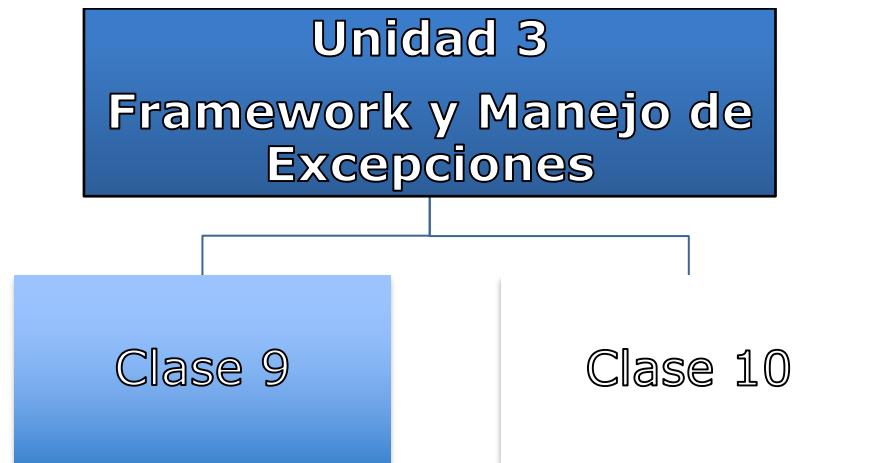


---

## PROGRAMACIÓN ORIENTADA A OBJETOS

---

---



---

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

---

---

## Presentación

---

En esta unidad abordaremos los temas referidos a los Frameworks y en particular nos concentraremos en uno denominado .NET Framework.

En particular haremos énfasis en las partes constitutivas del Framework, los esquemas de compilación y la administración de los objetos en la memoria.

El uso de Framework en los desarrollos actuales es muy importante ya que permite obtener soluciones fiables en tiempos razonables de producción.

Esperamos que luego de analizar estos temas Ud. pueda percibir el aporte que los temas tratados le otorgan a los desarrollos de los sistemas de información.

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad logre:

- Comprender la noción de framework a través del análisis de sus particularidades.
- Distinguir los distintos tipos de frameworks.
- Analizar y reconocer las particularidades del framework .NET para poder utilizarlas en el desarrollo de sistemas

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 9.

- **Concepto de frameworks. Elementos de un framework. Tipos de frameworks.**
- **Arquitectura de .NET. Interoperatividad entre .NET y COM.**
- **Código administrado y no administrado.**
- **Common Language Runtime CLR.**
- **Lenguaje intermedio IL.**
- **El compilador Just-in-Time (JIT).**
- **Concepto de assembly.**
- **Administración de la memoria en .NET. El Garbage Collector.**
- Manejo de excepciones. Control de excepciones. El objeto Exception. La instrucción Try – Catch – Finally. La instrucción Throw.
- Depuración de aplicaciones. Herramientas de depuración. Análisis del comportamiento de las aplicaciones.

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. 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 tutor.

## 1. Introducción a los Frameworks.

---

### Lectura requerida

<https://docs.microsoft.com/en-us/dotnet/framework/>

---

### Material multimedia requerido

- Material multimedia Frameworks.pptx

## 2. .NET Framework

---

### Lectura requerida

<https://docs.microsoft.com/en-us/dotnet/framework/>

---

### Material multimedia requerido

- Material multimedia .NET Frameworks.pptx

## 3. Administración de Memoria

---

### Lectura requerida

- <https://docs.microsoft.com/es-es/dotnet/standard/garbage-collection/memory-management-and-gc>  
[Material multimedia Administración de Memoria.pdf](#)

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

## 1. Introducción a los Frameworks.

Un **framework** es un marco de trabajo que ofrece a quien lo utiliza, una serie de herramientas que le facilitan la realización de tareas.

Puede contener librerías de clases, documentación, ayuda, ejemplos, tutoriales, pero sobre todo "*contiene experiencia sobre un dominio de problema específico*".

Un **framework** tiene como objetivos tres cosas: normalizar, estandarizar y reutilizar la experiencia. Sabemos lo importante que es la normalización de datos y procesos en cualquier aplicación. Los usuarios pueden manejar su información informalmente, en su propia memoria, tenerla duplicada, con incoherencias y omisiones, etc. Muchas veces parece que la única elección importante es la tecnología concreta a utilizar (lenguaje de programación, gestor de bases de datos, etc.) pero, a partir de ahí, cada programador puede crear, aplicando su imaginación y experiencia, que puede ser mucha o poca, su propia manera de generar código fuente.

La pregunta que surge es ¿por qué permitir ese "desorden" y "falta de administración" en un desarrollo, si en realidad están demostradas las bondades de estructurar y normalizar la información y el código generado?

Los **frameworks** intentan dotarnos de herramientas que permitan hacer realidad las bondades de estructurar y normalizar la información y el código generado. Los **frameworks** no necesariamente están ligados a un lenguaje concreto. Como ejemplo, podemos mencionar el **.NET framework**, sobre el cual se puede desarrollar en distintos lenguajes (VB.NET, C#, J#, etc). También es posible que el **framework** defina una estructura para una aplicación completa, o bien sólo se centre en un aspecto de ella.

Un párrafo aparte se le debe dedicar al concepto "*reutilizar la experiencia*". Un **framework** se desarrolla a partir del conocimiento profundo de algún dominio de problema, e intenta estandarizar y normalizar aquellas cosas que son comunes a todos esos tipos de problemas. Por ejemplo, si construimos un **framework** que sirve para el desarrollo de aplicaciones

financieras, este contendrá herramientas y elementos que son el resultado de haber capturado mucha experiencia en esa área en particular. Seguramente esas herramientas y elementos servirán para la mayoría de los sistemas de análisis financiero.

Los elementos básicos de un Framework son: "los puntos congelados o Frozen-Spots", "los puntos calientes o Hot Spots" y además poseen una característica denominada "*Inversión de Control*".

Los puntos congelados son aquellas partes del código provistas por el **framework**. Los puntos calientes son las partes de código que coloca el programador e interactúan con el **framework**. El código colocado en los puntos calientes es lo que el programador considera "*su programa*" pero *en realidad para que todo funcione correctamente al código generado por él se le suman muchos fragmentos proporcionados por el framework*.

Normalmente cuando se trabaja con un **framework**, este se encuentra funcionando desde antes que el programador escriba su primera línea de código. En muchos casos lo asiste en la generación de código, control de errores etc. Se podría decir que el **framework** está en funcionamiento mucho antes que el programa escrito por el programador, es decir "*lo controla*". A esta característica se la denomina "**inversión de control**".

En general se puede optar por utilizar un **framework** desarrollado por terceros o bien desarrollar uno propio. En caso de desarrollar uno propio se deberá seguir una secuencia de etapas constructivas.

Las tres etapas principales del desarrollo del **framework** son: análisis del dominio, diseño del **framework**, y la "instantiación" del **framework**.

El análisis del dominio procura descubrir los requisitos del dominio y los posibles requerimientos futuros. Para completar los requerimientos sirven las experiencias previamente publicadas, los sistemas similares de software existentes, las experiencias personales, y los estándares considerados. Durante el análisis del dominio, los puntos calientes y los puntos congelados se destapan parcialmente.

La fase del diseño del **framework** define sus abstracciones. Se modelan los puntos calientes y los puntos congelados (quizás con diagramas de

UML “Lenguaje de Modelado Unificado” - *Unified Modeling Language* -) , y la extensión y la flexibilidad propuesta en el análisis del dominio se esboza en líneas generales. Según lo mencionado arriba, los modelos del diseño se utilizan en esta fase.

Un **framework** se puede también clasificar según su extensibilidad, puede ser utilizado como una **caja blanca** o **caja negra**.

En los **frameworks** de **caja blanca** se crean de nuevas clases por herencia o composición.

En los **frameworks** de **caja negra** se producen instancias usando código o scripts de configuración. Después de la configuración, una herramienta automática de instancia crea las clases. En los **framework** de caja negra no se requiere que el usuario sepa los detalles internos del framework.

Entre las ventajas de utilizar **frameworks** podemos decir que el programador no necesita plantearse una estructura global de la aplicación, sino que el **framework** le proporciona un esqueleto que hay que “rellenar”. También proporciona mejores formas para definir y estandarizar, lo que redundará en ahorro de tiempo.

## 2 .NET Framework

El **.NET Framework** provee las herramientas necesarias en run-time y compile-time para construir y ejecutar aplicaciones basadas en .NET.

La plataforma **.NET Framework** debe ejecutarse sobre un Sistema Operativo. En la actualidad forma parte del sistema operativo y viene provista por este.

El **.NET Framework** expone servicios de aplicaciones a través de clases de la **.NET Framework Classs Library**.

La **Common Language Runtime** simplifica el desarrollo de aplicaciones, provee un entorno de ejecución robusto y seguro, soporta varios lenguajes y simplifica el despliegue y la administración.

La **CLR** es un entorno administrado (managed), en el cual los servicios comunes, como garbage collection y seguridad, son provistos automáticamente.

La librería de clases de .NET Framework (.NET Framework Class Library) expone características en tiempo de ejecución y provee otros servicios útiles para todos los desarrolladores. Las clases simplifican el desarrollo basado en .NET. Los desarrolladores pueden extenderlas creando sus propias librerías de clases. Las librerías de clases base son implementadas por el .NET Framework. Todas las aplicaciones (web, windows, web services) acceden a las mismas clases base. Estas están almacenadas en namespaces. Los diferentes lenguajes acceden a las mismas librerías.

El framework tambien provee **ADO.NET** como soporte para el acceso a datos en sus modalidades conectado y desconectado.

Por otro lado **XML Web Services** nos pone a disposición componentes Web programables que pueden ser compartidos entre aplicaciones, sobre Internet o una intranet. El .NET Framework provee herramientas y clases para desarrollo, testeo y distribución de XML Web Services.

El .NET Framework soporta tres **tipos de Interfaces** de usuario: **Web Forms, Windows Forms, Aplicaciones de Consola**. En este curso básicamente utilizaremos Windows Forms.

En cuanto a la programación de aplicaciones cualquier lenguaje que sea acorde a la **Common Language Specification (CLS)** puede ejecutarse sobre la CLR. En .NET Framework, Microsoft provee Visual Basic, Visual C++, Visual C#, Visual J#. Terceros nos pueden proveer de nuevos lenguajes.

El .NET Framework constituye las bases sobre las que, tanto aplicaciones como servicios, son ejecutadas y construidas. La naturaleza unificada del .NET Framework permite que cualquier tipo de aplicación sea desarrollada

mediante herramientas comunes, haciendo la integración mucho más simple. Dos de los elementos más importantes en el .NET Framework son:

**El CLR (Common Language Runtime)**

**La BCL (Base Class Library)**

El **Common Language Runtime** es el corazón del .NET Framework. Los compiladores y herramientas exponen funcionalidades en tiempo de ejecución y permiten escribir código con el beneficio de un entorno de ejecución administrado. El código que se desarrolla con un compilador de lenguaje que trabaja con el runtime se llama código administrado (**managed code**). Esto permite beneficios como la integración y el manejo de excepciones entre distintos lenguajes, seguridad mejorada, versionamiento y soporte para la implementación del sistema, además de un modelo simplificado para la interacción de componentes y servicios de debugging.

Para permitir al runtime proveer servicios al código administrado, los compiladores deben emitir **metadatos** (información adicional) que describen tipos, miembros y referencias en el código. Los metadatos se almacenan con el código. Cada archivo que el CLR puede cargar contiene metadatos. El runtime los utiliza para localizar y cargar las clases, mantener las instancias en memoria, resolver el llamado de métodos, generar código nativo, mejorar la seguridad y definir las fronteras del contexto de ejecución.

**CLR** administra la memoria utilizada por las aplicaciones, evitando perdidas de memoria, que podrían estar originadas por errores en el código escrito. El entorno de ejecución brinda además una forma de ejecución que permitirá y administrará la conversión de tipos con los que operan las aplicaciones, la inicialización de las variables, el control de overflows, etc.



A continuación, se realiza una breve reseña de cada uno de los componentes del CLR.

#### **Class loader:**

Administra metadatos (información provista con los archivos, analizada más adelante), carga y disponibilidad de las clases.

#### **Microsoft intermediate language (MSIL) to native compiler:**

Convierte MSIL a código nativo (JIT).

#### **Code manager:**

Administra la ejecución de código.

**Garbage collector (GC):**

Provee la administración automática del ciclo de vida de todos los objetos.

**Security engine:**

Provee la seguridad al código que se ejecuta.

**Debug engine:**

Permite realizar el debug de la aplicación a partir de un seguimiento del código que está siendo ejecutado.

**Type checker:**

Evita que se realicen cambios de tipos inseguros o se utilicen variables no inicializadas.

**Exception manager:**

Provee una estructura de manejo de excepciones, la cual se integra con Windows Structured Exception Handling.

**Thread support:**

Provee clases e interfaces para trabajar con programación multihilos.

**COM marshaler:**

Provee interoperabilidad entre .NET y COM.

**Base Class Library (BCL) support:**

Conjunto de clases que provee el Framework.

## **Sistema Común de Tipos**

**CTS (Common Type System).** El sistema común de tipos define la forma en la que los tipos deben ser declarados, utilizados y administrados por el

runtime. Además, provee una forma para que el runtime de soporte a la integración de varios lenguajes.

El sistema de tipos comunes realiza las siguientes funciones:

Establecer un framework para soporte de integración de múltiples lenguajes, seguridad de tipos y alta performance en la ejecución de código.

Provee un modelo orientado a objetos que soporta la implementación de varios lenguajes de programación.

Define las reglas que debe seguir un lenguaje, lo que asegura que distintos lenguajes puedan interactuar sin problemas.

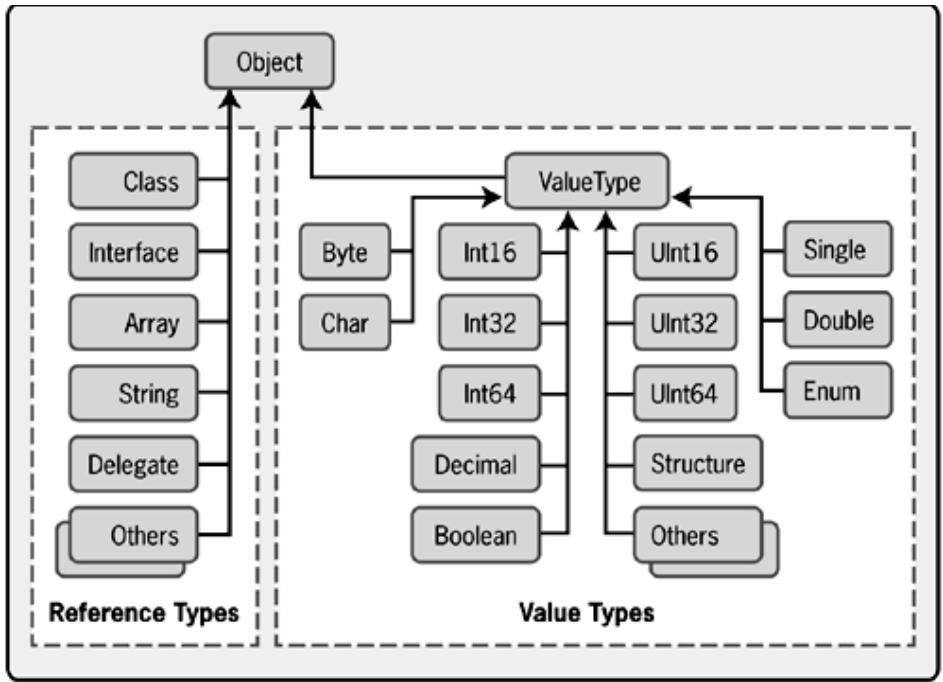
### ***Clasificación de tipos:***

**CTS** soporta dos categorías generales de tipos, cada una de las cuales se divide en subcategorías:

**Value types:** Directamente contienen sus datos. En memoria funcionan de manera similar a las variables tradicionales.

**Reference types:** Almacenan una referencia a una dirección de memoria con un valor, y son alojados en la “heap” o “pila administrada”. Los **Reference types** pueden contener tipos auto-descriptivos, punteros, o interfaces.

Todos los tipos derivan de **System.Object**, que es el tipo base.



## Microsoft Intermediate Language (MSIL)

Cuando se compila código soportado en .NET Framework, el compilador convierte el código fuente en Lenguaje intermedio de Microsoft (**MSIL**), que es un conjunto de instrucciones independiente de la CPU que se pueden convertir de forma eficaz en código nativo.

**MSIL** incluye instrucciones para cargar, almacenar, inicializar y llamar a métodos en los objetos, así como instrucciones para operaciones lógicas y aritméticas, flujo de control, acceso directo a la memoria, control de excepciones y otras operaciones.

Antes de poder ejecutar código, se debe convertir **MSIL** al código específico de la CPU, normalmente mediante un compilador **JIT (Just in time compiler)**.

**Common Language Runtime** proporciona uno o varios compiladores **JIT** para cada arquitectura de equipo compatible, por lo que se puede compilar y ejecutar el mismo conjunto de **MSIL** en cualquier arquitectura compatible.

Cuando el compilador produce **MSIL**, también genera información adicional sobre el código. Esta información describe los tipos que aparecen en el código, incluidas las definiciones de los tipos, las firmas de los miembros de tipos, los miembros a los que se hace referencia en el código y otros datos que el motor de tiempo de ejecución utiliza en tiempo de ejecución.

El lenguaje intermedio de Microsoft (**MSIL**) y los datos adicionales, conocidos como **metadatos**, se incluyen en un archivo ejecutable portable (**PE**), que se basa y extiende el **PE** de Microsoft publicado y el formato **Common Object File Format (COFF)** utilizado tradicionalmente para contenido ejecutable. Este formato de archivo, que contiene código MSIL o código nativo así como metadatos, permite al sistema operativo reconocer lo que debe hacer el **Common Language Runtime**.

La presencia de **metadatos** junto con el **Lenguaje intermedio de Microsoft (MSIL)** permite crear **códigos autodescriptivos**.

En consecuencia la utilización de un **framework** en el desarrollo de una aplicación implica un cierto costo inicial de aprendizaje, aunque a largo plazo es probable que facilite tanto el desarrollo como el mantenimiento.

Existen multitud de **frameworks** orientados a diferentes lenguajes, funcionalidades, etc. Aunque la elección de uno de ellos puede ser una tarea complicada, lo más probable es que a largo plazo sólo los mejor definidos (o más utilizados, que no siempre coinciden con los primeros) permanezcan. Y si ninguno de ellos se adapta a las necesidades de desarrollo, siempre es mejor definir uno propio que desarrollar "al por mayor".

El tema de los **frameworks** es algo que se encuentra en desarrollo. En consecuencia, aún quedan muchos problemas por resolver, tales como la documentación del **framework**. No hay un estándar oficial o de facto para la documentación de los **frameworks**. La falta de un terreno común crea un espacio vacío entre los desarrolladores de **frameworks**, los extendedores de **frameworks** y los usuarios.

Finalmente, los recién llegados al escenario de desarrollo basado en **frameworks**, deberían chequear los puntos expuestos aquí con sumo cuidado. Deberán considerar qué necesita, qué es lo que está haciendo y estar consciente que los **frameworks** tienen sus pros y sus contras, debido a que no son la solución para todos los problemas. Además, si Ud. está considerando usar un **framework** ya existente, estudie su documentación y verifique si hay una buena explicación sobre como utilizar sus facilidades.

### 3 Administración de Memoria

En .Net Framework el encargado de liberar memoria es el **Garbage Colector (GC)**.

En general libera memoria colocando como memoria disponible aquella que posee objetos que ya no se utilizan, pues no tienen referencias que los apunten.

Funciona automáticamente, aunque se lo puede invocar manualmente. Esto último no es recomendable debido a la gran cantidad de recursos que insume su ejecución.

Los objetos pertenecen a una generación. Al funcionar el recolector de basura los objetos que no son borrados sobreviven, esto causa que se eleven a la generación siguiente, esta puede oscilar entre 0 y 2. De esta forma se puede saber cuales objetos han sobrevivido más, asumiendo que esto se debe a que son más utilizados.

Para poder comprender como funciona y cuál es la generación de un objeto usaremos el ejemplo **Ej0001** de esta unidad. Se recomienda que abra el programa y lo ejecute paso a paso a fin de observar el funcionamiento del mismo.

En el código se observa a G, que es el objeto sobre el cual trabajamos y es del tipo string. En primera instancia el mismo será de generación 0.

#### Ejecutar el Ejemplo Ej0001

Es oportuno recordar que, en las relaciones de agregación con contención física, el objeto que agrega suele crear y agregar al resto en lo que denominamos **constructor**, tema ya abordado. Por otro lado, al finalizar el ciclo de vida del objeto que agrega a su fin, existe un último procedimiento que se

ejecuta, denominado **destructor o finalizador**. En este contexto este concepto se divide en dos procedimientos, uno denominado **FINALIZE** y otro llamado **DISPOSE**.

El **FINALIZE** es invocado por el **garbage collector** cuando se ejecuta, mientras que el **DISPOSE** lo invoca el programador.

El detalle que debemos analizar está relacionado con el momento en que se libera un recurso. Por ejemplo, si tenemos un objeto que apunta a un archivo y considerando que el **garbage collector** se ejecutará solo si se necesita liberar memoria, ese archivo referenciado quedará apuntado hasta que esto ocurra. Mientras el objeto siga existiendo en memoria ese archivo no estará disponible para algunas acciones, como borrarlo. Considerando que pueden pasar horas o días para que esto ocurra, sería recomendable colocar la liberación del archivo en el **DISPOSE** y no en el **FINALIZE** ya que el **DISPOSE** puede ser ejecutado por el programador cuando lo desee. Observemos como se pude combinar el uso del **FINALIZE** y **DISPOSE** en el ejemplo **Ej0002**.

### **Ejecutar el Ejemplo Ej0002**

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

## Frameworks

### PROGRAMACIÓN ORIENTADA A OBJETOS



## Frameworks

Titular: Dario Guillermo Cardacci



### PROGRAMACIÓN ORIENTADA A OBJETOS

Titular: Dario G. Cardacci

## ¿Qué es un Framework?

- Es un marco de trabajo que ofrece a quien lo utiliza, una serie de herramientas para facilitarle la realización de tareas.
- Puede contener librerías de clases, documentación, ayuda, ejemplos, tutoriales pero sobre todo contiene experiencia sobre un dominio de problema específico.

## ¿Cuál es el objetivo por el que se genera un Framework?

- Normalizar
- Estandarizar
- Reutilizar la experiencia

## Elementos básicos de un Framework

- Puntos congelados o frozen-spots .
- Puntos calientes o hot spots.
- La característica de inversión de control.

## Etapas constructivas de un Framework

- Análisis del dominio.
- Diseño del framework.
- Instanciación del framework.

## Clasificación de un Framework según su extensibilidad

- De caja Blanca.
- De caja Negra.

## ¿Qué ventajas tiene utilizar un ‘Framework’?

- Permite abstraerse de problemas resuelto con amplia experiencia previa.
- En el caso de .NET el programador no piensa en arquitectura sino en programas.
- Permite aprovechar todas las ventajas de la estandarización.

## Aspectos a considerar

- El costo del análisis del dominio.
- La experiencia.

## La Pregunta

¿Y si no necesito o no quiero utilizar un 'framework'?

## Motivos para no utilizar un framework

- Deseo de crear toda una aplicación sin seguir ningún framework conocido.
- Aplicación pequeña que no amerite aplicar un framework.
- Desconocimiento sobre algún framework que se adapte.
- Falta de tiempo para ejecutar la curva de aprendizaje.

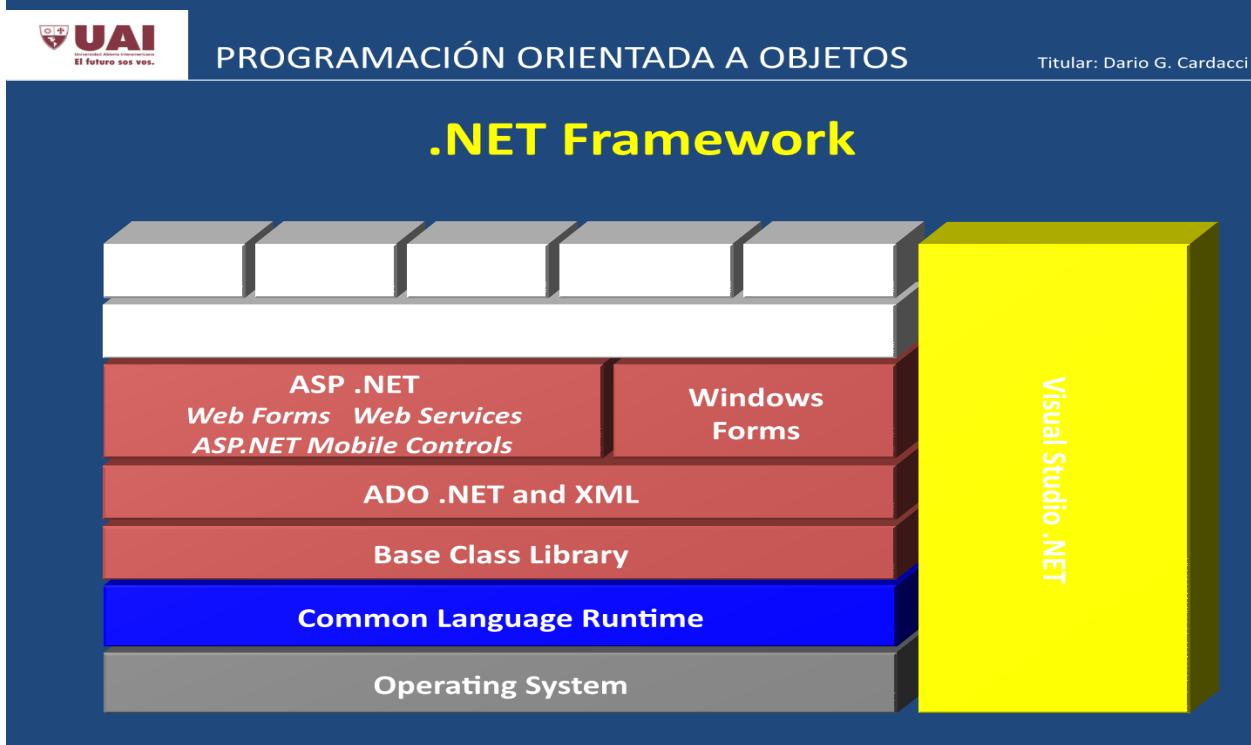
## Motivos para utilizar un framework

- Las aplicaciones crecen y necesitan mantenimiento. Los frameworks facilitan la separación de la presentación y la lógica además de mantener una sintaxis coherente.
- En general hacer un trabajo propio sobre algo que ya existe genera desconocimiento colectivo en el equipo de trabajo.
- El costo de aprender un framework generalmente se compensa con el costo de desarrollar las modificaciones a medida por no utilizarlo.



**FIN**

## DOT net Framework



## ¿Qué problemas resuelve .NET?

- Desde Internet, muchas aplicaciones y dispositivos están fuertemente comunicados entre sí.
- Los programadores escribían arquitectura en lugar de aplicaciones.
- Los programadores tenían conocimientos limitados o debían aprender nuevos lenguajes.

## .NET Framework

- El .NET Framework constituye las bases sobre las que, tanto aplicaciones como servicios, son ejecutadas y construidas.
- La naturaleza unificada del .NET Framework permite que cualquier tipo de aplicación sea desarrollada mediante herramientas comunes haciendo la integración mucho más simple.
- El .NET Framework está compuesto de:
  - El CLR (Common Language Runtime)
  - La BCL (Base Class Library)

## CLR – Common Language Runtime

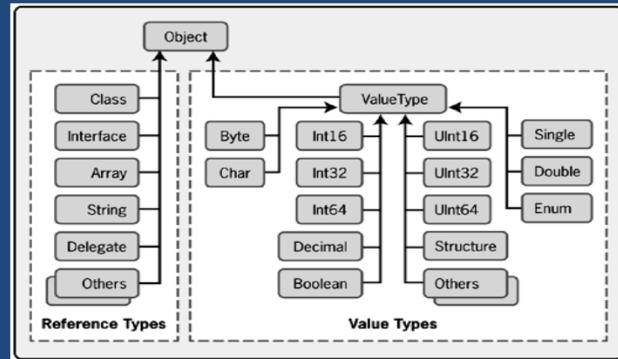
- El CLR es el motor de ejecución (runtime) del .NET Framework.
- Ofrece servicios automáticos tales como:
  - Administración de la memoria
  - Seguridad del código, asegurando:
  - Conversión de tipos
  - Inicialización de variables
  - Indexación de arreglos fuera de sus límites
  - Versionamiento

## Componentes de CLR



## CTS (Common Type System)

- Define un conjunto común de “tipos” orientado a objetos.
- Todo lenguaje de programación debe implementar los tipos definidos por el CTS.
- Todo tipo hereda directa o indirectamente del tipo OBJECT
- Tipos de VALOR y de REFERENCIA



## Microsoft Intermediate Language (MSIL)

- El compilador convierte al código soportado por .NET en lenguaje intermedio.
- Es un conjunto de instrucciones independientes de la CPU, que se convierten en código nativo al ejecutarse.
- Para convertir MSIL a código nativo, se utilizan compiladores llamados “Just In Time” JIT
- Los archivos ejecutables están conformados por
  - MSIL
  - Datos Adicionales (Metadata)
- El MSIL es independiente del lenguaje en el que se desarrolla

## Conclusiones

- Costo inicial de aprendizaje para el uso.
- Facilita el desarrollo.
- Facilita el mantenimiento.
- Arduo trabajo de elección.
- Adecuados para productos cuyos requisitos cambian rápidamente.
- Documentación



FIN

## Administración de Memoria

### PROGRAMACIÓN ORIENTADA A OBJETOS



## Administración de la Memoria

Titular: Dario Guillermo Cardacci



### PROGRAMACIÓN ORIENTADA A OBJETOS

Titular: Dario G. Cardacci

## Garbage Collection GC (Garbage Collector)

- Es el encargado de liberar memoria cuando se ha agotado.
- En general libera memoria colocando como memoria disponible a aquella que posee objetos que ya no se utilizan pues no tienen referencias que los apunten.
- Funciona automáticamente aunque se lo puede invocar manualmente.

## Garbage Collection

- REFERENCIAS CIRCULARES:

- Era uno de los problemas de COM.
- En general las referencias circulares generaban consumo innecesario de memoria.
- En .NET no existen los contadores de referencia (AddRef y Release) en lugar de ello el objeto se ubica en un bloque de memoria administrado (managed heap).

## Garbage Collection

- GENERACIONES:

- Indica la “edad” de un objeto.
- Indica el número de recolecciones de objetos no utilizados a la que el objeto ha sobrevivido.
- Este valor oscila entre 0 y 2.

## Garbage Collection

### Ejemplo

```
Public Class Form1
    'Definición de un Objeto de tipo String
    Dim G As String = "Prueba sobre subsistencia de generaciones"

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        ' Primera consulta. G es un objeto de generación 0.
        TextBox1.Text = TextBox1.Text & Leyenda() & GC.GetGeneration(G)
        Call SaltoDeLinea()
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
        ' Segunda consulta. G es un objeto de generación 1. Observar que se ejecutó el GC.
        GC.Collect() : GC.WaitForPendingFinalizers()
        TextBox1.Text = TextBox1.Text & Leyenda() & GC.GetGeneration(G)
        Call SaltoDeLinea()
    End Sub

    Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button3.Click
        ' Tercera consulta. G es un objeto de generación 2. Observar que se ejecutó el GC.
        GC.Collect() : GC.WaitForPendingFinalizers()
        TextBox1.Text = TextBox1.Text & Leyenda() & GC.GetGeneration(G)
        Call SaltoDeLinea()
    End Sub
```

## Garbage Collection

### Ejemplo

```
Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button4.Click
    ' N consulta. G es un objeto de generación 2. Observar que se ejecutó el GC.
    ' Observar que la máxima generación es 2.
    GC.Collect() : GC.WaitForPendingFinalizers()
    TextBox1.Text = TextBox1.Text & Leyenda() & GC.GetGeneration(G)
    Call SaltoDeLinea()
End Sub

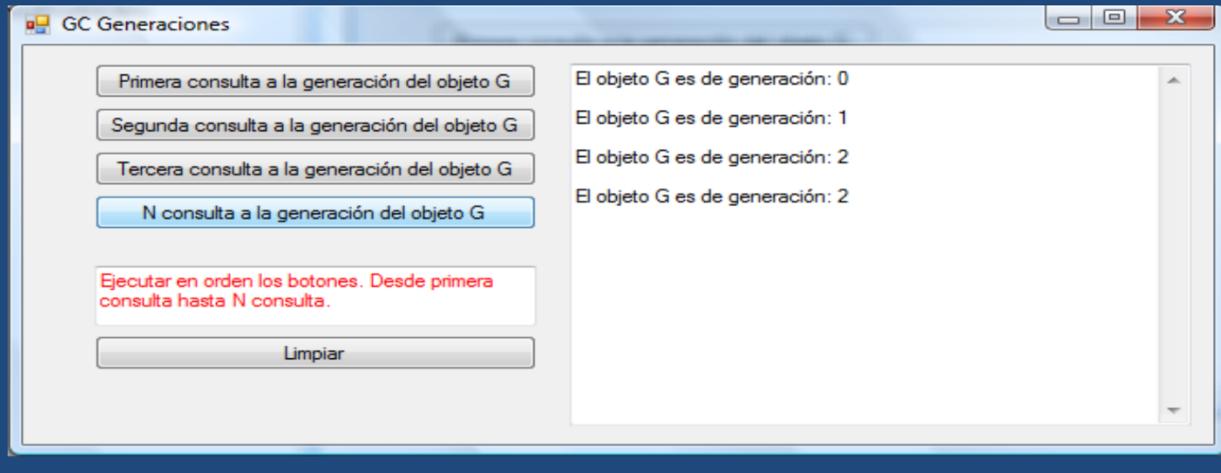
Private Sub SaltoDeLinea()
    TextBox1.Text = TextBox1.Text & vbCrLf & vbCrLf
End Sub

Private Function Leyenda() As String
    Return "El objeto G es de generación: "
End Function

End Class
```

## Garbage Collection

### Ejemplo



## Garbage Collection

### MIEMBROS – MÉTODOS 1

Nombre	Descripción
AddMemoryPressure	Informa al motor en tiempo de ejecución de una asignación grande de memoria no administrada que se debe tener en cuenta al programar la recolección de elementos no utilizados.
Collect	Sobrecargado. Obliga a que se lleve a cabo la recolección de elementos no utilizados.
CollectionCount	Devuelve el número de veces que se ha producido la recolección de elementos no utilizados para la generación de objetos especificada.
GetGeneration	Sobrecargado. Devuelve el número de generación actual de un objeto.
GetTotalMemory	Recupera el número de bytes que se considera que están asignados en la actualidad. Un parámetro indica si este método puede esperar un breve intervalo de tiempo antes de regresar, para permitir que el sistema recoja los elementos no utilizados y finalice los objetos.
KeepAlive	Hace referencia al objeto especificado, convirtiéndolo en un objeto no válido para la recolección de elementos no utilizados desde el principio de la rutina actual hasta el momento en que se llamó a este método.

## Garbage Collection

### MIEMBROS – MÉTODOS 2

Nombre	Descripción
RemoveMemoryPressure	Informa al motor en tiempo de ejecución de que se ha liberado la memoria no administrada y ya no se necesita tener en cuenta al programar la recolección de elementos no utilizados.
ReRegisterForFinalize	Solicita que el sistema llame al finalizador del objeto especificado, para el que previamente se ha llamado a <code>SuppressFinalize</code> .
SuppressFinalize	Solicita que el sistema no llame al finalizador del objeto especificado.
WaitForPendingFinalizers	Suspende el subproceso actual hasta que el subproceso que está procesando la cola de finalizadores vacíe dicha cola.

## Garbage Collection

### MIEMBROS – MÉTODOS 1

Nombre	Descripción
MaxGeneration	Obtiene el número máximo de generaciones que el sistema admite en la actualidad.

## Método Finalize

- Es llamado por el recolector de basura antes de liberar la memoria asignada a un objeto.
- Se hereda este método de System.Object.
- Se deberá declarar como “Protected” y “Overrides”.
- Se dispara cuando todos los punteros al objetos se han perdido. Esto pudo haber ocurrido por asignarles Nothing a las variables que los apuntaban o por que las variables quedaron fuera de scope.

## Método Dispose

- Como los objetos no poseen destructor las clases bien diseñadas deberían exponer el ‘Dispose’.
- Se debe implementar por medio de la interfaz IDisposable.
- Esta interfaz expone un único método denominado “Dispose”.
- Una buena práctica de programación indica que el método “Dispose” de un objeto debe llamar a los métodos “Dispose” de los objetos internos que estén contenidos en él.
- El “Dispose se utiliza cuando usamos objetos que son “NO Administrados” (p.e apertura de un archivo), ya que la real liberación de la memoria se produce cuando actúa el GC

## Finalize y Dispose

- Se puede utilizar el “Finalize” y el “Dispose” de manera conjunta.
- El “Dispose” en general para liberar los recursos no administrados.
- El “Finalize” para los recursos administrados por el GC.



**FIN**

---

## **Actividades asincrónicas**

Guía de preguntas de repaso conceptual

1. ¿Qué es un Framework?
2. ¿Qué son los frozen-spots en un Framework?
3. ¿Qué son los hot-spots en un Framework?
4. ¿Cómo se puede clasificar un Framework según su extensibilidad?
5. ¿Qué es un Framework de Caja Blanca?
6. ¿Qué es un Framework de Caja Negra?
7. ¿Qué ventajas posee utilizar un Framework?
8. ¿Qué problemas resuelve .NET Framework?
9. ¿Qué es y qué permite hacer el CLR?
10. ¿Qué es el MSIL?
11. ¿Qué es el CTS?
12. ¿Qué es el CLS?
13. ¿Dónde se encuentran las instancias de los objetos administrados por el GC?
14. ¿Cuáles son los dos métodos más notorios que deben implementar las clases para trabajar correctamente con la recolección de elementos no utilizados y matar las instancias administradas y no administradas?
15. ¿De dónde heredan las clases el método Finalize?
16. ¿Cuál es la firma que implementa el método “Finalize”?
17. ¿Qué método se utiliza para que el GC recolecte los elementos no utilizados?

18. ¿Qué método se utiliza para suspender el subprocesso actual hasta que el subprocesso que se está procesando en la cola de finalizadores vacíe dicha cola?
19. ¿Cuándo se ejecuta el método collect del GC que método se ejecuta en los objetos afectados?
20. ¿Qué método debería exponer una clase bien diseñada teniendo en consideración que no posee destructor?
21. ¿Cómo obtengo el método “Dispose”?
22. ¿Qué se programa en el método “Dispose”?
23. ¿Se pueden combinar el uso de “Dispose” y “Finalize”?
24. ¿A qué se denomina “Resurrección de Objetos”?
25. ¿A qué se denomina “Generación” en el contexto de la recolección de elementos no utilizados?
26. ¿Qué valores puede adoptar la “Generación” de un objeto?
27. ¿Cómo se puede obtener el número de veces que se ha producido la recolección de elementos no utilizados para la generación de objetos especificada?
28. ¿Cómo se obtiene el número de generación actual de un objeto?
29. ¿Cómo se puede recuperar el número de bytes que se considera que están asignados en la actualidad?
30. ¿Qué utiliza para convertir un objeto en “no” válido para la recolección de elementos no utilizados desde el principio de la rutina actual hasta el momento en que se llamó a este método?
31. ¿Cómo se solicita que el sistema no llame al finalizador del objeto especificado?
32. ¿Cómo se solicita que el sistema llame al finalizador del objeto especificado, para el que previamente se ha llamado a “SuppressFinalize”?

33. ¿Cómo se obtiene el número máximo de generaciones que el sistema admite en la actualidad?

### Guía de ejercicios

1. Desarrollar un programa que genera varias instancias (una cantidad importante), verifique la memoria utilizada, pase el GC y vuelva a verificar el espacio de memoria. ¿Qué se observa?
2. Desarrollar un programa que genere una instancia, pierda la referencia a la misma y aplicando la técnica de “resurrección de objetos” logre obtener la referencia a ese mismo objeto.