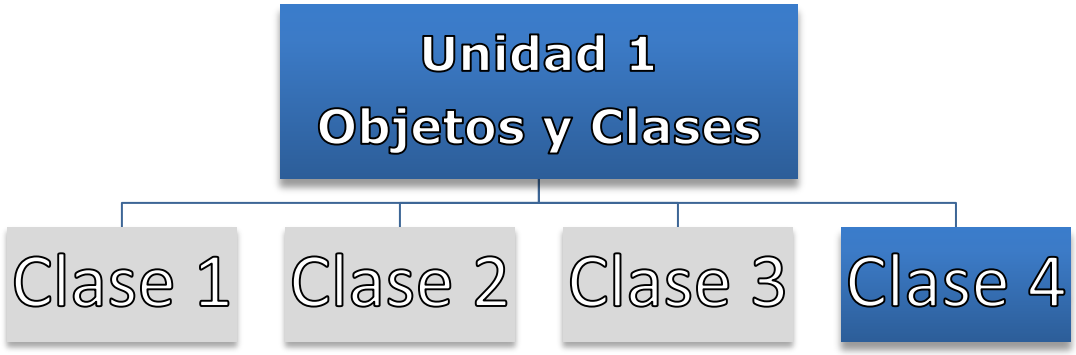
# PROGRAMACIÓN ORIENTADA A OBJETOS




Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci

## Presentación

La clase 4 corresponde a la unidad 1 de la asignatura. En esta unidad presentaremos la forma de utilizar adecuadamente constructores, finalizadores y destructores.

Lo invitamos a incursionar en los temas propuestos, ya que le brindarán no sólo un aporte tecnológico a su formación, sino que podrá mejorar su perspectiva profesional sobre la visión que posee acerca de *cómo desarrollar software*.

Los siguientes **contenidos** conforman el marco teórico y práctico de esta unidad. A partir de ellos lograremos alcanzar el resultado de aprendizaje propuesto: En negrita encontrará lo que trabajaremos en la clase 3.

- El modelo orientado a objetos.
- Jerarquías "Es - Un" y "Todo - Parte".
- Concepto de Clase y Objeto.
- Características básicas de un objeto: estado, comportamiento e identidad.
- Ciclo de vida de un objeto.
- Modelos. Modelo estático. Modelo dinámico. Modelo lógico. Modelo físico.
- Concepto de análisis diseño y programación orientada a objetos.
- Conceptos de encapsulado, abstracción, modularidad y jerarquía.
- Concurrencia y persistencia.
- Concepto de clase.
- Definición e implementación de una clase
- Campos y Constantes
- Propiedades. Concepto de Getter() y Setter(). Propiedades de solo lectura. Propiedades de solo escritura. Propiedades de lectura-escritura. Propiedades con indizadores. Propiedades autoimplementadas. Propiedades de acceso diferenciado.
- Métodos. Métodos sin parámetros. Métodos con parámetros por valor. Métodos con parámetros por referencia. Valores de retorno de referencia.
- Sobrecarga de métodos.
- **Constructores. Constructores predeterminados. Constructores con argumentos.**
- **Finalizadores.**

- **Clases anidadas.**

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta clase. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor.

## **1.** Constructores y Finalizadores

### Lecturas requeridas

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/constructors

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/finalizers

https://docs.microsoft.com/es-es/dotnet/api/system.idisposable?view=net-6.0

## **2.** Clases anidadas

### Lecturas requeridas

Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo IX. Puntos 9.7 y 9.9

## 1. Constructores y Destructores

**Constructores.**

Cada vez que se crea un objeto se llama a su constructor, que es el que se ha definido en la clase que se instancia para obtenerlo. Una clase puede tener varios constructores que toman argumentos diferentes (deben poseer distinta firma). Los constructores permiten al programador establecer valores predeterminados, inicializar valores del objeto, permitir la agregación con cntención física y escribir código flexible y fácil de leer.

Si no se proporciona un constructor para la clase, C# creará uno de manera predeterminada que cree instancias del objeto y establezca las variables miembro en los valores predeterminados.

Un constructor es un método cuyo nombre es igual que el nombre de su clase. La firma del método incluye solo el nombre del método y su lista de parámetros. No incluye un tipo de valor devuelto. En el ejemplo **Ej0022** se muestra el constructor de una clase denominada Alumno.

```
public class Alumno
{
    private string Nombre;
    private string Apellido;

    0 referencias
    public Alumno(string pNombre, string pApellido)
    {
        this.Nombre = pNombre;
        this.Apellido = pApellido;
    }
}
```

Ej0022

Si un constructor puede implementarse como una instrucción única, puede usar una definición del cuerpo de expresión. En el ejemplo **Ej0023** define una clase **Lugar** cuyo constructor tiene un único parámetro de cadena denominado **Nombre**.

```
public class Lugar
{
    private string Vnombre;
    0 referencias
    public Lugar(string pNombre) => Vnombre = pNombre;
    0 referencias
    public string Nombre
    {
        get => Vnombre;
        set => Vnombre = value;
    }
}
```

Ej0023

Un constructor que no toma ningún parámetro se denomina ***constructor predeterminado***. Los constructores predeterminados se invocan cada vez que se crea una instancia de un objeto mediante el operador **new** y no se especifica ningún argumento en **new**.

Se puede impedir crear una instancia de una clase convirtiendo el constructor en privado, como se muestra en el ejemplo **Ej0024**.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        // En la línea siguiente se puede observar como intentar instanciar
        // la clase Pi de error debido al nivel de protección que posee la misma
        Pi valor1 = new Pi();
        // En la línea siguiente se observa como se puede obtener el valor de la
        // propiedad ValorPi.
        double valor2 = Pi.ValorPi;
    }
}
4 referencias
class Pi
{
    // Private Constructor:
    1 referencia
    private Pi() { }
    public static double ValorPi = Math.PI;  //3.14159.....
}
```

Ej0024

Más adelante se abordará una relación entre clases denominada **herencia.** Sin profundizar en este tema ahora, solo se mencionará que esta relación se da cuando una clase, denominada **clase base**, le hereda su estructura y comportamiento a otra clase denominada **sub clase** o **clase derivada.**

Si esta relación se da, puede suceder que se desea acceder al **constructor** de la **clase base** desde la **sub clase**. Esto puede darse por varios motivos, pero quizá uno de los que más justifica esta práctica, sea reutilizar el código que está programado en el **constructor** de la **clase base**.

En el siguiente ejemplo **Ej0025**, la clase **Empleado hereda** de la clase **Persona**. Esto transforma a la clase **Persona** en una **super clase** y a la clase **Empledo** en una **sub clase.**

La **sub clase** implementa la propiedad **legajo** y **hereda** de la **super clase** las propiedades **nombre** y el **apellido.** El constructor de la **super clase** permite ingresar el nombre y apellido cuando se instancia el objeto. Ese código ya se encuentra programado. No tiene sentido volver a programar esto en la **sub clase.** Es por ello que la sub clase en su constructor, llama al constructor de la **super clase** (base(pNombre, pApellido)). Esto permite que la inicialización del nombre y el apellido se pueda aprovechar desde la **sub clase** aunque esté programado en la **super clase.**

```
public partial class Form1 : Form
{
    1 referencia
    public Form1() {InitializeComponent();}
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Empleado Ve = new Empleado("Juan","Perez","L0001");
        MessageBox.Show(Ve.Datos());
        Application.Exit();
    }
}
2 referencias
public class Persona
{
    string Vnombre = "";
    string Vapellido = "";
    1 referencia
    public Persona(string pNombre, string pApellido)
    { this.Vnombre = pNombre; this.Vapellido = pApellido; }
    1 referencia
    public string Nombre { get { return this.Vnombre; }  set { this.Vnombre = value; } }
    1 referencia
    public string Apellido { get { return this.Vapellido; } set { this.Vapellido = value; } }
}
3 referencias
public class Empleado : Persona
{
    string Vlegajo = "";
    1 referencia
    public Empleado(string pNombre, string pApellido,string pLegajo)
        : base(pNombre, pApellido) { this.Vlegajo = pLegajo; }
    1 referencia
    public string Legajo { get { return this.Vlegajo; } set { this.Vlegajo = value; } }
    1 referencia
    public string Datos() { return "Legajo: " + this.Legajo + "\nNombre: " + this.Nombre +
                                   "\nApellido: " + this.Apellido; }
}
```

Ej0025

**Destructores.**

Los **destructores** también denominados **finalizadores**, se usan para destruir instancias de clases. Una clase solo puede tener un finalizador. No se pueden

heredar ni sobrecargar y tampoco se puede llamar, los **finalizadores** se invocan automáticamente. Un **finalizador** no permite modificadores ni tiene parámetros.

El finalizar es el último método que se ejecuta cuando el objeto que existe en la memoria por algún motivo de destruye. Es destrucción puede ocurrir debido a que se pierde el ámbito de la variable que lo apunta o simplemente no es apuntado por ninguna variable, aspectos que hacen que el objeto quede como basura en la memoria. Es esos casos automáticamente se ejecuta el **finalizador.**

En el ejemplo Ej0026 se puede observar como al crear el objeto se ejecuta el **constructor** y como al finalizar la aplicación se ejecuta el **finalizador.**

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }

    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Moto M = new Moto();
        Application.Exit();
    }
}
4 referencias
public class Moto
    {
    // Constructor
    1 referencia
    public Moto(){ MessageBox.Show("Se ha creado un Moto !!!"); }
    // Finalizador
    0 referencias
    ~Moto() { MessageBox.Show("Se ha destruido una Moto !!!"); }
    }
```

Ej0026

Cuando existe herencia, si se ejecuta el **finalizador** de una **sub clase,** al finalizar este llama al **finalizador** de la **super clase** de la cual hereda. Si existiera un herarquía de herencia de varios niveles, la invocación será desde la clase más derivada hacia la clase más específoca.

En el ejemplo Ej0027 se puede observar lo enunciado.

```
    public Form1()
    {
        InitializeComponent();
    }

    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        MotoEspecial M = new MotoEspecial();
    }
}
2 referencias
public class Moto
{
    0 referencias
    ~Moto() { MessageBox.Show("Se ha destruido una Moto !!!"); }
}
3 referencias
public class MotoEspecial : Moto
{
    0 referencias
    ~MotoEspecial() { MessageBox.Show("Se ha destruido una MotoEspecial !!!"); }
}
```

Ej0027

## 2. Clases anidadas.

Una clase anidada es una clase definida dentro de otra clase. Por defecto la clase anidad tendre un modificador de acceso privado a pesar que se le puede modificar.

El uso de las clases anidadas entre otras cosas sirve para organizar clases que en general se crean con el objetivo de darle servicios a quien la contiene o la naturaleza de su existencia está muy ligada a ella.

Si la clase contenida se define como privada solo se podrá utilizar dentro de la clase contenedora.

En el ejemplo **Ej0019** se puede observar como la clase **Contenedor** anida la clase **Contenido** cuyo modificdor de acceso es **private**. Esto causa que dentro de la clase **Contenedor,** en el método denominado **Uso,** se pueda instanciar un objeto del tipo **Contenido** sin inconvenientes. Pero si observamos el método **Form1_Load** de la clase **Form1,** encontraremos que al intentar declarar la variable **VarC** del tipo **Contenedor.Contenido**, no solo arroja un error la definición de la variable, sino que también el intento de instanciar un objeto de ese tipo. Esto es debido a la visibilidad (private) que posee la clase **Contenido**.

Así como en este caso se ha aplicado el modoficador de acceso **private**, también se pueden utilizar los otros modificadores de acceso: public, protected, internal, protected internal o private protected.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }

    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
2 referencias
public class Contenedor
{
    0 referencias
    void Uso() { Contenido C = new Contenido(); }
    4 referencias
    private class Contenido
    {

    }
}
```

'Contenedor.Contenido' no es accesible debido a su nivel de protección

Ej0019

Si la clase contenida se define como pública también se la podrá utilizar fuera de la clase contenedora. A continuación se observa esto en el ejemplo **Ej0020** desarrollado con esa modificación en base al **Ej0019**. Se puede observar como los errores observados en el ejemplo anterior han desaparecido.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
2 referencias
public class Contenedor
{
    0 referencias
    void Uso() { Contenido C = new Contenido(); }
    4 referencias
    public class Contenido {}
}
```

Ej0020

El tipo anidado o interno puede tener acceso al tipo contenedor o externo. Para tener acceso al tipo contenedor, se lo puede pasar como un argumento al constructor del tipo anidado. El tema constructores se analizará más adelante en el material. El ejemplo **Ej0021** muestra como hacer esto.

Un tipo anidado tiene acceso a todos los miembros que estén accesibles para el tipo contenedor. Puede tener acceso a los miembros privados y protegidos del tipo contenedor, incluidos los miembros protegidos heredados.

```
    private void Form1_Load(object sender, EventArgs e)
    {
        Contenedor.Contenido VarC = new Contenedor.Contenido();
    }
}
4 referencias
public class Contenedor
{
    1 referencia
    void Uso()
    { Contenido C = new Contenido(this); }
    6 referencias
    public class Contenido
    {
        Contenedor C;
        1 referencia
        public Contenido() { }
        1 referencia
        public Contenido(Contenedor pContenedor)
        { C = pContenedor; C.Uso(); }
    }
}
```

Ej0021

- NOTA: TODO EL CÓDIGO QUE SE UTILIZA EN LAS EXPLICACIONES LO PUEDE BAJAR DEL **MÓDULO RECURSOS Y BIBLIOGRAFÍA**.

## Actividades asincrónicas

### Guía de preguntas de repaso conceptual

1. ¿Para qué se utilizan los constructores?
2. ¿qué modificadores se le pueden colocar a los constructores y cómo los afecta en su funcionamiento?
3. ¿Para qué se utilizan los destructores?
4. ¿Qué diferencia conceptual existe entre un finalizador y un destructor?
5. ¿Cómo desarrollaría un finalizador?
6. ¿Cómo desarrollaría un destructor?
7. ¿Quién invoca al finalizador?
8. ¿Quién invoca al destructor?
9. ¿Una clase puede poseer un finalizador y un destructor?
10. ¿Una clase puede tener muchos constructores?

### Guía de ejercicios

1. Desarrollar un programa que posea una clase que aplique un destructor que finalice el ciclo de vida de un objeto que se instranció en el constructor.

UAI

Universidad Abierta Interamericana

# TRABAJO PRÁCTICO INTEGRADOR Nro 1

## PROGRAMACIÓN ORIENTADA A OBJETOS

Profesor: DARIO CARDACCI

# TRABAJO PRACTICO NÚMERO 1

Nos solicitan crear un programa que maneje una lista de alumnos.
Un alumno posee:

**Características**

| | | |
|---|---|---|
| Legajo | int | lectura/escritura |
| Nombre | string | lectura/escritura |
| Apellido | string | lectura/escritura |
| Fecha_Nacimiento | date | solo escritura |
| Fecha_Ingreso | date | solo escritura |
| Edad | int | solo lectura |
| Activo | boolean | lectura escritura |
| Cant_Materia_Aprobadas | int | solo escritura |

**Métodos**

| | | |
|---|---|---|
| Antigüedad | int | Retorna la cantidad de años/meses/días que hace que el alumno asiste a la institución. El método posee un parámetro que indica en que unidad (años, meses, días) se desea el retorno. Para el cálculo se considera solo año/mes cumplido, que hace que el alumno acude a la institución. Par determinar el número de años se considera la fecha actual y la fecha de ingreso. |
| Materias_No_Aprobadas | int | Retorna la cantidad de materias no aprobadas a la fecha sabiendo que la carrera posee 36 Materias y considerando la cantidad de materias aprobadas. |
| Edad_De_Ingreso | int | Retorna la edad a la que el alumno ingresó a la institución considerando la fecha de nacimiento y la fecha de ingreso. |

**Constructores:**

1. Constructor sin parámetros
2. Constructor con todos los parámetros que permiten inicializar las propiedades.

**Finalizadores:**

1. Que cuando el objeto queda liberado muestre una leyenda indicando el Legajo, Nombre y Apellido del Alumno.

Nos solicitan que la GUI (interfaz gráfica del usuario) permita visualizar la lista de los alumnos ingresados en una lista del tipo DataGridView.

La GUI debe tener botones para:

Agregar un alumno a la lista.
Borrar el alumno seleccionado de la lista.
Modificar el alumno seleccionado de la lista.

La GUI debe tener cajas de texto para ver del alumno seleccionado en la grilla. Estas cajas de texto se deben actualizar cuando el usuario de desplaza por las filas de la grilla con el cursor o les hace click

Antigüedad
Materias_No_Aprobadas
Edad_De_Ingreso

Recursos a considerar para la resolución:

1. **Tema** Listas de programación 1. **Sugerencia**: Usar **List(of…)**

2. **Tema** Clases, Instancias, Propiedades, Métodos, Constructores, Finalizadores de la Unidad I de Programación Orientada a Objetos. **Sugerencia**: Ver el Orientador y la bibliografía recomendada.

3. **Tema** DatagridView. Grilla vista el programación 1. **Sugerencia**: https://docs.microsoft.com/es-es/dotnet/framework/winforms/controls/datagridview-control-windows-forms