Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia

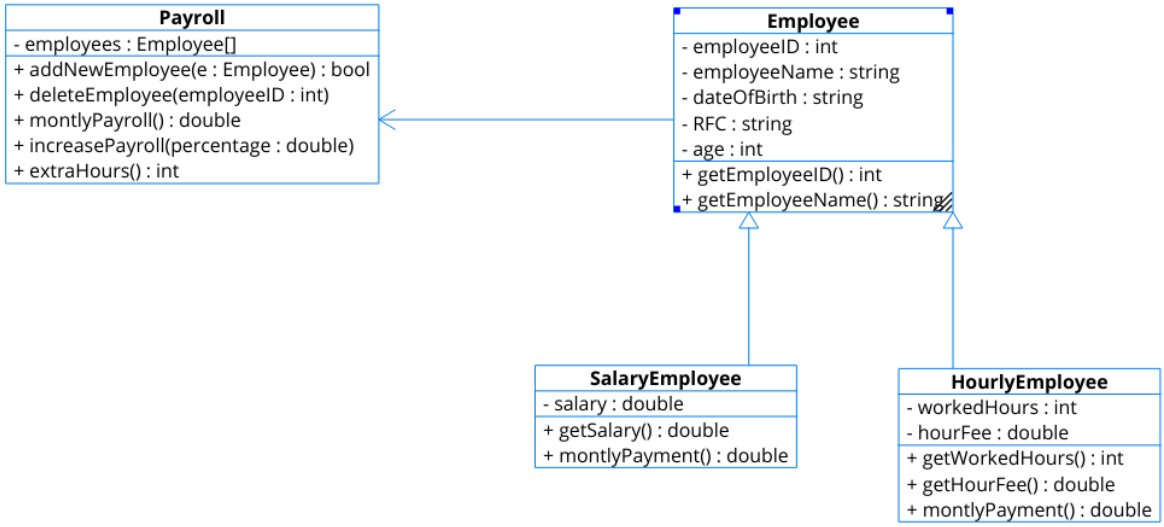
Laboratorios de computación salas A y B

Profesora:	Rocío Alejandra Aldeco Pérez
Asignatura:	Programación Orientada a Objetos
Grupo:	6
No de Práctica(s):	4
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	
Semestre:	2021-1
Fecha de entrega:	23 de octubre de 2020
Observaciones:	

CALIFICACIÓN: _____

Práctica de Estudio 4: Clases y objetos
Programación Orientada a Objetos Grupo 6
Facultad de Ingeniería
Departamento de Computación

Objetivo de la práctica: Aplicar los conceptos básicos de la programación orientada a objetos en un lenguaje de programación.



Realiza las siguientes actividades después de leer y revisar en clase la **Práctica de Estudio 4: Clases y objetos.**

1. Implementa las clases descritas en el diagrama anterior en Java tomando en cuenta los siguientes puntos:

   a) Para calcular el salario mensual (**monthlyPayment**): si el salario es mayor de 10,000 los impuestos a pagar serán del **30%**, en cualquier otro caso serán del **20%**. Para los empleados por hora los impuestos son de **10%**.

   b) El método **monthlyPayroll()** regresa la suma por mes de todos los salarios de todos los empleados.

   c) El método **increasePayroll(double percentage)** incrementa el salario de un empleado en un porcentaje dado.

   d) El método **extraHours()** regresa cuantos empleados trabajaron más de **40** horas.

2. Cuando estés seguro de que tu programa es correcto, súbelo a **Alphagrader**. Recuerda que si no pasas todos los test no obtendrás los puntos de ejecución.

3. Finalmente, en esta práctica, explica el código que generaste proceso y concluye. Recuerda incluir la caratula correspondiente y subir este documento en **PDF** junto con tu(s) archivo(s) **.java** a **Schoology**.

Primero, al notar que la clase **Employee** no se instanciaría en ningún momento, pero compartiría ciertas funcionalidades con sus hijos, se decidió definirla de forma abstracta. Además de los atributos propuestos, se agregaron algunos más: **currentId**: un atributo estático para almacenar los **ID**s a crear, y **birthLocalDate**, una instancia de **LocalDate** para almacenar la fecha de nacimiento en este formato.

```java
public abstract class Employee
{
    private static final Set<String> NOMBRES_PROHIBIDOS;
    private static final Set<String> APELLIDOS_PROHIBIDOS;
    private static final Set<Character> VOCALES;
    private static int currentId = 1;

    static
    {
        APELLIDOS_PROHIBIDOS = Set.of("DA", "DAS", "DE", "DEL", "DER", "DI", "DIE", "DD", "EL", "LA", "LOS"
        NOMBRES_PROHIBIDOS = new HashSet<>();
        NOMBRES_PROHIBIDOS.addAll(APELLIDOS_PROHIBIDOS);
        NOMBRES_PROHIBIDOS.addAll(Set.of("MARIA", "MA.", "MA", "JOSE", "J", "J."));
        VOCALES = Set.of('A', 'E', 'I', 'O', 'U');
    }

    private final LocalDate birthLocalDate;

    private final int employeeId;
    private final String employeeName;
    private final String dateOfBirth;
    private final String RFC;
    private final int age;
```

Atributos de Employee

De igual forma, se añadieron los atributos y métodos realizados en la práctica pasada para calcular el **RFC**. Además, en el constructor se inicializaron los atributos respectivos:

```java
public Employee(String fullName, String fecha) throws Exception
{
    this.employeeId = currentId++;
    this.employeeName = fullName;
    this.dateOfBirth = fecha;
    this.birthLocalDate = LocalDate.parse(this.dateOfBirth, DateTimeFormatter.ofPattern("dd/MM/yyyy"))
    this.age = LocalDate.now().getYear() - this.birthLocalDate.getYear();
    this.RFC = this.calcularRFC();
}
```

Constructor de Employee

Posteriormente, se implementaron los métodos planteados y se definieron los métodos abstractos mostrados a continuación:

```java
public int getEmployeeId() { return employeeId; }

public String getEmployeeName() { return employeeName; }


private char findFirstInternalVowel(String str) throws Exception
{
    char[] internalStr = str.substring(1, str.length() - 1).toCharArray();
    for (char c : internalStr)
    {
        if (Employee.VOCALES.contains(c)) return c;
    }
    throw new Exception("La cadena no contiene ninguna vocal interna");
}


private String getCleanName(String name, Set<String> forbiddenNames)
{
    // Quitar acentos y transformar a mayúsculas
    name = Normalizer.normalize(name.toUpperCase(), Normalizer.Form.NFD).rep

    String[] names = name.split( regex: " ");
    if (names.length == 1) return names[0];

    String cleanName = Arrays.stream(names)
            .filter(Predicate.not(forbiddenNames::contains))
            .reduce( identity: "", (acc, part) -> acc + " " + part)
            .strip();

    if (cleanName.length() == 0) return names[0];
    else return cleanName;
}
```

Métodos implementados

```java
    public String calcularRFC() throws Exception
    {
        String[] names = this.employeeName.split( regex: " ");
        String cleanName = this.getCleanName(names[0], NOMBRES_PROHIBIDOS);
        String cleanFirstLastName = this.getCleanName(names[1], APELLIDOS_PROHIBIDOS);
        String cleanSecondLastName = this.getCleanName(names[2], APELLIDOS_PROHIBIDOS);
        String dateInfo = this.birthLocalDate.format(DateTimeFormatter.ofPattern("yyMMdd"));

        return cleanFirstLastName.substring(0, 1)
                + this.findFirstInternalVowel(cleanFirstLastName)
                + cleanSecondLastName.charAt(0)
                + cleanName.charAt(0)
                + dateInfo;
    }

    abstract public double monthlyPayment();

    abstract public double monthlyPaymentBeforeTaxes();

    abstract public void increaseSalaryBy(double percentage);
}
```

Método calcularRFC y métodos abstractos

Los métodos abstractos se implementarían finalmente en las clases **HourlyEmployee** y **SalaryEmployee**. La función de **monthlyPayment** es calcular el pago mensual de un empleado, después de impuestos, mientras que **monthlyPaymentBeforeTaxes** retorna el salario mensual de un empleado antes de impuestos. El método **increaseSalaryBy** aumenta el sueldo del empleado por el porcentaje proporcionado.

A continuación, se muestra la implementación de la clase **SalaryEmployee**:

```java
public class SalaryEmployee extends Employee
{
    private static final double MAX_SALARY = 10000;
    private static final double EXCEEDED_PERCENTAGE = 0.7;
    private static final double NORMAL_PERCENTAGE = 0.8;
    private double salary;

    public SalaryEmployee(String fullName, String fecha, String salary) throws Exception
    {
        super(fullName, fecha);
        this.salary = Double.parseDouble(salary);
    }

    public double getSalary() { return salary; }

    @Override
    public double monthlyPayment()
    {
        return salary > MAX_SALARY ? salary * EXCEEDED_PERCENTAGE : salary * NORMAL_PERCENTAGE;
    }

    @Override
    public double monthlyPaymentBeforeTaxes() { return this.salary; }

    @Override
    public void increaseSalaryBy(double percentage) { this.salary *= 1 + (percentage / 100); }
}
```

Clase SalaryEmployee

Como se logra observar, se definieron las constantes **MAX_SALARY**, **EXCEEDED_PERCENTAGE** Y **NORMAL_PERCENTAGE**, que contendrían el salario máximo antes de cambiar el porcentaje de cobro de impuestos, el porcentaje de cobro de impuestos cuando el salario excede la cantidad especificada y el porcentaje cuando no lo hace, respectivamente. De igual forma, se declaró el atributo **salary**. En el **constructor**, se inicializaron los datos respectivos utilizando el constructor de la clase padre.

Además, se implementaron los métodos abstractos antes mencionados: **monthlyPayment,** que retorna el salario con el porcentaje de impuestos aplicado dependiendo de la cantidad de salario; **monthlyPaymentBeforeTaxes,** que retorna el salario sin modificaciones; e **increaseSalaryBy**, que aumenta el salario por el porcentaje proporcionado.

A continuación, se muestra la implementación de la clase **HourlyEmployee**:

```java
public class HourlyEmployee extends Employee
{
    private final static double NORMAL_PERCENTAGE = 0.9;
    private final int workedHours;
    private double hourFee;

    public HourlyEmployee(String fullName, String fecha, String workedHours, String hourFee) throw
    {
        super(fullName, fecha);
        this.workedHours = Integer.parseInt(workedHours);
        this.hourFee = Double.parseDouble(hourFee);
    }

    @Override
    public double monthlyPayment() { return monthlyPaymentBeforeTaxes() * NORMAL_PERCENTAGE; }

    @Override
    public double monthlyPaymentBeforeTaxes() { return hourFee * workedHours; }

    @Override
    public void increaseSalaryBy(double percentage) { this.hourFee *= 1 + (percentage / 100); }

    public int getWorkedHours()
    {
        return workedHours;
    }

    public double getHourFee()
    {
        return hourFee;
    }
}
```

Clase HourlyEmployee

Se definió la constante **NORMAL_PERCENTAGE**, que contendría el porcentaje de impuestos a aplicar. En el constructor, se inicializaron los atributos correspondientes y se utilizó el constructor de la clase padre. Se implementaron los *getters* especificados en el diagrama, y los métodos abstractos de la clase padre, de una forma muy similar a la de la clase **SalaryEmployee**.

Finalmente, se implementó la clase **Payroll**:

```java
public class Payroll
{
    private final Map<Integer, Employee> employees;

    public Payroll(List<Employee> employees)
    {
        this.employees = new HashMap<>();
        employees.forEach(employee → this.employees.put(employee.getEmployeeId(), employee));
    }

    public boolean addNewEmployee(Employee employee)
    {
        this.employees.put(employee.getEmployeeId(), employee);
        return true;
    }

    public void deleteEmployee(int employeeID)
    {
        this.employees.remove(employeeID);
    }

    public double monthlyPayroll()
    {
        return employees.values() Collection<Employee>
                .stream() Stream<Employee>
                .map(Employee::monthlyPayment) Stream<Double>
                .reduce( identity: 0.0, Double::sum);
    }
```

Clase Payroll

Para almacenar a los empleados se decidió utilizar la estructura **Map**, ya que, al almacenarlos en una estructura secuencial como **ArrayList** y eliminar alguna instancia, los índices donde éstas se encuentran se modificarían y serían inconsistentes con sus los **IDs** respectivos. Al utilizar **Map**, podemos eliminar cualquier instancia sin modificar el proceso de búsqueda de las demás.

El constructor recibe una lista de empleados, inicializa la lista de empleados con una instancia de la clase **HashMap**, y añade cada elemento a ésta última, utilizando como llaves los **IDs** de los empleados. Los métodos **addNewEmployee** y **deleteEmployee**

utilizan los métodos **put** y **remove**, respectivamente, para proporcionar la funcionalidad deseada. El método **monthlyPayroll** retorna la suma de los salarios después de impuestos, utilizando las utilidades **map** y **reduce**, para calcular la cantidad mencionada para cada empleado y posteriormente sumarlas.

El método **monthlyPayrollBeforeTaxes** retorna la suma de todos los salarios antes de impuestos. Su mecanismo es el mismo que el del método anterior, pero utiliza la función **monthlyPaymentBeforeTaxes** para calcular la cantidad deseada para cada empleado.

```
public double monthlyPayrollBeforeTaxes()
{
    return employees.values() Collection<Employee>
            .stream() Stream<Employee>
            .map(Employee::monthlyPaymentBeforeTaxes) Stream<Double>
            .reduce( identity: 0.0, Double::sum);
}

public void increasePayroll(int employeeID, double percentage)
{
    this.employees.get(employeeID).increaseSalaryBy(percentage);
}

public int extraHours()
{
    return employees.values() Collection<Employee>
            .stream() Stream<Employee>
            .filter(employee → employee instanceof HourlyEmployee)
            .map(employee → ((HourlyEmployee) employee).getWorkedHours()) Stream<Integer>
            .reduce( identity: 0, (acc, hoursWorked) → hoursWorked ⩾ 40 ? acc + 1 : acc);
}
```

Métodos de la clase Payroll

El método **increasePayroll** incrementa el salario de un empleado por cierto porcentaje. Para lograr lo anterior, primero se busca el empleado en el **Map** de empleados, y se ejecuta el método de la clase **Empleado, increaseSalaryBy**, el cual ya fue analizado anteriormente. Finalmente, el método **extraHours** retorna la cantidad de empleados que trabajaron una cantidad mayor o igual a **40** horas. Nótese que esta condición sólo aplica para los **HourlyEmployees**. Para cumplir con dicha funcionalidad, primero se obtienen todos los valores del **Map** de empleados, se utiliza la utilidad **filter** para *"filtrar"* los

empleados que trabajan por hora, se calcula el número de horas que trabajaron utilizando **map**, y finalmente se calcula el número de trabajadores con un número de horas mayor o igual a 40, utilizando la utilidad **reduce**.

```
public static void main(String[] args) throws Exception
{
    Scanner sc = new Scanner(System.in);
    List<Employee> employees = new ArrayList<>();
    while (sc.hasNext())
    {
        String[] data = sc.nextLine().split( regex: ",");
        Employee newEmployee;
        if (data[2].equals("S"))
        {
            newEmployee = new SalaryEmployee(data[0], data[1], data[3]);
        }
        else
        {
            newEmployee = new HourlyEmployee(data[0], data[1], data[3], data[4]);
        }
        employees.add(newEmployee);
    }
    Payroll payroll = new Payroll(employees);
    printFormattedValue(payroll.monthlyPayrollBeforeTaxes());
    printFormattedValue(payroll.monthlyPayroll());
    printFormattedValue(payroll.extraHours());
    payroll.increasePayroll( employeeID: 4,  percentage: 5);
    payroll.increasePayroll( employeeID: 6,  percentage: 3);
    printFormattedValue(payroll.monthlyPayrollBeforeTaxes());
    printFormattedValue(payroll.monthlyPayroll());
}
```

Método main

Por último, en el método principal se obtienen los datos correspondientes de la entrada estándar, se crea una lista de empleados, y se instancian las clases respectivas, dependiendo del tipo de salario. Se crea una instancia de **Payroll**, se ejecutan las operaciones planteadas y se imprimen los resultados de ser necesario.

CONCLUSIONES

El paradigma orientado a objetos nos permite modelar ciertos problemas de una forma muy natural. Mediante la creación de **clases**, podemos representar entidades que cuentan con determinada información y funcionalidad asociada. Además, podemos establecer relaciones entre los objetos, lo cual es una característica frecuente en el modelado de problemas.

En el problema presentado en la práctica, se utilizaron los constructos mencionados para modelar y resolver el problema planteado. La naturaleza del lenguaje y del problema facilitaron en gran medida la tarea anterior, al permitir representar las entidades correspondientes con su información y funcionalidades respectivas.

Por otro lado, en un lenguaje cuyo paradigma no es el orientado a objetos, la resolución de este problema se hubiera realizado de una forma distinta. En muchos casos, el paradigma orientado a objetos provee un enfoque que parece adecuado para problemas como el presentado en la práctica. Sin embargo, lo anterior es totalmente subjetivo, ya que es posible resolver un mismo problema de distintas formas.