IMA

Bienvenido a el programa de gestión de usuarios de IMA.

Introducción a la app:

En este caso nos encontramos con la obra social IMA, particularmente, con el programa que permite ingresar usuarios, modificarlos, darlos de baja y acceder a los datos de IMA. Se busca generar una app instructiva para el usuario para la facilidad de su uso.

Desarrollo de la app:

A continuación, explicaré paso a paso cómo funciona el programa:

1- En primer lugar, nos encontramos con el primer formulario FrmMenuIma. Este, en su evento load instancia al evento FrmSaludar, por lo cual, va a ser el primer formulario que vean.

FrmSaludar, da una bienvenida al usuario, presentando su nombre y el inicio de la jornada. Por un lado, utiliza la estructura DateTime.Now, para así obtener el horario de ingreso. Por otro lado, utiliza la clase Enviroment.UserName para obtener el nombre del usuario. Debería verse así:

2- Una vez cerrado el formulario FrmSaludar, entramos finalmente en el FrmMenuIma. En este formulario nos encontraremos con una diversidad de botones, los cuales nos brindaran la información necesaria para atender a un futuro cliente. Al mismo tiempo, el formulario en su evento load, si es que el archivo existe, lee un archivo xml, con el método Leer y carga una lista de usuarios en la ruta AppDomain.CurrentDomain.BaseDirectory/clientes.xml;

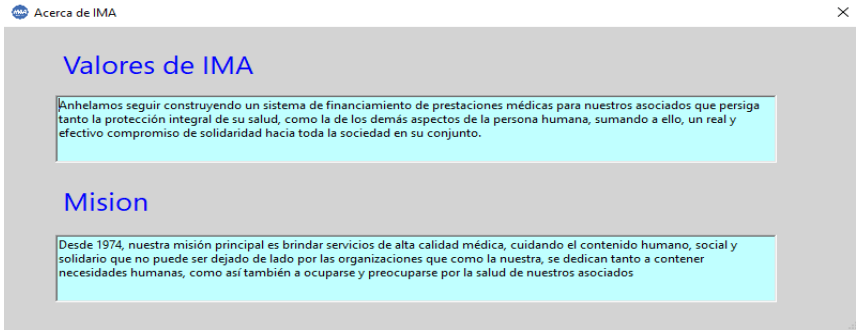
Debería verse de la siguiente manera:

Explicaremos que función cumple cada uno de los botones:

A- Acerca de IMA (btnInfoIma): Instancia el formulario FrmAcercaIma.

Luego, lo presenta mediante el método ShowDialog(). Este formulario posee los métodos MostrarValoresIMA y MostrarMisionIMA, los cuales retornan una string con información de IMA. Estos 2 métodos son llamados en el evento Load del formulario y cargados en el ritch text box correspondiente.

Debería verse así:



B- Planes (btnPlanes): Instancia el formulario FrmInfoPlanes. Luego, lo despliega utilizando el método ShowDialog().

Este formulario posee 4 botones. El primer botón, Plan clásico (btnPlanClasico) instancia un nuevo plan clásico y carga el rich text box con los datos del plan, mediante la sobrecarga del método ToString. El segundo botón, Plan Plus (btnPlanPlus) instancia un nuevo plan plus y carga el rich text box con los datos del plan, mediante la sobrecarga del método ToString. El tercer botón, Plan Premiun (btnPlanPremiun) instancia un nuevo plan premium y carga el rich text box con los datos del plan, mediante la sobrecarga del método ToString, Por último, el cuarto botón, cierra el formulario. Debería verse así:

C- El botón Normas de cobertura (btnCobertura): Instancia el formulario FrmNormasCobertura y lo despliega con el método ShowDialog().

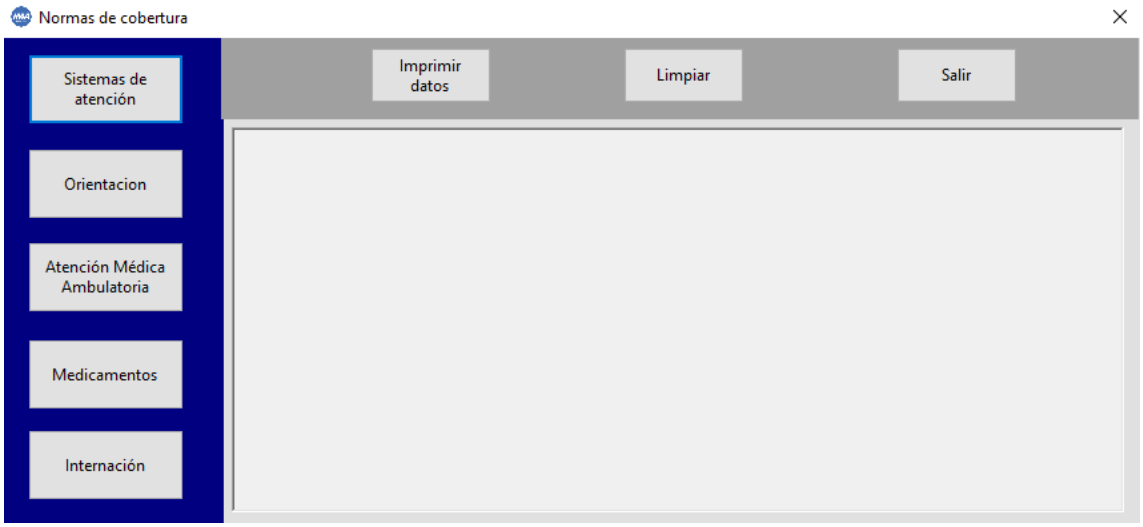
El formulario en el constructor por defecto instancia 6 rutas ubicadas en:

```
this.rutaSistAtencion = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"SistAtencion.txt";
this.rutaOrientacion = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"Orientacion.txt";
this.rutaAtencionAmbulatoria = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"AtencionAmbulatoria.txt";
this.rutaMedicamentos = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"Medicamentos.txt";
this.rutaInternacion = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"Internacion.txt";
this.rutaImprimir = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"datosCobertura.txt";
```

Los distintos eventos de click en los botones imprime en el rich text box los datos correspondientes leyendo el archivo txt que corresponda en cada caso.

El botón Imprimir datos, imprime la información con todos los datos en un archivo de tipo txt. Si el archivo existe, pisa los datos. Si el archivo no existe, lo crea.

El botón limpiar, carga el rich text box en string.Empty.
El botón salir cierra el formulario.
El formulario se ve de la siguiente manera:



D- El botón Acceder a contactos (btnContacto): Instancia el formulario FrmSucursales y lo despliega mediante el método ShowDialog().

Este formulario posee las 3 sucursales adheridas a IMA.

Los rich text box se cargan según el botón asignado con el método Mostrar de la clase Sucursales. Luego si se presiona imprimir datos, se guardan los datos de las 3 sucursales en un archivo en formato Json. La ruta del archivo es la siguiente:

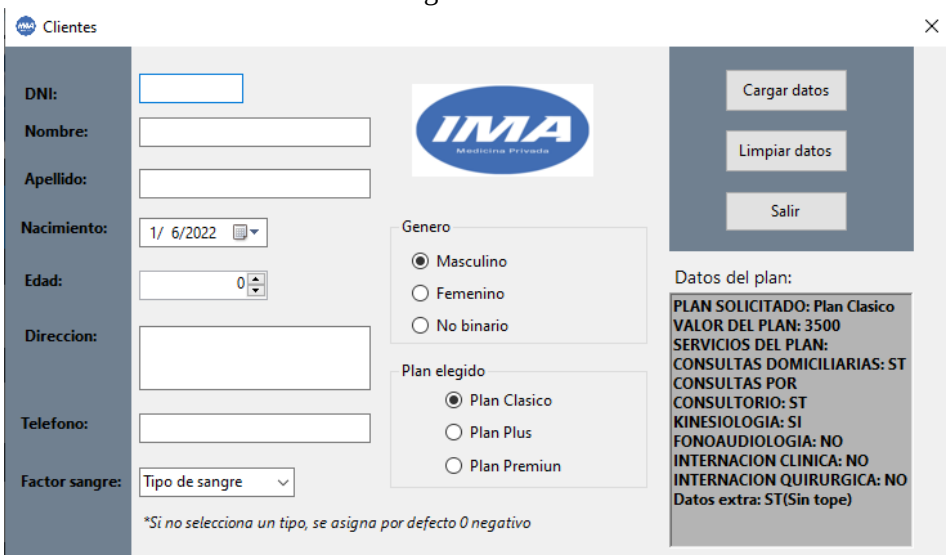
```
this.ruta = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"datosSucurales.json";
```

Al presionar el botón salir, se cierra el formulario.

E- El botón Alta usuario (btnAltaUsuario): Instancia el formulario FrmAltaCliente y los despliega mediante el método ShowDialog(). A su vez, asigna a la administración ubicada en FrmMenuIma con el valor del retorno de la propiedad de solo lectura de FrmAltaCliente, para actualizar la lista constantemente.

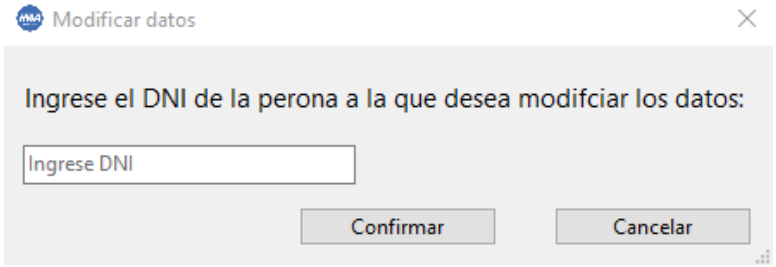
Una vez desplegado el formulario de para dar de alta la cliente, se deben llenar los campos correspondientes, los cuales están validados según el tipo dato.

El formulario se debería ver de la siguiente manera:



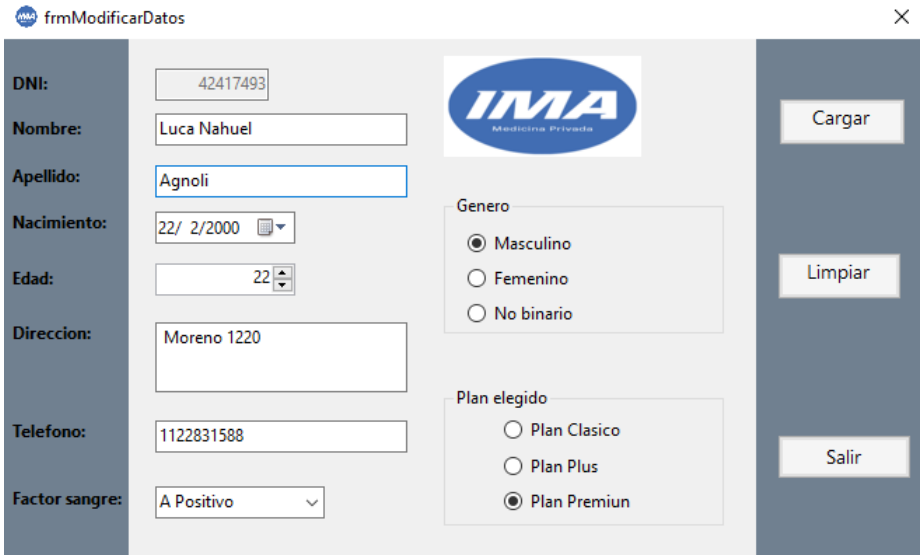
F- El botón Modificar Datos (btnModificarDatosUsuario) : Instancia el formulario FrmModificar y lo despliega con el método ShowDialog().

Se desplegará el siguiente formulario:



Este formulario en el textbox recibe el DNI y valida el ingreso de datos según corresponda. Si la persona existe, al presionar confirmar, instancia el formulario FrmModificarDatos y lo despliega mediante el metodo ShowDialog.

Si el texto está en blanco o la persona no existe, se arrojará una excepción según corresponda. Si todo sale bien, el siguiente formulario se ve de la siguiente forma:



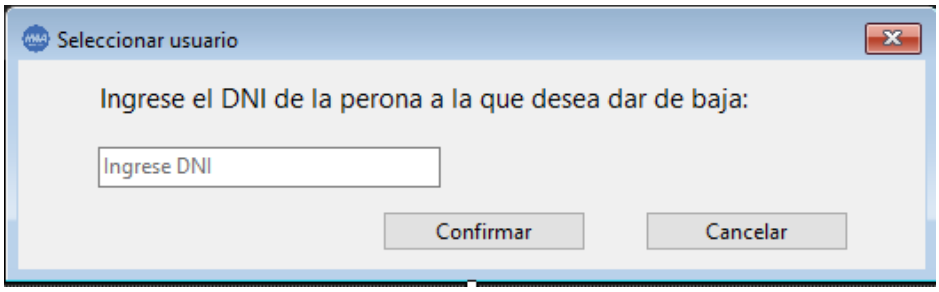
G- El formulario FrmModifcarDatos, se instancia ya cargado con los datos de la persona seleccionada. El DNI no se puede cambiar, los otros datos sí. Los text box están preparados para que no se puedan ingresar datos erróneos según corresponda.

Si se presiona confirmar se actualiza la lista de personas y se guardan los datos en el archivo clientes.xml. La ruta es la siguiente:

```
ruta = $"{AppDomain.CurrentDomain.BaseDirectory}" + @"clientes.xml";
```

H- El botón Baja usuario (btnBajaUsuario): Revisa que la lista este cargada, sino esta cargada tira un mensaje con un error. Si la lista esta cargada instancia un formulario FrmBajaUsuario y lo despliega con el metodo ShowDialog.

El formulario se debería ver de la siguiente manera:



Si el texto esta en blanco o la persona no existe, arroja una excepción según corresponda. A su vez, valida que la entrada de datos sea acorde a un DNI. Si presiona confirmar y la persona es valida en el sistema, se cambia el atributo estadoPlan de la clase persona a false. Los datos de la persona siguen en el sistema, pero ya no se encuentra activa, es decir, no se realiza una baja lógica, se pretende que el sistema sea realista conteniendo los datos de la persona en un estado inactivo.

I- Si se presiona el botón Clientes activos (btnMostrarClientesActivos): Se instancia el formulario FrmClientesActivos y se despliega con el metodo ShowDialog(). En este formulario podremos observar un listado de todos los clientes con el estadoPlan en true. A su vez, podemos filtrar para buscar a un cliente en especifico por DNI, la idea es que si alguien llama al operador/operadora pueda acceder a los datos de esta manera. Al mismo tiempo, si se desconoce el DNI, el formulario posee 2 botones para ordenar por apellido o DNI, en caso de ser necesario, para facilitar la búsqueda de la persona activa en el sistema. Este formulario SOLO muestra personas activas.

TEMAS APLICADOS

- Excepciones: Se crearon 3 excepciones para el programa. La primera, llamada MiExcepcionArchivoInvalido, se lanza cuando el archivo no existe o no posee una ruta valida. Mostrando su mensaje mediante la sobrecarga del método ToString. Ejemplo de implementación:

```
public void Guardar(string ruta, string contenido)
{
    if (ValidarSiExisteElArchivo(ruta) && ValidarExtension(ruta))
    {
        GuardarArchivo(ruta, contenido);
    }
    else
    {
        throw new MiExcepcionArchivoInvalido("La ruta o la extension no son validas", "Guardar", "PuntoTxt.cs");
    }
}
```

La segunda, llamada MiExcepcionCasillerosEnBlanco, arroja una excepción si los casilleros de los text boxs no puedan ser espacios en blanco. Ejemplo de implementación:

```
private void CargarTextoPorDni()
{
    Persona personaBuscada;
    if (!string.IsNullOrEmpty(txtFiltrarDni.Text) && !string.IsNullOrEmpty(txtFiltrarDni.Text))
    {
        personaBuscada = administracion.BuscarPorDni(administracion, Convert.ToInt32(txtFiltrarDni.Text));
        if (personaBuscada is not null && personaBuscada.EstadoPlan)
        {
            rtbClientesActivos.Text = personaBuscada.ToString();
        }
        else
        {
            throw new MiExcepcionPersonaNula($"La persona con DNI {txtFiltrarDni.Text} no se encuentra en el sistema o no esta activa", "FrmC
        }
    }
    else
    {
        throw new MiExcepcionCasillerosEnBlanco("Los campos de DNI no pueden estar vacios", "FrmClientesActivos.cs", "CargarTextoPorDni");
    }
```

La tercera y última, llamada MiExcepcionPersonaNula, arroja una excepción si la persona tiene valor nulo o no pertenece a la nula. Ejemplo de implementación:

```
private void AccionConfirmarClick()
{
    if (txtDniSolicitado.Text != string.Empty)
    {
        if(txtDniSolicitado.Text.Length == 8)
        {
            personaBuscada = administracion.BuscarPorDni(administracion, Convert.ToInt32(txtDniSolicitado.Text));
            if (personaBuscada is not null)
            {
                if (personaBuscada.EstadoPlan)
                {
                    FrmModificarDatos formModificar = new FrmModificarDatos(administracion, personaBuscada, listaPersonasImprimir);
                    formModificar.ShowDialog();
                }
                else
                {
                    MessageBox.Show($"La persona no se encuentra activa en el sistema ", "Error al modificar",
                     MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
            }
            else
            {
                throw new MiExcepcionPersonaNula("Persona nula, no se encuentra en el sistema", "FrmModificar.cs", "AccionConfirmarClick");
            }
```

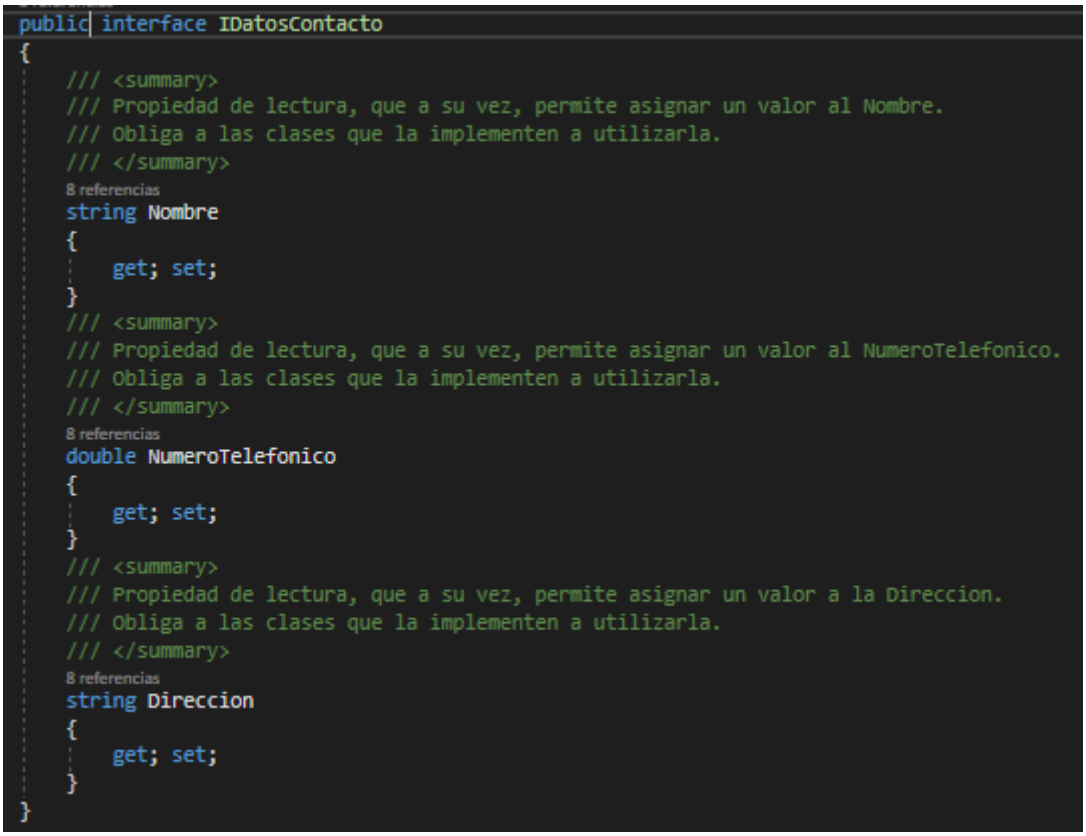
➢ Pruebas unitarias: Las pruebas unitarias validan que los archivos utilizados en formato json, txt y xml tengan una ruta y extensión valida. Ejemplos de implementación en el caso de formato json:

```
[TestMethod]
| 0 referencias
public void PuntoJson_DeberiaRetornarTrue_SiLaExtensionEsTipoJson()
{
    //Arrange
    string ruta = $"{Environment.GetFolderPath(Environment.SpecialFolder.Desktop)}\\MiArchivo.json";
    PuntoJson<string> json = new PuntoJson<string>();
    bool rutaValidacion = json.ValidarExtension(ruta);
    // string
    Assert.IsTrue(rutaValidacion);
}

[TestMethod]
[ExpectedException(typeof(MiExcepcionArchivoInvalido))]
| 0 referencias
public void PuntoJson_DeberiaLanzarMiExcepcionArchivoInvalido_SiElArchivoNoEsExiste()
{
    //Arrange
    string ruta = $"{Environment.GetFolderPath(Environment.SpecialFolder.Desktop)}\\ArchivoArchivo.json";
    PuntoJson<string> json = new PuntoJson<string>();
    if(!json.ValidarSiExisteElArchivo(ruta))
    {
        throw new MiExcepcionArchivoInvalido("Archivo invalido", "Error", "Test puntoJson");
    }
}
```

➢ Tipos genéricos: Las clases PuntoTxt, PuntoJson y PuntoXml son genéricas, para poder imprimir cualquier tipo de dato que se requiera. En este caso se utiliza para imprimir personas, strings y contactos. Se crearon genéricas en caso de a futuro, para el tp4, sea necesario imprimir mayor cantidad de datos de distintas clases.

➢ Interfaces: La interfaz implementada se llama IDatosContacto, obliga a las clases que la implementen a utilizar los campos contenidas en ella. Es utilizada en las clases contactoInterno, Persona y Sucursales.



```
public interface IDatosContacto
{
    /// <summary>
    /// Propiedad de lectura, que a su vez, permite asignar un valor al Nombre.
    /// Obliga a las clases que la implementen a utilizarla.
    /// </summary>
    8 referencias
    string Nombre
    {
        get; set;
    }
    /// <summary>
    /// Propiedad de lectura, que a su vez, permite asignar un valor al NumeroTelefonico.
    /// Obliga a las clases que la implementen a utilizarla.
    /// </summary>
    8 referencias
    double NumeroTelefonico
    {
        get; set;
    }
    /// <summary>
    /// Propiedad de lectura, que a su vez, permite asignar un valor a la Direccion.
    /// Obliga a las clases que la implementen a utilizarla.
    /// </summary>
    8 referencias
    string Direccion
    {
        get; set;
    }
}
```

➢ Archivos y serialización: Se imprimen y leen datos en formato txt, xml y json. Se creo una clase padre Archivo, de la cual heredan las hijas. Estas son, Puntojson, PuntoXml y PuntoTxt. Según el caso lo amerite, se imprimen y leen datos en dichos formatos. Ejemplo de la clase PuntoXml:

```
3 referencias
public void Guardar(string ruta, T contenido)
{

    if (ValidarSiExisteElArchivo(ruta) && ValidarExtension(ruta))
    {
        Serializar(ruta, contenido);
    }
    else
    {
        throw new MiExcepcionArchivoInvalido("La ruta o la extension no son validas", "Guardar", "PuntoXml.cs");
    }
}

/// <summary>
/// Si el archivo no existe, guarda los datos y crea una carpeta en la ruta seleccionada.
/// Caso contrario arroja MiExcepcionArchivoInvalido.
/// </summary>
/// <param name="ruta">ruta del archivo</param>
/// <param name="contenido">contenido del archivo</param>
0 referencias
public void GuardarComo(string ruta, T contenido)
{
    if (ValidarExtension(ruta))
    {
        Serializar(ruta, contenido);
    }
    else
    {
        throw new MiExcepcionArchivoInvalido("La ruta o la extension no son validas", "GuardarComo", "PuntoXml.cs");
    }
}
```

Archivos cargados: