Ejemplo Herencia en Java

El siguiente ejercicio plantea un caso de estudio con base a una problemática inicial, como parte del diseño se realizará el diagrama de clases y finalmente se realizará la codificación en la fase de implementación.

Se espera que más adelante se pueda dar continuidad al ejercicio con interfaces gráficas de usuario.

PLANTEAMIENTO DEL PROBLEMA.

El departamento de Informática de un hospital está realizando un nuevo registro de datos de todas las personas que tienen relación con esa institución, que son: los empleados, los médicos y los pacientes, Los empleados son categorizados en función de si son contratados por Planilla o de forma Eventual y son los encargados de los procesos administrativos.
Los médicos (que podrían considerarse un tipo especial de empleados contratados por Planilla) se encargan de las atenciones de las consultas médicas.

Para solicitar una cita, el paciente es atendido por un empleado, el paciente indica el servicio en el cual quiere pasar consulta y el empleado le indica el nombre del médico, la fecha y la hora de la cita.

Los atributos de cada uno de ellos se indican a continuación:

- Persona: número de DNI, nombre, apellido, fecha de nacimiento, dirección, ciudad de procedencia.
- Paciente: número de historia clínica, sexo, grupo sanguíneo, lista de medicamentos a los que es alérgico.
- Empleado: código de Empleado, número de horas extras, fecha de ingreso, área, cargo.
- Empleado por Planilla: salario mensual, porcentaje adicional por hora extra.
- Empleado Eventual: honorarios por hora, fecha de término del contrato.
- Médico: especialidad, número de consultorio.

Para desarrollar el sistema, en la clase Persona se deberá crear el método concreto **imprimirDatosPersona(String datos)**, así como un método **registrarDatos()** , y un método **registrarCitaMedica()** que sea implementados por las clases que lo requieran.

La aplicación a desarrollar debe permitir:

1. Registrar los datos de los empleados, los pacientes y los médicos.
2. Registrar los datos de una cita médica.
3. Imprimir los datos de cada persona.

Tenga en cuenta que lo anterior es solo el planteamiento inicial, usted deberá aplicar los conceptos de programación vistos hasta el momento para dar solución al problema, deberá crear los métodos y clases que considere necesario, por ejemplo, en el enunciado se le dice que debe crear un método llamado **registrarDatos()** en la clase persona pero debe analizar como registrar Pacientes, empleados o médicos.

Solución Propuesta

Para dar solución al problema planteado inicialmente se crea un diagrama de clases propuesto como parte del análisis del sistema de información, recordemos que este diagrama hace parte de **UML (Lenguaje de Modelado Unificado)**, UML permite modelar sistemas de información abarcando diferentes procesos y etapas del funcionamiento de un sistema siguiendo una notación gráfica especifica, por eso se habla de que es un lenguaje, un lenguaje grafico para especificar el comportamiento del software, hay diagramas UML para modelar como interactúan los usuarios con el sistema, para modelar como interactúan internamente los componentes del sistema, otros para modelar como se va a desplegar el sistema y que elementos intervienen en este despliegue entre muchos otros modelos.

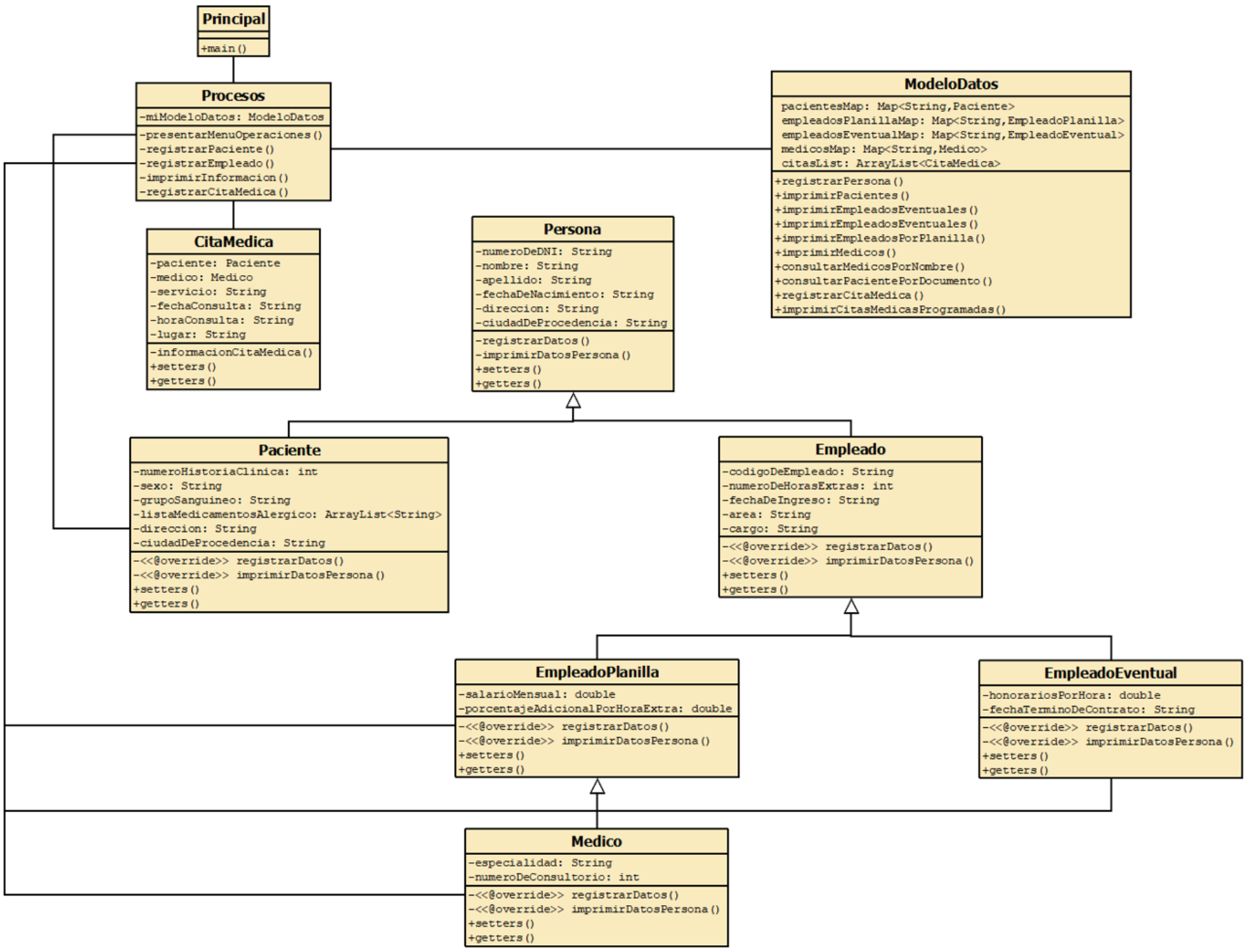
Diagrama de clases.

Para este caso, vamos a crear un Diagrama de Clases, este es un tipo de diagrama estático que describe la estructura del sistema mostrando las clases que lo componen junto con sus atributos, métodos y relaciones con otras clases.

El siguiente diagrama define la estructura del proyecto, basado en el problema planteado, aquí se puede evidenciar el árbol de herencia definido así como otras clases creadas que si bien, no son explicitas en la definición del problema, su creación es necesaria para la construcción del sistema.

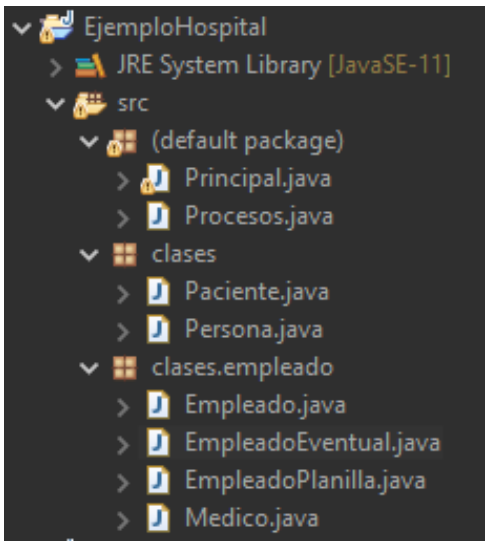
Por ejemplo se crea la clase "**Procesos**" que controlará el sistema en general, así como la clase "**ModeloDatos**" que centralizará el almacenamiento de la información y la clase "**CitaMedica**" que permitirá gestionar las citas medicas como objetos del sistema.

Por otro lado veamos las relaciones, vemos como algunas clases se comunican con otras pero por ejemplo no vemos una comunicación entre **Procesos** y **Persona** o **Empleado**, ya que estas ultimas tienen sentido en el sistema para definir elementos que pueden ser extendidos (Heredados) por otras clases, pero nunca se va a crear un objeto de tipo Persona o de tipo Empleado de forma explicita, ya que el problema plantea la creación de Pacientes, Empleados eventuales, de planilla o Médicos.



Construcción del proyecto.

Para la construcción del sistema se creará un proyecto java con la siguiente estructura inicial (es una estructura propuesta basada en el diagrama de clases, pero no quiere decir que elementos como el nombrado o estructura de paquetes tenga que ser así)



En esta estructura hacen falta clases o elementos de los definidos en el diagrama de clases, ya que esto se irá agregando en la medida que se va construyendo el proyecto.

A continuación se presentan las diferentes clases según el árbol de herencia definido, nótese que hacemos uso de la encapsulación para proteger el estado de los objetos según la clase definida, esto lo hacemos al definir los atributos privados y establecer el acceso a los mismos mediante los métodos set() y get() (los niveles de seguridad los definen los desarrolladores, en este caso tan solo estamos encapsulando los atributos para accederse mediante los métodos públicos pero no se hace ninguna validación.)

Clase Persona.java

Se crea la clase junto con los métodos **imprimirDatos(String datos)** y **registrarDatos()** con una lógica definida, esta lógica se realiza como parte de la estructura de la clase, pero a medida que se avanza en el desarrollo puede ser modificada.

Persona.java

```java
package clases;
import javax.swing.JOptionPane;

public class Persona {

    private String numeroDeDNI;
    private String nombre;
    private String apellido;
    private String fechaDeNacimiento;
    private String direccion;
    private String ciudadDeProcedencia;

    public void imprimirDatosPersona(String datos){

        datos+="Nombre: "+nombre+"\n";
        datos+="Apellido: "+apellido+"\n";
        datos+="Fecha de Nacimiento: "+fechaDeNacimiento+"\n";
        datos+="Dirección: "+direccion+"\n";
        datos+="Ciudad de Procedencia: "+ciudadDeProcedencia+"\n";

        System.out.println(datos);
    }

    public void registrarDatos() {
        numeroDeDNI=JOptionPane.showInputDialog("Ingrese el numero del documento");
        nombre=JOptionPane.showInputDialog("Ingrese el nombre");
        apellido=JOptionPane.showInputDialog("Ingrese el Apellido");
        fechaDeNacimiento=JOptionPane.showInputDialog("Ingrese fecha nacimiento (dd/mm/aaaa)");
        direccion=JOptionPane.showInputDialog("Ingrese la dirección");
        ciudadDeProcedencia=JOptionPane.showInputDialog("Ingrese la ciudad de Procedencia");
    }


    public String getNumeroDeDNI() {
        return numeroDeDNI;
    }
    public void setNumeroDeDNI(String numeroDeDNI) {
        this.numeroDeDNI = numeroDeDNI;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
```

Clase Paciente.java

En esta clase se crean los atributos propios de la clase, noten que en el enunciado se menciona que los pacientes deben tener la lista de medicamentos a los que son alérgicos, por esa razón se construye un ArrayList que permita almacenar esta información (la lista es de tipo String, esto porque tan solo vamos a almacenar el nombre del medicamento, en caso de que se requiriera gestionar más detalles de los medicamentos como por ejemplo código, nombre, descripción, tipo etc, entonces se debería haber creado una clase **Medicamento** que permitiera la construcción de objetos y de esta manera la lista seria de tipo **ArrayList<Medicamento>** pero dado el enunciado, no es necesario).

Adicional nótese como se hereda de la clase Persona usando la palabra **extends**, lo que nos permite extender los atributos de **Persona** a **Paciente**.

Paciente.java

```java
package clases;

import java.util.ArrayList;

public class Paciente extends Persona{

    private int numeroHistoriaClínica;
    private String sexo;
    private String grupoSanguineo;
    private ArrayList<String> listaMedicamentosAlergico;

    public int getNumeroHistoriaClínica() {
        return numeroHistoriaClínica;
    }
    public void setNumeroHistoriaClínica(int numeroHistoriaClínica) {
        this.numeroHistoriaClínica = numeroHistoriaClínica;
    }
    public String getSexo() {
        return sexo;
    }
    public void setSexo(String sexo) {
        this.sexo = sexo;
    }
    public String getGrupoSanguineo() {
        return grupoSanguineo;
    }
    public void setGrupoSanguineo(String grupoSanguineo) {
        this.grupoSanguineo = grupoSanguineo;
    }
    public ArrayList<String> getListaMedicamentosAlergico() {
        return listaMedicamentosAlergico;
    }
    public void setListaMedicamentosAlergico(ArrayList<String> listaMedicamentosAlergico) {
        this.listaMedicamentosAlergico = listaMedicamentosAlergico;
    }
}
```

Clase Empleado.java

Esta clase tendrá la función de servir como superClase para los tipos de empleados, pero en ningún momento se crearán objetos directos de ella, los objetos creados corresponderán a sus hijos.

Empleado.java

```java
package clases.empleado;

import clases.Persona;

public class Empleado extends Persona{

    private String codigoDeEmpleado;
    private int numeroDeHorasExtras;
    private String fechaDeIngreso;
    private String area;
    private String cargo;

    public String getCodigoDeEmpleado() {
        return codigoDeEmpleado;
    }
    public void setCodigoDeEmpleado(String codigoDeEmpleado) {
        this.codigoDeEmpleado = codigoDeEmpleado;
    }
    public int getNumeroDeHorasExtras() {
        return numeroDeHorasExtras;
    }
    public void setNumeroDeHorasExtras(int numeroDeHorasExtras) {
        this.numeroDeHorasExtras = numeroDeHorasExtras;
    }
    public String getFechaDeIngreso() {
        return fechaDeIngreso;
    }
    public void setFechaDeIngreso(String fechaDeIngreso) {
        this.fechaDeIngreso = fechaDeIngreso;
    }
    public String getArea() {
        return area;
    }
    public void setArea(String area) {
        this.area = area;
    }
    public String getCargo() {
        return cargo;
    }
    public void setCargo(String cargo) {
        this.cargo = cargo;
    }

}
```

Clase EmpleadoEventual.java

Esta clase Hereda de la clase Empleado.java y como esta también hereda de Persona.java quiere decir que los empleados eventuales también podrán acceder a todas las funciones y atributos de dichas clases dado el árbol de herencia.

EmpleadoEventual.java

```java
package clases.empleado;

public class EmpleadoEventual extends Empleado{

    private double honorariosPorHora;
    private String fechaTerminoContrato;

    public double getHonorariosPorHora() {
        return honorariosPorHora;
    }
    public void setHonorariosPorHora(double honorariosPorHora) {
        this.honorariosPorHora = honorariosPorHora;
    }
    public String getFechaTerminoContrato() {
        return fechaTerminoContrato;
    }
    public void setFechaTerminoContrato(String fechaTerminoContrato) {
        this.fechaTerminoContrato = fechaTerminoContrato;
    }

}
```

Clase EmpleadoPlanilla.java

Esta clase Hereda también de la clase Empleado.java y como esta también hereda de Persona.java quiere decir que los empleados por planilla también podrán acceder a todas las funciones y atributos de dichas clases dado el árbol de herencia.

EmpleadoPlanilla.java

```java
package clases.empleado;

public class EmpleadoPlanilla extends Empleado{

    private double salarioMensual;
    private double porcentajeAdicionalPorHoraExtra;

    public double getSalarioMensual() {
        return salarioMensual;
    }
    public void setSalarioMensual(double salarioMensual) {
        this.salarioMensual = salarioMensual;
    }
    public double getPorcentajeAdicionalPorHoraExtra() {
        return porcentajeAdicionalPorHoraExtra;
    }
    public void setPorcentajeAdicionalPorHoraExtra
                                (double porcentajeAdicionalPorHoraExtra) {
        this.porcentajeAdicionalPorHoraExtra = porcentajeAdicionalPorHoraExtra;
    }
}
```

Clase Medico.java

Esta clase Hereda de la clase EmpleadoPlanilla.java y basado en lo anterior sabemos que el Medico puede tomar características de su padre (**EmpleadoPlanilla**), su abuelo (**Empleado**) y su Bisabuelo (**Persona**)

Medico.java

```java
package clases.empleado;

public class Medico extends EmpleadoPlanilla{

    private String especialidad;
    private int numeroDeConsultorio;

    public String getEspecialidad() {
        return especialidad;
    }
    public void setEspecialidad(String especialidad) {
        this.especialidad = especialidad;
    }
    public int getNumeroDeConsultorio() {
        return numeroDeConsultorio;
    }
    public void setNumeroDeConsultorio(int numeroDeConsultorio) {
        this.numeroDeConsultorio = numeroDeConsultorio;
    }
}

```

Uso de la sobre-escritura de métodos.

La sobre- escritura de métodos es un mecanismo que nos permite indicarle al sistema que a pesar de que una clase Hija hereda métodos de una clase Padre, esta clase Hija puede dar una implementación propia a dichos métodos, sin tener que depender de lo que el Padre diga, esto lo hacemos mediante la anotación **@override** (si no se pone se tomará el método como propio más no heredado)

Aplicado a nuestro ejemplo, vemos como la clase Paciente sobre-escribe el método **registrarDatos()** de la clase Persona dando su propia implementación, que en este caso es necesaria ya que se deben solicitar los datos de ingreso para sus atributos, pero a la vez noten como en la linea 18 se hace el llamado al método **registrarDatos()** de la clase **Persona** (mediante el **super**.) ya que primero se llama a este método en el que se pide la información de la persona y luego de esto se piden los datos del Paciente.

(Se debe tener mucho cuidado ya que si no se usa la palabra "super" se podría estar incurriendo en un llamado al propio método de la clase Paciente lo que ocasionaría un llamado infinito que desbordaría el sistema.)

Como parte de la lógica de este método, se hace el llenado de la lista de medicamentos en caso de que estos existan.

Paciente.java

```java
package clases;

import java.util.ArrayList;

public class Paciente extends Persona{

    private int numeroHistoriaClínica;
    private String sexo;
    private String grupoSanguineo;
    private ArrayList<String> listaMedicamentosAlergico;

    @Override
    public void registrarDatos() {

        //llamamos al metodo registrar datos de la superClase para continuar llenando los datos del paciente junto con los heredados
        super.registrarDatos();

        //llenamos los datos especificos del paciente
        listaMedicamentosAlergico=new ArrayList<String>();
        numeroHistoriaClínica=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el numero de la historia clinica"));
        sexo=JOptionPane.showInputDialog("Ingrese el sexo");
        grupoSanguineo=JOptionPane.showInputDialog("Ingrese el grupo sanguineo");

        String pregunta=JOptionPane.showInputDialog("¿Es alergico a algun medicamento? ingrese si o no");

        if (pregunta.equalsIgnoreCase("si")) {
            String medicamento="";
            String continuar="";
            do {
                medicamento=JOptionPane.showInputDialog("Ingrese el nombre del medicamento al que es alergico");
                listaMedicamentosAlergico.add(medicamento);

                continuar=JOptionPane.showInputDialog("Ingrese si, si desea continuar");

            } while (continuar.equalsIgnoreCase("si"));
        }


    }
```

En la clase Medico.java también vemos el proceso de sobre-escritura anterior, aunque en este caso no se desarrolla mayor lógica más que el llenado de datos.

Medico.java

```java
package clases.empleado;
import javax.swing.JOptionPane;

public class Medico extends EmpleadoPlanilla{

    private String especialidad;
    private int numeroDeConsultorio;

    @Override
    public void registrarDatos() {
        super.registrarDatos();

        especialidad=JOptionPane.showInputDialog("Ingrese su especialidad");
        numeroDeConsultorio=Integer.parseInt(JOptionPane.showInputDialog("Num consultorio"));
    }
```

Lo mismo sucede con la clase EmpleadoEventual.java

EmpleadoEventual.java

```java
package clases.empleado;

import javax.swing.JOptionPane;

public class EmpleadoEventual extends Empleado{

    private double honorariosPorHora;
    private String fechaTerminoContrato;

    @Override
    public void registrarDatos() {
        super.registrarDatos();

        honorariosPorHora=Double.parseDouble(JOptionPane.showInputDialog("Ingrese el salario Mensual"));
        fechaTerminoContrato=JOptionPane.showInputDialog("Ingrese fecha nacimiento (dd/mm/aaaa)");

    }
```

En EmpleadoPlanilla.java también se hace necesario realizar el proceso.

EmpleadoPlanilla.java

```java
package clases.empleado;
import javax.swing.JOptionPane;

public class EmpleadoPlanilla extends Empleado{

    private double salarioMensual;
    private double porcentajeAdicionalPorHoraExtra;

    @Override
    public void registrarDatos() {
        super.registrarDatos();

        salarioMensual=Double.parseDouble(JOptionPane.showInputDialog("Ingrese el salario Mensual"));
        porcentajeAdicionalPorHoraExtra=Double.parseDouble(JOptionPane
                                .showInputDialog("Ingrese el Porcentaje Adicional por Hora Extra"));
    }

    public double getSalarioMensual() {
```

Finalmente se hace lo mismo en la clase Empleado.java ya que esta es la SuperClase de los diferentes tipos de empleados (incluyendo al Medico) y si queremos crear objetos independientes entonces la información del empleado y de Persona es fundamental.

Empleado.java

```java
package clases.empleado;

import javax.swing.JOptionPane;

public class Empleado extends Persona{

    private String codigoDeEmpleado;
    private int numeroDeHorasExtras;
    private String fechaDeIngreso;
    private String area;
    private String cargo;

    @Override
    public void registrarDatos() {
        super.registrarDatos();

        codigoDeEmpleado=JOptionPane.showInputDialog("Ingrese el codigo del empleado");
        numeroDeHorasExtras=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el numero de horas extras"));
        fechaDeIngreso=JOptionPane.showInputDialog("Ingrese la fecha de ingreso (dd/mm/aaaa)");
        area=JOptionPane.showInputDialog("Ingrese el area");
        cargo=JOptionPane.showInputDialog("Ingrese el cargo");
    }

```

Nota: Este mismo proceso de la sobre-escritura se realizará más adelante con el método de imprimirDatosPersona(datos) ya que se requiere imprimir toda la información tanto de Persona como de sus hijos.

Lógica del Sistema.

Hasta este punto se ha creado la estructura interna del sistema, ahora vamos a ver como se realiza la lógica para darle vida a los diferentes procesos.

Clase Principal.java

Como vemos esta clase tan solo nos brindará el llamado a la clase Procesos.java desde donde se le dará vida al sistema (en este caso teniendo en cuenta que solo requerimos una instancia de Procesos, hubiera bastado con escribir new Procesos() sin necesidad de crear el objeto "misProcesos")

Principal.java

```
1
2  public class Principal {
3
4      public static void main(String arg[]) {
5
6          Procesos misProcesos=new Procesos();
7
8      }
9  }
```

Clase Procesos.java

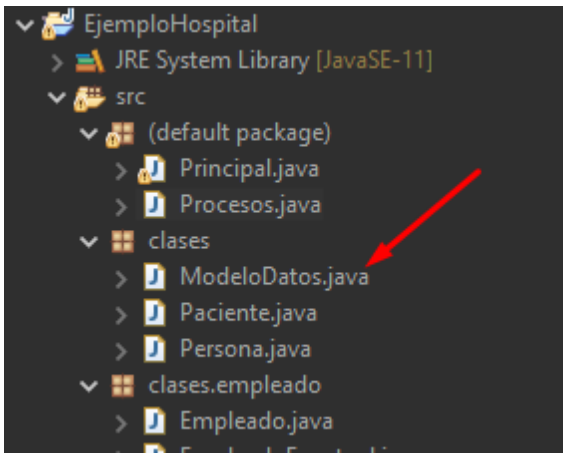
Como se mencionó, esta clase será la encargada de darle vida al sistema, por el momento se construye la estructura de la misma, usando el constructor para llamar al método que presentará el menú de opciones y la lógica para el llamado a los diferentes métodos del sistema. (Por ahora están vacíos pero a medida que se avance se van construyendo.)

Procesos.java

```java
import javax.swing.JOptionPane;

public class Procesos {

    public Procesos() {
        presentarMenuOpciones();
    }

    private void presentarMenuOpciones() {

        String menu="MENU HOSPITAL\n\n";
        menu+="1. Registrar Paciente\n";
        menu+="2. Registrar Empleado\n";
        menu+="3. Registrar Cita Medica\n";
        menu+="4. Imprimir Información\n";
        menu+="5. Salir\n\n";
        menu+="Ingrese una Opción\n";

        int opcion=0;

        do {
            opcion=Integer.parseInt(JOptionPane.showInputDialog(menu));
            switch (opcion) {
            case 1: registrarPaciente();    break;
            case 2: registrarEmpleado();    break;
            case 3: registrarCitaMedica();  break;
            case 4: imprimirInformacion();  break;
            case 5: System.out.println("El sistema ha terminado!"); break;
            default: System.out.println("No existe el código, verifique nuevamente");
                break;
            }

        } while (opcion!=5);
    }

    private void registrarPaciente() {
        // TODO Auto-generated method stub

    }

    private void registrarEmpleado() {
        // TODO Auto-generated method stub

    }
```

Clase ModeloDatos.java

Para gestionar la información de los diferentes Pacientes, Empleados, Médicos y Citas, se crea la clase ModeloDatos.java que centralizará todos esos procesos de almacenamiento, esta la crearemos en el paquete clases (tan solo por organización)



Lógica de Registro - (Aplicación de la sobre-carga de métodos en la clase ModeloDatos.java)

Esta clase será nuestro "modelo de datos" allí crearemos una estructura mediante **HashMap** de los diferentes tipos de elementos que queremos gestionar, como vemos por estructura creamos con un ámbito global los Mapas y se instancian

en el constructor de la clase.

Adicional creamos la lógica de almacenamiento para cada elemento (Pacientes, Empleados, Médicos) y para esto hacemos uso del concepto de sobre-carga de métodos que como sabemos, consiste en crear varios métodos con el mismo nombre pero con diferente "firma" o argumentos, esto nos facilitará el llamado a los procesos sin preocuparnos por como se llama el método puntual y preocupándonos solo por los parámetros enviados.

De esta manera cuando ingresemos a alguno de los métodos de registro que reciben alguno de los objetos específicos, entonces almacenamos en el mapa correspondiente el objeto recibido como parámetro.

ModeloDatos.java

```java
package clases;

import java.util.HashMap;

public class ModeloDatos {

    HashMap<String, Paciente> pacientesMap;
    HashMap<String, EmpleadoPlanilla> empleadosPlanillaMap;
    HashMap<String, EmpleadoEventual> empleadosEventualMap;
    HashMap<String, Medico> medicosMap;

    public ModeloDatos() {
        pacientesMap=new HashMap<String, Paciente>();
        empleadosPlanillaMap=new HashMap<String, EmpleadoPlanilla>();
        medicosMap=new HashMap<String, Medico>();
        empleadosEventualMap=new HashMap<String, EmpleadoEventual>();
    }

    public void registrarPersona(Paciente miPaciente) {
        pacientesMap.put(miPaciente.getNumeroDeDNI(), miPaciente);
        System.out.println("Se ha registrado el paciente "+miPaciente.getNombre()+" Satisfactoriamente");
    }

    public void registrarPersona(EmpleadoPlanilla miEmpPlanilla) {
        empleadosPlanillaMap.put(miEmpPlanilla.getNumeroDeDNI(), miEmpPlanilla);
        System.out.println("Se ha registrado el empleado por planilla "+miEmpPlanilla.getNombre()+" Satisfactoriamente");
    }

    public void registrarPersona(EmpleadoEventual miEmpEventual) {
        empleadosEventualMap.put(miEmpEventual.getNumeroDeDNI(), miEmpEventual);
        System.out.println("Se ha registrado el empleado eventual "+miEmpEventual.getNombre()+" Satisfactoriamente");
    }

    public void registrarPersona(Medico miMedico) {
        medicosMap.put(miMedico.getNumeroDeDNI(), miMedico);
        System.out.println("Se ha registrado el medico "+miMedico.getNombre()+" Satisfactoriamente");
    }

}
```

Regresando a la clase Procesos.java debemos crear la instancia de nuestro Modelo de Datos (de forma global para poder usar un único objeto con los datos internos de los mapas, si se crean múltiples instancias de esta clase, cada vez tendríamos mapas nuevos y debemos garantizar que los mapas sean los mismos para toda la ejecución del sistema...)

Procesos.java

```java
import javax.swing.JOptionPane;

public class Procesos {

    ModeloDatos miModeloDatos;

    public Procesos() {
        miModeloDatos=new ModeloDatos();
        presentarMenuOpciones();
    }

    private void presentarMenuOpciones() {

        String menu="MENU HOSPITAL\n\n";
        menu+="1. Registrar Paciente\n";
```

Veamos entonces como desde el método de **registrarPaciente**() creamos la lógica de registro, que consiste simplemente en crear un objeto de tipo Paciente, llamamos al método **registrarDatos**() (qué al ser sobre-escrito solicitará los datos del

Paciente y luego los de Persona) y por último llamamos al método **registrarPersona**(**miPaciente**) al que le enviamos el paciente creado y gracias a la sobre-carga entonces ingresará al método que recibe pacientes para almacenarlo en el mapa correspondiente.

```
33              case 4: imprimirInformacion();  break;
34              case 5: System.out.println("El sistema ha terminado!"); break;
35              default: System.out.println("No existe el código, verifique nuevamente");
36                  break;
37              }
38
39          } while (opcion!=5);
40      }
41
42      private void registrarPaciente() {
43          Paciente miPaciente=new Paciente();
44          miPaciente.registrarDatos();
45
46          miModeloDatos.registrarPersona(miPaciente);
47      }
48
49      private void registrarEmpleado() {
```

Un proceso similar es el realizado para el registro de empleados, la diferencia radica en que como hay una jerarquía de herencia mayor entonces nos aseguramos mediante un menú de que el usuario seleccione el tipo de empleado a crear, pero note que al final se realiza el mismo proceso de creación de la instancia y posterior llamado al método de registro que se llama igual pero le enviamos los objetos diferentes.

Procesos.java

```
49          miModeloDatos.registrarPersona(miPaciente);
50      }
51
52      private void registrarEmpleado() {
53          String menuTipoEmpleado="Registro de Empleado\n";
54          menuTipoEmpleado+="1. Empleado Eventual\n";
55          menuTipoEmpleado+="2. Empleado por Planilla\n";
56          menuTipoEmpleado+="Seleccione el tipo de empleado a registrar\n";
57
58          int tipoEmpleado=Integer.parseInt(JOptionPane.showInputDialog(menuTipoEmpleado));
59
60          switch (tipoEmpleado) {
61          case 1: //Registro Empleado Eventual
62                  EmpleadoEventual miEmpleadoEventual=new EmpleadoEventual();
63                  miEmpleadoEventual.registrarDatos();
64                  miModeloDatos.registrarPersona(miEmpleadoEventual);
65              break;
66          case 2:
67              String resp=JOptionPane.showInputDialog("Ingrese si, si es un médico, de lo contrario es otro tipo de empleado");
68              if (resp.equalsIgnoreCase("si")) {
69                  //Registro Medico
70                  Medico miMedico=new Medico();
71                  miMedico.registrarDatos();
72                  miModeloDatos.registrarPersona(miMedico);
73              }else {
74                  //Registro Empleado Planilla
75                  EmpleadoPlanilla miEmpleadoPlanilla=new EmpleadoPlanilla();
76                  miEmpleadoPlanilla.registrarDatos();
77                  miModeloDatos.registrarPersona(miEmpleadoPlanilla);
78              }
79
80              break;
81
82          default:System.out.println("El tipo de empleado no es valido, intentelo nuevamente");
83              break;
84          }
85
86      }
87
```

Lógica para la Impresión de información.

Para la impresión de la información creamos en la clase Procesos.java un método que presenta un menú de opciones donde el usuario define que información quiere imprimir.

Noten como se hace uso del objeto "**miModeloDatos**" para acceder a los diferentes métodos.

Procesos.java

```java
    private void imprimirInformacion() {

        String menuImprimir="MENU IMPRESIONES\n";
        menuImprimir+="1. Listar Pacientes\n";
        menuImprimir+="2. Listar Empleados Eventuales\n";
        menuImprimir+="3. Listar Empleados Por Planilla\n";
        menuImprimir+="4. Listar Medicos\n";
        menuImprimir+="Ingrese una opción\n";

        System.out.println("************************************************************");

        int opcion=Integer.parseInt(JOptionPane.showInputDialog(menuImprimir));

        switch (opcion) {
        case 1: miModeloDatos.imprimirPacientes();  break;
        case 2: miModeloDatos.imprimirEmpleadosEventuales();    break;
        case 3: miModeloDatos.imprimirEmpleadosPorPlanilla();   break;
        case 4: miModeloDatos.imprimirMedicos();    break;

        default: System.out.println("No exite esa opción");
            break;
        }
    }
```

Desde el ModeloDatos.java se presentan diferentes formas de imprimir los **HashMap**, en este caso cualquiera de las formas es valida y se podría usar la misma en todos los métodos, sin embargo se exponen las diferentes maneras posibles para su análisis.

Nota: Analice las diferentes maneras de recorrer los **HashMap** y determine la lógica para acceder a la información.

ModeloDatos.java

```java
    public void imprimirPacientes() {
        String msj="PACIENTES REGISTRADOS\n";
        Iterator<String> iterador=pacientesMap.keySet().iterator();

        while (iterador.hasNext()) {

            String clave = iterador.next();
            pacientesMap.get(clave).imprimirDatosPersona(msj);
        }

    }

    public void imprimirEmpleadosEventuales() {
        String msj="EMPLEADOS EVENTUALES REGISTRADOS\n";

        for (String clave : empleadosEventualMap.keySet()) {
            empleadosEventualMap.get(clave).imprimirDatosPersona(msj);
        }
    }

    public void imprimirEmpleadosPorPlanilla() {
        String msj="EMPLEADOS POR PLANILLA REGISTRADOS\n";

        for (EmpleadoPlanilla empleadoPlanilla : empleadosPlanillaMap.values()) {
            empleadoPlanilla.imprimirDatosPersona(msj);
        }
    }

    public void imprimirMedicos() {
        String msj="MEDICOS REGISTRADOS\n";

        for (Map.Entry<String, Medico> elemento : medicosMap.entrySet()) {
            //System.out.println("Key = " + elemento.getKey() + ", Value = " + elemento.getValue());
            //medicosMap.get(elemento.getKey()).imprimirDatosPersona(msj);
            elemento.getValue().imprimirDatosPersona(msj);
        }
    }

```

Veamos el proceso interno con más detalle.

Imprimir Pacientes.

Para imprimir pacientes usamos un Iterator con el cual obtenemos las diferentes claves y de esta manera llamamos al metodo imprimirDatosPersona(msj) de cada objeto paciente dentro del mapa.

ModeloDatos.java

```
39      public void registrarPersona(Medico miMedico) {
40          medicosMap.put(miMedico.getNumeroDeDNI(), miMedico);
41          System.out.println("Se ha registrado el medico "+miMedico.getNombre()+" Satisfactoriamente");
42      }
43
44      public void imprimirPacientes() {
45          String msj="PACIENTES REGISTRADOS\n";
46          Iterator<String> iterador=pacientesMap.keySet().iterator();
47
48          while (iterador.hasNext()) {
49
50              String clave = iterador.next();
51              pacientesMap.get(clave).imprimirDatosPersona(msj);
52          }
53
54      }
55
```

Para esto en la clase Paciente debemos tener el método sobre-escrito de imprimirDatosPersona(String datos) que recibe un mensaje que se irá alimentando para imprimir la información cada vez.

Inicialmente en la linea 45 se llama al método de la superClase y se envía el mensaje recibido el cual llega en la linea 13 de la clase Persona para que se concatene con la información de los atributos y se impriman.

Luego se continua nuevamente con la linea 47 de la clase Paciente donde se reasigna a la variable "datos" los datos propios del paciente (no se concatena ya que en la clase persona se imprimió con un System la información)

y luego se recorre la lista de medicamos del paciente en caso de que existan y se imprime la información.

Paciente.java

```
42
43      @Override
44      public void imprimirDatosPersona(String datos) {
45          super.imprimirDatosPersona(datos);
46
47          datos="Numero Historia Clinica: "+numeroHistoriaClínica+"\n";
48          datos+="Sexo: "+sexo+"\n";
49          datos+="Grupo Sanguineo: "+grupoSanguineo+"\n";
50
51          if (listaMedicamentosAlergico.size()>0) {
52              datos+="Lista de Medicamentos a los que es Alergico\n";
53              for (int i = 0; i < listaMedicamentosAlergico.size(); i++) {
54                  datos+="\t"+listaMedicamentosAlergico.get(i)+"\n";
55              }
56          }else {
57              datos+="El paciente, no es alergico a ningún medicamento";
58          }
59
60          System.out.println(datos);
61      }
62
```

Persona.java

```
10      private String direccion;
11      private String ciudadDeProcedencia;
12
13      public void imprimirDatosPersona(String datos){
14
15          datos+= "Nombre: "+nombre+"\n";
16          datos+= "Apellido: "+apellido+"\n";
17          datos+= "Fecha de Nacimiento: "+fechaDeNacimiento+"\n";
18          datos+= "Dirección: "+direccion+"\n";
19          datos+= "Ciudad de Procedencia: "+ciudadDeProcedencia;
20
21          System.out.println(datos);
22      }
23
```

Imprimir empleados eventuales

El mismo proceso se realiza con los empleados eventuales, veamos como se recorre el mapa pero ahora haciendo uso de un foreach, en el que obtenemos las claves gracias al método keySet() del HashMap y al hacerlo hacemos el llamado al método imprimir de los objetos almacenados.

```
public void imprimirEmpleadosEventuales() {
    String msj="EMPLEADOS EVENTUALES REGISTRADOS\n";

    for (String clave : empleadosEventualMap.keySet()) {
        empleadosEventualMap.get(clave).imprimirDatosPersona(msj);
    }
}
```

En este caso vemos como se realiza el mismo proceso de llamado en la clase de EmpleadoEventual.java y posteriormente a su Clase Padre Empleado.java y a la vez se hace el llamado al super de su SuperClase Persona.java comportándose de

la misma manera que se explicó anteriormente.

EmpleadoEventual.java

```java
18
19    @Override
20    public void imprimirDatosPersona(String datos) {
21        super.imprimirDatosPersona(datos);
22
23        datos = "Honorarios por Hora: " + honorariosPorHora + "\n";
24        datos += "Fecha Termino del Contrato: " + fechaTerminoContrato + "\n";
25
26        System.out.println(datos);
27    }
28
```

Empleado.java

```java
27    public void imprimirDatosPersona(String datos) {
28        super.imprimirDatosPersona(datos);
29
30        datos = "Codigo De Empleado: " + codigoDeEmpleado + "\n";
31        datos += "Numero De Horas Extras: " + numeroDeHorasExtras + "\n";
32        datos += "Fecha De Ingreso: " + fechaDeIngreso + "\n";
33        datos += "Area: " + area + "\n";
34        datos += "Cargo: " + cargo + "\n";
35
36        System.out.println(datos);
37    }
38
```

Persona.java

```java
13    public void imprimirDatosPersona(String datos){
14
15        datos+= "Nombre: "+nombre+"\n";
16        datos+= "Apellido: "+apellido+"\n";
17        datos+= "Fecha de Nacimiento: "+fechaDeNacimiento+"\n";
18        datos+= "Dirección: "+direccion+"\n";
19        datos+= "Ciudad de Procedencia: "+ciudadDeProcedencia;
20
21        System.out.println(datos);
22    }
23
```

Imprimir empleados eventuales

El mismo procedimiento se realiza con los empleados por planilla, pero en este caso el foreach no recorre las claves sinó que recorre los valores directamente para acceder a los objetos y a su método de imprimir

```java
public void imprimirEmpleadosPorPlanilla() {
    String msj="EMPLEADOS POR PLANILLA REGISTRADOS\n";

    for (EmpleadoPlanilla empleadoPlanilla : empleadosPlanillaMap.values()) {
        empleadoPlanilla.imprimirDatosPersona(msj);
    }
}
```

Si analizamos el proceso de impresión tendrá el mismo comportamiento explicado con anterioridad.

EmpleadoPlanilla.java

```java
18    @Override
19    public void imprimirDatosPersona(String datos) {
20        super.imprimirDatosPersona(datos);
21
22        datos = "Salario Mensual: " + salarioMensual + "\n";
23        datos += "Porcentaje Adicional Por Hora Extra: " + porcentajeAdicionalPorHoraExtra + "\n";
24
25        System.out.println(datos);
26    }
27
```

Imprimir Médicos

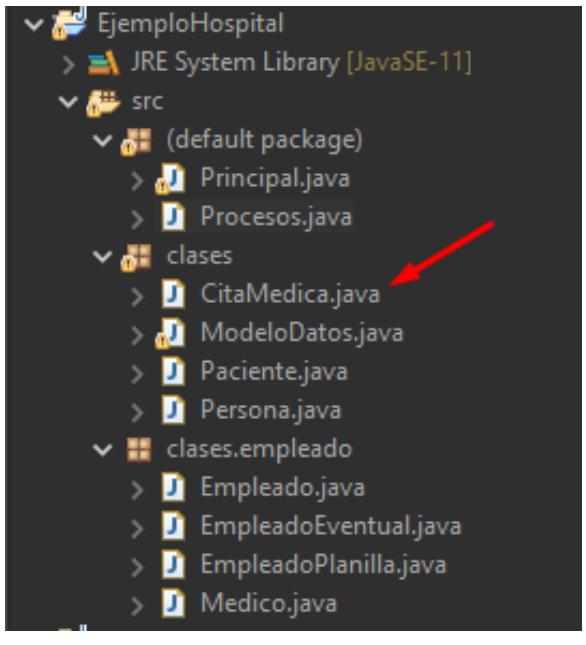
Finalmente para el caso de los médicos vemos qué se usa otro foreach pero utilizando el Map.Entry lo que nos da un elemento desde el cual obtenemos los objetos gracias al getValue (en los comentarios se puede ver más claramente la estructura de los elementos retornados y otra manera de acceder a la información.)

```java
public void imprimirMedicos() {
    String msj="MEDICOS REGISTRADOS\n";

    for (Map.Entry<String, Medico> elemento : medicosMap.entrySet()) {
        //System.out.println("Key = " + elemento.getKey() + ", Value = " + elemento.getValue());
        //medicosMap.get(elemento.getKey()).imprimirDatosPersona(msj);
        elemento.getValue().imprimirDatosPersona(msj);
    }
}
```

El proceso de impresión de información es el mismo, navegando por los métodos sobre-escritos para mostrar los datos.

Lógica registro de citas médicas.

Para la creación de citas médicas se crea una clase llamada "CitaMedica.java" ya que lo ideal es poder tener registrado en todo momento que citas médicas se solicitan, por eso la mejor alternativa es mediante la creación de objetos que puedan ser almacenados en otra estructura de datos.



Clase CitaMedica.java

Para la creación de la cita médica se definen unos atributos base, noten como estos atributos son de tipo String y de Tipo "Clase" en este caso tenemos que almacenar información del paciente y del médico por lo que se opta por tener atributos de tipo Paciente y Medico respectivamente ya que se debe obtener información adicional de cada uno, por ejemplo, el numero de consultorio que tiene asignado el medico será el lugar de la cita y que se evidencia en el método de imprimir Información de la cita.

Como vemos también se crea un constructor explicito el cual nos permitirá crear de forma directa los objetos de tipo **CitaMedica** (lo veremos más adelante) al recibir y asignar directamente los valores, sin embargo por temas de estructura también se encapsula la clase creando los métodos set y get.

Nota: En este punto ya se darán cuenta que al tener el constructor explicito, no se hará uso de los métodos de acceso para el envió de información (setter) por lo tanto se puede pensar que no es necesario crearlos, sin embargo por escalabilidad deben crearse, pues a medida que el sistema avance, ya se tendrá una estructura definida que facilitará el proceso de desarrollo.

CitaMedica.java

```java
public class CitaMedica {

    private Paciente paciente;
    private Medico medico;
    private String servicio;
    private String fechaConsulta;
    private String horaConsulta;
    private String lugar;

    public CitaMedica(Paciente paciente, Medico medico, String servicio, String fechaConsulta, String horaConsulta,
            String lugar) {
        this.paciente = paciente;
        this.medico = medico;
        this.servicio = servicio;
        this.fechaConsulta = fechaConsulta;
        this.horaConsulta = horaConsulta;
        this.lugar = lugar;
    }

    public String informacionCitaMedica() {

        String datosCita="<< INFORMACIÓN CITA MEDICA >>\n";
        datosCita="Paciente: "+paciente.getNombre()+"\n";
        datosCita+="Medico: "+medico.getNombre()+"\n";
        datosCita+="Motivo Consulta: "+servicio+"\n";
        datosCita+="Fecha Consulta: "+fechaConsulta+" - Hora: "+horaConsulta+"\n";
        datosCita+="Lugar: "+lugar+"\n\n";

        return datosCita;
    }

    public Paciente getPaciente() {
        return paciente;
    }
    public void setPaciente(Paciente paciente) {
        this.paciente = paciente;
    }
    public Medico getMedico() {
        return medico;
    }
    public void setMedico(Medico medico) {
        this.medico = medico;
    }
    public String getServicio() {
        return servicio;
```

Lógica para el registro de citas

Como parte del proceso de creación de citas médicas, debemos tener muy presente el proceso o paso a paso para la creación de las mismas, este proceso consiste en lo siguiente:

- Inicialmente si se va a crear una cita para un paciente, entonces debemos garantizar que este paciente esté previamente registrado para poderle asociar la cita por lo tanto en el modelo de datos creamos un método de consulta de pacientes por documento de identidad, el cual recibirá el documento y retornará el paciente encontrado para posteriormente registrarlo en la cita.

ModeloDatos.java

```java

    public Paciente consultarPacientePorDocumento(String documentoPaciente) {
        Paciente miPaciente=null;

        if (pacientesMap.containsKey(documentoPaciente)) {
            miPaciente=pacientesMap.get(documentoPaciente);
        }

        //si el paciente existe lo retorna, sinó retorna null
        return miPaciente;
    }
```

De la misma manera se debe consultar el médico al que se le va a asociar la cita del paciente, según el planteamiento del problema se deberá ingresar el nombre del médico, por lo tanto creamos un método para realizar esta consulta, el cual

recibirá el nombre del médico a asociar y con este nombre hacemos la lógica necesaria para saber si existe o no y así asociarlo, si el médico existe se retorna para su almacenamiento.

ModeloDatos.java

```
84      }
85
86      public Medico consultarMedicoPorNombre(String nombreMedico) {
87
88          for (Medico medico : medicosMap.values()) {
89
90              if (medico.getNombre().equalsIgnoreCase(nombreMedico)) {
91                  return medico;//al retornar automaticamente termina el ciclo
92              }
93
94          }
95
96          //En caso de no encontrar ningún médico retorna null
97          return null;
98      }
```

Para poder almacenar la cita se crea una Lista de tipo **CitaMedica** la cual almacenará cada cita medica creada, en este caso se opta por usar un ArrayList dado que la cita será asociada a un paciente y este puede tener muchas citas, por lo que un hashMap no seria la mejor alternativa a menos que queramos asignarle un código único a cada cita.

ModeloDatos.java

```
1   package clases;
2
3   import java.util.ArrayList;
11
12  public class ModeloDatos {
13
14      HashMap<String, Paciente> pacientesMap;
15      HashMap<String, EmpleadoPlanilla> empleadosPlanillaMap;
16      HashMap<String, EmpleadoEventual> empleadosEventualMap;
17      HashMap<String, Medico> medicosMap;
18      ArrayList<CitaMedica> citasList;
19
20      public ModeloDatos() {
21          pacientesMap=new HashMap<String, Paciente>();
22          empleadosPlanillaMap=new HashMap<String, EmpleadoPlanilla>();
23          medicosMap=new HashMap<String, Medico>();
24          empleadosEventualMap=new HashMap<String, EmpleadoEventual>();
25          citasList=new ArrayList<CitaMedica>();
26      }
27
```

Finalmente en el Modelo de Datos agregamos el método que permite registrar la cita médica al almacenarla en la lista y posteriormente imprimiendo la información al llamar a la función correspondiente.

ModeloDatos.java

```
110
111     public void registrarCitaMedica(CitaMedica miCita) {
112         citasList.add(miCita);
113         System.out.println("Se ha registrado la cita exitosamente\n");
114         System.out.println(miCita.informacionCitaMedica());
115     }
```

Desde la clase Procesos armamos la estructura de los pasos mencionados anteriormente, en este caso veamos como se solicita la información del paciente para luego ser consultado y dependiendo de si existe o no se hace lo mismo con el médico hasta finalmente empezar a solicitar el resto de información.

Noten como en la linea 130 se crea la cita haciendo el llamado al constructor explicito y enviando los diferentes datos necesarios para crear el objeto (sin necesidad de hacer el llamado a los set y get) y por último en la linea 131 se registra la cita medica.

Procesos.java

```java
    private void registrarCitaMedica() {
        String documentoPaciente=JOptionPane.showInputDialog("Ingrese el documento del paciente");

        Paciente pacienteEncontrado=miModeloDatos.consultarPacientePorDocumento(documentoPaciente);

        if (pacienteEncontrado!=null) {
            String nombreMedico=JOptionPane.showInputDialog("Ingrese el nombre del médico");
            Medico medicoEncontrado=miModeloDatos.consultarMedicoPorNombre(nombreMedico);

            if (medicoEncontrado!=null) {
                String servicio=JOptionPane.showInputDialog("Ingrese el servicio o motivo de la consulta");
                String fechaConsulta=JOptionPane.showInputDialog("Ingrese la fecha de la consulta");
                String horaConsulta=JOptionPane.showInputDialog("Ingrese la hora de la consulta");

                String lugar="La cita será en el consultorio "+medicoEncontrado.getNumeroDeConsultorio();
                CitaMedica miCita=new CitaMedica(pacienteEncontrado, medicoEncontrado, servicio, fechaConsulta, horaConsulta, lugar);
                miModeloDatos.registrarCitaMedica(miCita);
            }else {
                System.out.println("El medico no se encuentra registrado en el sistema");
            }
        }else {
            System.out.println("El paciente no se encuentra registrado en el sistema");
        }

    }

```

Por último agregamos la opción correspondiente en el menú para imprimir la información de la cita médica así como el llamado al método encargado de imprimir las mismas.

Procesos.java

```java
    private void imprimirInformacion() {

        String menuImprimir="MENU IMPRESIONES\n";
        menuImprimir+="1. Listar Pacientes\n";
        menuImprimir+="2. Listar Empleados Eventuales\n";
        menuImprimir+="3. Listar Empleados Por Planilla\n";
        menuImprimir+="4. Listar Medicos\n";
        menuImprimir+="5. Listar Citas Programadas\n";
        menuImprimir+="Ingrese una opción\n";

        System.out.println("************************************************************");

        int opcion=Integer.parseInt(JOptionPane.showInputDialog(menuImprimir));

        switch (opcion) {
        case 1: miModeloDatos.imprimirPacientes();  break;
        case 2: miModeloDatos.imprimirEmpleadosEventuales();    break;
        case 3: miModeloDatos.imprimirEmpleadosPorPlanilla();   break;
        case 4: miModeloDatos.imprimirMedicos();    break;
        case 5: miModeloDatos.imprimirCitasMedicasProgramadas();    break;

        default: System.out.println("No exite esa opción");
            break;
        }
    }

```

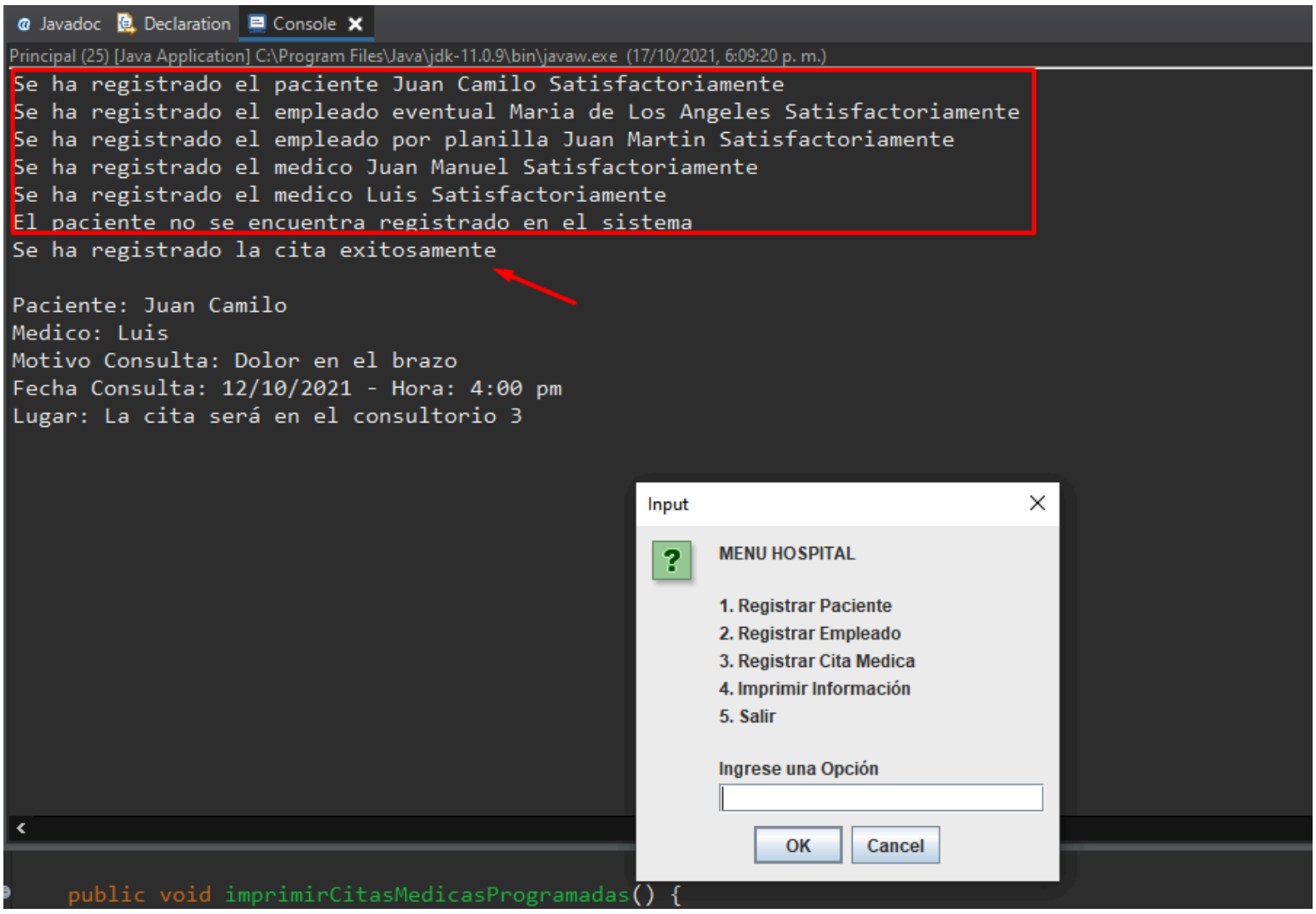
Y de esta manera en el Modelo de Datos imprimimos la información recorriendo nuestra lista de citas medicas.

ModeloDatos.java

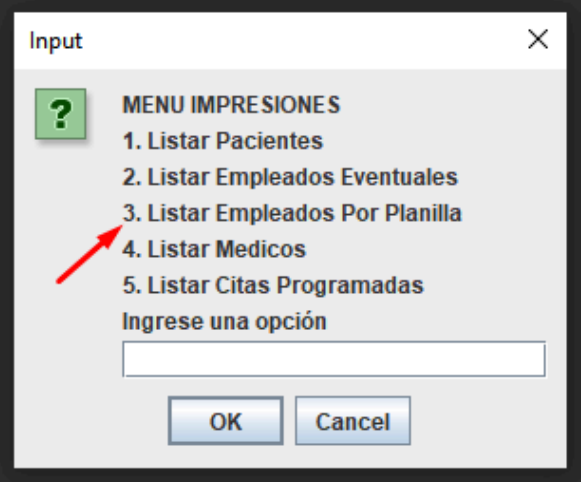
```java
    public void imprimirCitasMedicasProgramadas() {
        String msj="CITAS MEDICAS PROGRAMADAS\n";
        CitaMedica miCita=null;

        System.out.println(msj+"\n");

        if (citasList.size()>0) {
            for (int i = 0; i < citasList.size(); i++) {
                miCita=citasList.get(i);
                System.out.println(miCita.informacionCitaMedica());
            }
        }else {
            System.out.println("No existen citas programadas");
        }

    }

```

Ejemplo sistema en ejecución.

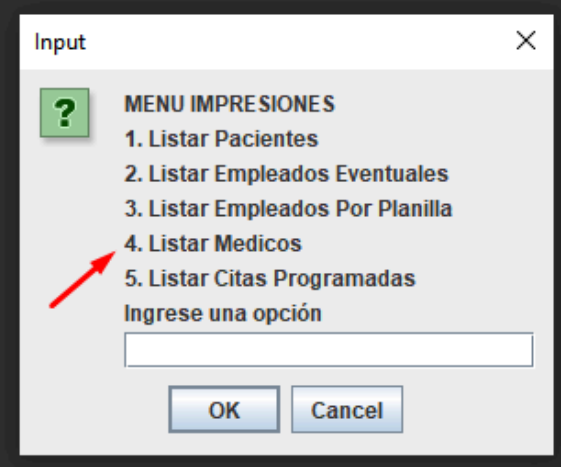
A continuación podemos ver algunas pantallas del funcionamiento del sistema.

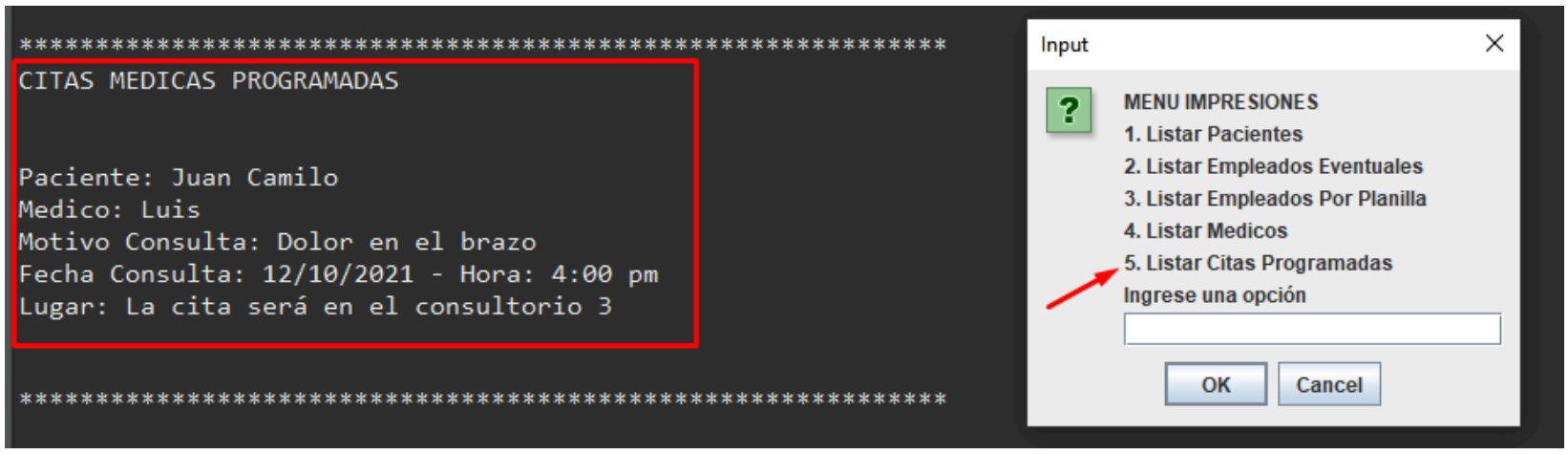


```
**********************************************************************
EMPLEADOS POR PLANILLA REGISTRADOS
Nombre: Juan Martin
Apellido: Zapata
Fecha de Nacimiento: 12/03/1980
Dirección: Mzna3 # 23- 09 B/ Colinas
Ciudad de Procedencia: Calarcá
Codigo De Empleado: 231
Numero De Horas Extras: 4
Fecha De Ingreso: 12/02/1999
Area: Archivo
Cargo: Gestor Documental

Salario Mensual: 3200000.0
Porcentaje Adicional Por Hora Extra: 2.0

**********************************************************************
```




```
**********************************************************************
MEDICOS REGISTRADOS
Nombre: Luis
Apellido: Perez
Fecha de Nacimiento: 12/09/1998
Dirección: Barrio Arboleda Mzna3
Ciudad de Procedencia: Pereira
Codigo De Empleado: 456
Numero De Horas Extras: 4
Fecha De Ingreso: 12/09/2020
Area: Cardiologia
Cargo: Medico General

Salario Mensual: 4000000.0
Porcentaje Adicional Por Hora Extra: 2.0

Especialidad: Cardiologo
Numero De Consultorio: 3

MEDICOS REGISTRADOS
Nombre: Juan Manuel
Apellido: Cardona
Fecha de Nacimiento: 12/02/1979
Dirección: Barrio Gibraltar
Ciudad de Procedencia: Armenia
Codigo De Empleado: 2341
Numero De Horas Extras: 3
Fecha De Ingreso: 21/09/2018
Area: Pediatria
Cargo: Medico General

Salario Mensual: 4000000.0
Porcentaje Adicional Por Hora Extra: 3.0

Especialidad: Pediatra
Numero De Consultorio: 5

**********************************************************************
```





Actividades Prácticas.

Después de construir el sistema, se espera que haya podido analizar el código para entender su funcionamiento ya que de eso depende el desarrollo de las siguientes actividades.

Teniendo en cuenta lo anterior, usted deberá complementar el sistema con las siguientes funcionalidades:

1. Cuando se intente imprimir información se debe validar si existen o no datos registrados (para cada opción de consulta disponibles)
2. Cuando se consulte empleados por planilla, debería mostrar también a los médicos registrados (se debe presentar un mensaje para que se note la diferenciación).
3. Valide para que el sistema no permita registro de pacientes, empleados o médicos duplicados.
4. Haga uso de excepciones (Try Catch) para controlar cuando una persona no ingresa un dato correcto (Por ejemplo cuando se ingresa un texto en el numero de historia clínica del paciente, el sistema se cae) esto lo puede hacer en los métodos de encapsulación.
5. Crear los métodos necesarios para consultar pacientes, empleados eventuales, empleados por planilla y médicos de forma individual mediante el ingreso del documento de identidad.

Instructor: Cristian David Henao Hoyos