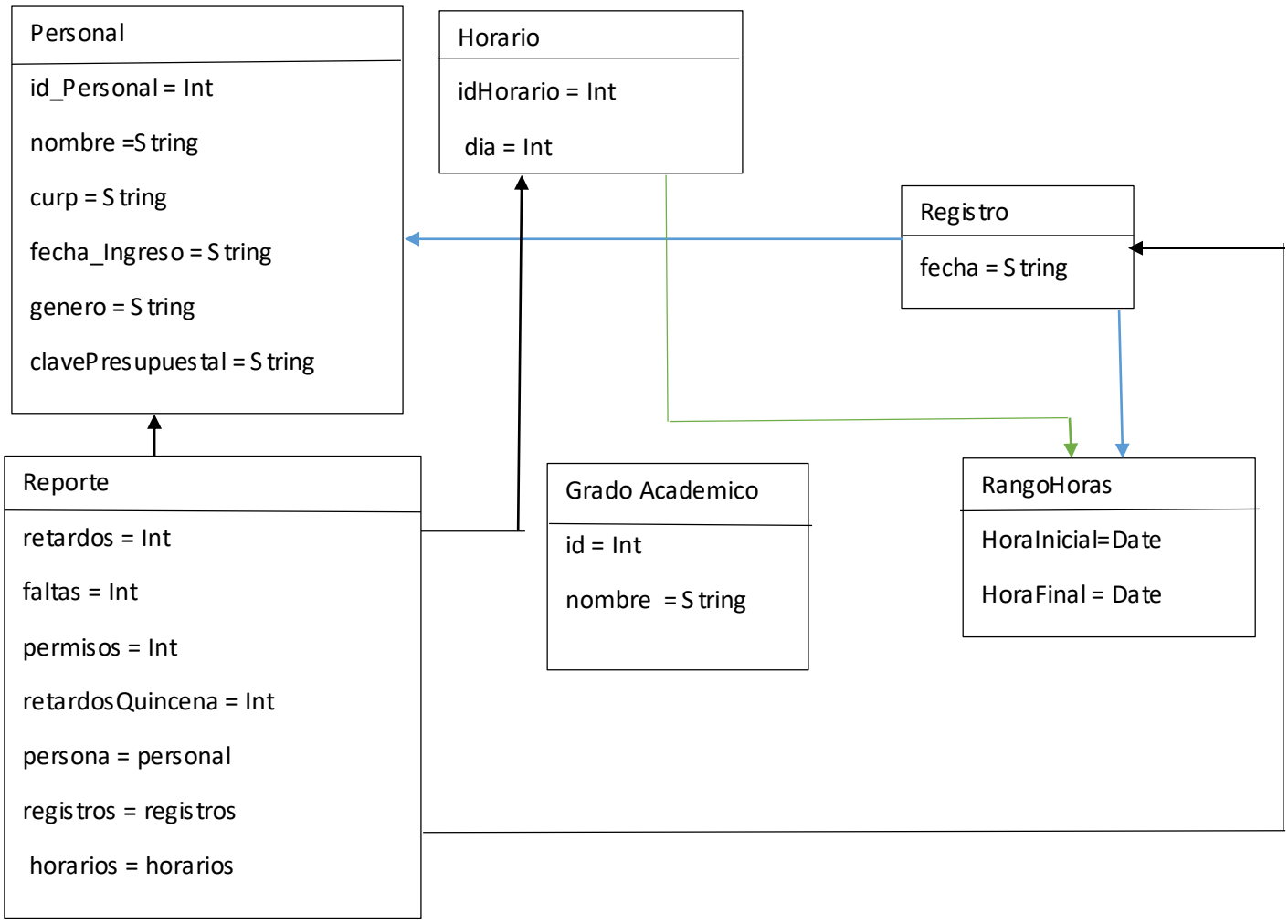
```
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Total sin descuento:26000.0
Total :15600.0
```

PRACTICA 2

Se desea crear un programa para el control de registros de entrada y salida de personal, deberá mostrar las faltas, retardos e inasistencias.

Solución: utilizare dos métodos el de responsabilidad Única, Segregación de interfaces

Las clases son las siguientes:



1- Primero creamos la clase persona con sus respectivos atributos, a la clase principal agregamos los parámetros:

```
data class Personal(val id_Personal: Int,val  nombre: String, val curp: String, var fecha_Ingreso:
LocalDate, val genero: String,val  gradoAcademico: GradoAcademico,val clavePresupuestal: String) {

    fun calcularAntigüedad() = LocalDate.now().year - fecha_Ingreso.year

}
```

2- Se crea una función en la clase personal, que contendrá el cálculo de la antigüedad del trabajador, para ello ocupamos una variable llamada LocalDate.now que se refiere a la fecha actual en años, menos la variable fechaIngreso que se creó en los parámetros de la clase en años

```
    fun calcularAntigüedad() = LocalDate.now().year - fecha_Ingreso.year

}
```

1- Creamos la clase registro de personal con sus respectivos atributos, la cual va a necesitar la clase de rangoHoras

```
import java.time.LocalDate

class Registro (val personal: Personal, fecha: String,val rangoHoras: RangoHoras){

    val fecha: LocalDate = LocalDate.parse(fecha)

}
```

1- Ahora creamos la clase horario, es donde estarán los datos del personal, su hora de registro y también necesitara a la clase RangoHoras

```
class Horario(val idHorario: Int, val rangoHoras: RangoHoras, val dia: DayOfWeek) {

}
```

2- De igual manera se crea una clase llamada gradoAcademico, será un data class y solo tendrá en el parámetro un id y un nombre del grado académico que tiene el personal

```
data class GradoAcademico(val id: Int, val nombre: String) {

}
```

3- Creamos una clase llamada RangoHoras esta clase tendrá la hora inicial y la hora final para poder calcular el tiempo cuando termina su hora de trabajo

```
class RangoHoras(val horaInicial: LocalTime, val horaFinal: LocalTime) {

}
```

4- Creamos una de las clases más importantes llamada reporte, su función es pasar los datos del personal, horarios y registros para poder generar el reporte final
   - Como mencione, se obtiene registros y el array registros, también sería lo mismo para horarios y el array de horarios y personal

```
ss Reporte(personal: Personal, registros: List<Registro>, horarios: List<Horario>) {

 var retardos = 0
 var faltas = 0
 var permisos = 0
 var retardosQuincena = 0
 val personal = personal
 val registros = registros
 val horarios = horarios
```

5- Se crea la primera función llamada generarReporte, en esta función es necesario obtener los datos de persona, su fecha inicial y final, el día actual en que inicio, primero mandamos a llamar a la fecha inicial de tipo LocalDate, y también la fechaFinal, mientras fechaActual es después de la fecha final
   - Cuando sea quincena se reinician los días para volver a contar
   - Se recorre el array de horario, si la clase horario de dia es diferente del dia actual, se calcula retardo.
   - La fecha actual es igual a los días agregados

```
fun generarReporte(fechaInicial: String, fechaFinal: String): Reporte {

    var fechaActual: LocalDate = LocalDate.parse(fechaInicial)
    var fin: LocalDate = LocalDate.parse(fechaFinal)
    var diaActual = fechaActual.dayOfWeek

    while (!fechaActual.isAfter(fin)) { //
        if (esQuincena(fechaActual))
            reiniciarRetardosPorQuincena()
        horarios.forEach { horario ->
            if (horario.dia == diaActual ){/
                var retardo = calcularRetardo( fechaActual, horario)
                calcularFalta(fechaActual, horario, retardo)
            }
        }
        fechaActual = fechaActual.plusDays( daysToAdd: 1)

    }
    return this
```

6- Se crea la función calcularRetardo el cual debe tener el día actual y necesita a la clase horario, se crea registro el cual obtendrá el registro del día actual, si el registro es diferente o no hay, no se tomará en cuenta, se crea una variable llamada llegada que obtendrá el registro de hora inicial y el rango de horas.

- Si llegada en minutos es de 5 a 15 después de la hora de entrada se hará un incremento en retardos.

```
fun calcularRetardo(diaActual: LocalDate, horario: Horario): Boolean {
    val registro = obtenerRegistro(diaActual)
    if (registro != null) { //
        var llegada = registro.rangoHoras.horaInicial.minus(
            horario.rangoHoras.horaInicial.getLong(ChronoField.MINUTE_OF_DAY),
            ChronoUnit.MINUTES)
        if (llegada.minute in 5..15){
            retardos++
            retardosQuincena++
            return true
        }
    }
    return false
```

7- Se crea una función llamada calcular falta la cual necesitara a la clase horario, y en sus parámetros tenemos dia actual con LocalDate y retardos

Se crea registros que obtendrá los registros del día actual, si registro es diferente o no se llega a registrar entonces genera una falta y se incrementan, mandamos a llamar a la clase personal con su método calcularAntiguedad y si la antigüedad es mayor a 10 años cada 3 retardos a la quincena serán contabilizados como una falta y se van incrementando

```
fun calcularFalta(diaActual: LocalDate, horario: Horario, retardo: Boolean): Boolean {

    val registro = obtenerRegistro(diaActual)
    if (registro == null) {//si no hay registro es falta
        //println("falta por ausencia")
        faltas++
        return true
    }
    if (personal.calcularAntigüedad() > 10)
        return false
    if (registro.rangoHoras.horaFinal.isBefore(horario.rangoHoras.horaFinal) || retardo && retardosQuincena % 3 == 0) {
        faltas++
        return true
    }
    return false
}
```

8- Se crean 3 funciones privadas cada una con una función importante, para poder realizar algunos métodos de la clase registro, obtenerRegistro se encarga que la clase registro de fecha sea diferente a día actual.

- Quincena obtiene los quince días del mes y los minutos del día
- Cada que se termine una quincena se reinicia desde cero y se vuelve a contar

```
private fun obtenerRegistro(diaActual: LocalDate): Registro? {
    return registros.find { registro -> registro.fecha == diaActual }
}


private fun esQuincena(diaActual: LocalDate): Boolean{
    return  diaActual.dayOfMonth == 15 || diaActual.minusDays( daysToSubtract: 1).month > diaActual.month

}

private fun reiniciarRetardosPorQuincena() {
    retardosQuincena = 0
}
```

9- Ahora nos vamos al main para realizar las pruebas y comprobar que funciona correctamente:

En la clase principal creamos un array donde van las especialidades de la clase grado académico

```
import org.junit.jupiter.api.Test

class MainControl {
    val grados = listOf<GradoAcademico>(
        GradoAcademico( id: 1, nombre: "Bachillerato"),
        GradoAcademico( id: 2, nombre: "Universidad"),
        GradoAcademico( id: 3, nombre: "Postgrado"),
    )
```

Creamos la función main donde se insertarán los datos de la clase Persona:

```
fun main(){
    print("ingrese su nombre de usuario")
    val nombre = "Lizbeth Veronica"
    print("ingrese su curp")
    val curp = "CAPL961127MOCSRZ05"
    print("ingrese su fecha de ingreso")
    val fechaI = "2021-09-18"
    print("ingrese su género")
    val genero = "M"
    print("ingrese su clave presupuestal")
    val clavePresupuestal = "drt"
```

Mandamos a llamar a personal ,con sus respectivos atributos, imprimimos la hora de entrada y salida y su dia de la semana

NOTA: el dia de la semana se puso como un arreglo lo cual es con numeros

```
    val personal: Personal = Personal(
        id_Personal = 1, nombre, curp,
        fechaI, genero, grados[0],clavePresupuestal)

    print("por favor asigne un horario al trabajador")

    print("ingrese la hora de entrada")
    val horaE = "8:00"
    print("ingrese la hora de salida")
    val horaS = "15:00"
    println("ingrese el dia de la semana")
    val dia = 1
```

Se agrega las horas en las que el trabajador entra y sale, también se registra la hora en que llego tarde y la hora en que salió antes de su horario establecido y se obtienen los datos:

```
val hora1: LocalTime = LocalTime.parse( text: "10:00")
val hora2: LocalTime = LocalTime.parse( text: "15:00")
val horaEntradaRetardo: LocalTime = LocalTime.parse( text: "10:15")
val horaSalidaFalta: LocalTime = LocalTime.parse( text: "14:50")
val rangoHoras = RangoHoras(hora1, hora2)
val rangoHorasTarde = RangoHoras(horaEntradaRetardo, hora2)
val rangoHorasFalta = RangoHoras(hora1, horaSalidaFalta)
  println(rangoHoras.compararHoraInicial(horaEntada))
```

10- Se agregan dos array uno donde se agregan los días de la semana y el segundo array es el de registro, donde vienen las fechas de los 5 días que trabajo por semana y sus quincenas

```
val horarios: List<Horario> = listOf(
    Horario( idHorario: 1,rangoHoras,DayOfWeek.MONDAY),
    Horario( idHorario: 2,rangoHoras, DayOfWeek.TUESDAY),
    Horario( idHorario: 3,rangoHoras, DayOfWeek.WEDNESDAY),
    Horario( idHorario: 4,rangoHoras, DayOfWeek.THURSDAY),
    Horario( idHorario: 5,rangoHoras, DayOfWeek.FRIDAY),
)
val registros: List<Registro> = listOf(
    Registro(personal,  fecha: "2021-09-13", rangoHoras),
    Registro(personal,  fecha: "2021-09-14", rangoHorasTarde),//retardo
    Registro(personal,  fecha: "2021-09-15", rangoHoras),
    Registro(personal,  fecha: "2021-09-16", rangoHoras),
    Registro(personal,  fecha: "2021-09-17", rangoHoras),
    Registro(personal,  fecha: "2021-09-20", rangoHoras),//1
    Registro(personal,  fecha: "2021-09-21", rangoHoras),
    Registro(personal,  fecha: "2021-09-22", rangoHoras),
    Registro(personal,  fecha: "2021-09-23", rangoHorasTarde),
    Registro(personal,  fecha: "2021-09-24", rangoHorasTarde),
    Registro(personal,  fecha: "2021-09-27", rangoHorasTarde),//2 faltas
    Registro(personal,  fecha: "2021-09-28", rangoHorasTarde),
    Registro(personal,  fecha: "2021-09-29", rangoHorasTarde),
    Registro(personal,  fecha: "2021-09-30", rangoHorasTarde),
    Registro(personal,  fecha: "2021-10-01", rangoHoras),
    Registro(personal,  fecha: "2021-10-02", rangoHoras),
    Registro(personal,  fecha: "2021-10-03", rangoHoras),
)
```

Inicia con un retardo en una quincena

En la siguiente quincena tiene 6 retardos , pero recordemos que es una persona que ingreso este año, lo que significa que por cada 3 retardos tendrá una falta, entonces tendría 2 faltas en esta quincena

11-Entonces verificamos:

```
Lizbeth Veronica ha tenido 7 retardos
Lizbeth Veronica ha tenido 2 faltas

Process finished with exit code 0
```

Conclusión: podríamos decir que cada retardo es contado hasta quince días que serían la quincena, de ahí iniciaría de nuevo el control, dependiendo de los años de antigüedad se puede contabilizar y en caso de salir antes del horario establecido.

Para trabajar los ejercicios me reuní con algunos compañeros para realizarlo en equipo

Emmanuel Josué López Zermeño, Gustavo Romero y Damián Martínez Jiménez ya que nos pareció una buena manera de interactuar y cambiar ideas en equipo, apoyándonos en algunas partes que también se complicaron al desarrollo del programa y resolviendo las dudas de manera práctica haciendo reuniones en meet

Adjunto evidencia: