Unidad 1

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 1

Introducción

Introducción

En Este Capítulo

- ¿Qué es .Net?
- ¿Cómo se maneja un programa en memoria?
- ¿Dónde se almacenan y qué son las variables?
- Comentarios
- Identificadores
- Tipos primitivos
- Tipos referenciados
- Dónde se almacenan y qué son los métodos
- Clases
- Uso de Bloques, Espacios en Blanco y Finalización de Sentencia en C#
- Constructores
- Objetos y Mensajes
- Strings
- Ocultamiento de la información
- Encapsulado
- ¿Dónde empiezan los programas y por qué?
- Espacios de nombre
- Construcción e Inicialización de Objetos
- Palabras clave

¿Qué es .Net?

.Net no es tan solo un lenguaje, es una tecnología para el desarrollo de aplicaciones que conforma la plataforma sobre la cual se ejecutarán las mismas y en conjunto esta se compone de:

- Un conjunto de lenguajes de programación
- Un entorno de desarrollo
- Un entorno para aplicaciones
- Un entorno para despliegue de aplicaciones

La sintaxis es similar en todos los lenguajes porque se basan en el Framework de .Net, pero el manejo y la semántica son específicos de cada uno. La diferencia fundamental entre cada uno de los lenguajes son las capacidades de cada uno. Si bien son mínimas existen y eso hace que uno sea mejor o peor que el otro. Esta en cada desarrollador elegir el lenguaje que le sea más cómodo según sus necesidades.

Bytecodes

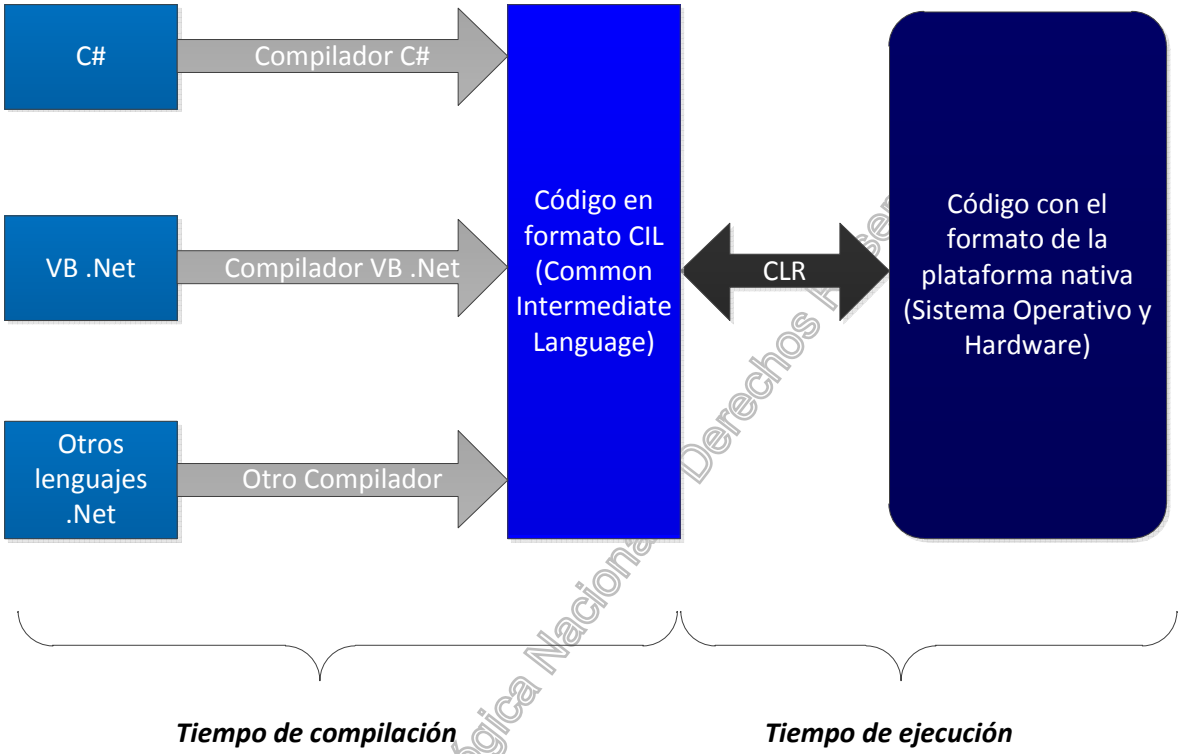
En los lenguajes de programación de .Net, se puede "escribir una vez, ejecutar en cualquier plataforma que contenga un CLR (Common Language Runtime)". Sin embargo, cabe destacar que existen proyectos para que la plataforma se ejecute también en otras plataformas. Esto significa que cuando se compila un programa, no se generan instrucciones para una plataforma específica. En su lugar, se generan bytecodes, que son instrucciones para el CLR. El bytecode no es el código máquina de ninguna computadora en particular, y puede por tanto ser portable entre diferentes arquitecturas.

Common Language Runtime

El Common Language Runtime o CLR ("lenguaje común en tiempo de ejecución") es un entorno de ejecución para el código de programas que corren sobre la plataforma Microsoft .NET. El CLR es el encargado de compilar una forma de código intermedio llamada Common Intermediate Language (CIL, anteriormente conocido como MSIL, por Microsoft Intermediate Language), al código de maquina nativo, mediante un compilador en tiempo de ejecución. **No debe confundirse el CLR con una máquina virtual**, ya que una vez que el código esta compilado, corre nativamente sin intervención de una capa de abstracción sobre el hardware subyacente (en el que se encuentra en ese momento). Es una implementación del estándar Common Language Infrastructure (CLI).

Los desarrolladores que usan CLR escriben el código fuente en un lenguaje compatible con .NET, como C# o Visual Basic .NET. En tiempo de compilación, un compilador .NET convierte el código a IL, lo cuál es conocido como compilación estática porque no se realiza en tiempo de ejecución. Al ejecutarse los programas, el compilador en tiempo de ejecución del CLR convierte el código CIL en código nativo para el sistema operativo subyacente. Alternativamente, el código CIL es compilado a código nativo en un proceso separado previo a la ejecución. Esto acelera las posteriores ejecuciones del software debido a que la compilación de MSIL a nativo ya no es necesaria. A esto se lo denomina compilación por demanda, compilación en tiempo de ejecución (también conocida

por sus siglas inglesas, JIT, *just-in-time*) o traducción dinámica, la cual es una técnica para mejorar el rendimiento de sistemas de programación que compilan a bytecode, que consiste en traducirlo a código de máquina nativo en tiempo de ejecución. La compilación en tiempo de ejecución se construye a partir de las dos ideas anteriores relacionadas con los entornos de ejecución: la compilación a bytecode y la compilación dinámica.



A pesar de que algunas implementaciones del Common Language Infrastructure se ejecutan en sistemas operativos que no sean Windows, el CLR se ejecuta solo en Microsoft Windows.

Se puede resumir el manejo que realiza el Framework de .Net del código escrito en un programa de la siguiente manera:

1. Se escribe un programa. Net en C#, VB.Net, F#, o algún lenguaje compatible.
2. Ese código se compila a un lenguaje intermedio (IL), que es similar al código de bytes (bytecodes) de Java.
3. La IL es empaquetado en ensamblados (por ejemplo, archivos *.dll o *.exe) para la implementación del mismo (despliegue de aplicación).
4. La IL en los ensamblados se distribuye en las computadoras de los usuarios finales.
5. El usuario final invoca el programa por primera vez en un equipo con la versión correcta del Framework de .Net instalado.

6. El equipo considera que se trata de un ensamblado de .Net en lugar de código "en bruto" de la máquina (binario), y se lo pasa al compilador JIT.
7. El compilador JIT compila el código IL al de la máquina para que éste sea totalmente nativo.
8. El código nativo se guarda en la memoria para el ciclo de vida de ejecución de ese programa.
9. A partir de ese punto, el código guardado nativo es invocado, y la IL ya no se utiliza.

Hay un par de puntos importantes en lo enunciado, pero lo importante es que en ningún momento ningún código es interpretado nunca. En su lugar, se puede ver en el paso 7 que se compila a código nativo. Esta es una gran diferencia de cargarlo en una máquina virtual, por varias razones:

a. El código es ejecutado por la CPU directamente en lugar de ser interpretado por una capa de abstracción, lo que debería ser más rápido.
b. El compilador JIT puede tomar ventaja de optimizaciones específicas para la máquina, en lugar de conformarse con un mínimo común denominador.
c. Si se desea, incluso se puede pre-compilar el código y en esencia ocultar el paso 7 del usuario por completo.

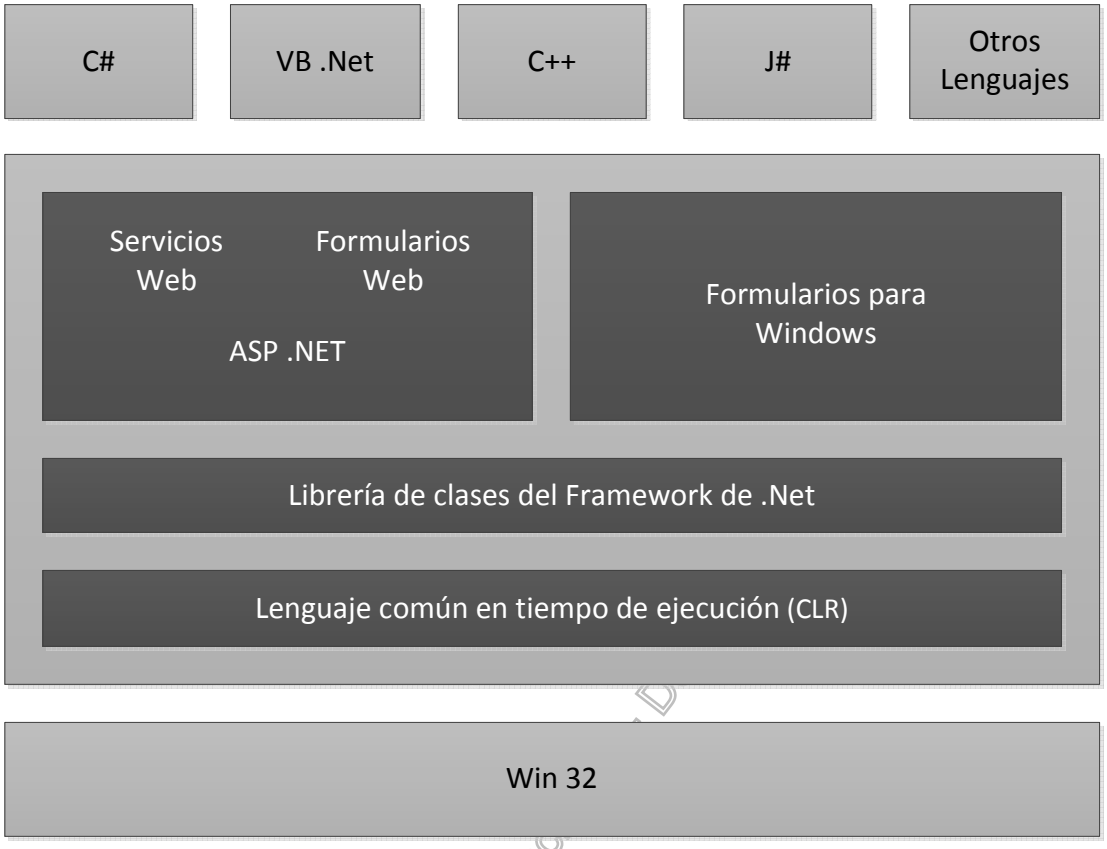
La manera en que la maquina se relaciona con el CLR permite a los programadores ignorar muchos detalles específicos del hardware que estará ejecutando el programa. El CLR también permite otros servicios importantes, incluyendo los siguientes:

- Administración de la memoria
- Administración de hilos
- Manejo de excepciones
- Recolección de basura
- Seguridad

Arquitectura del Framework de .NET

El Framework de .NET tiene dos componentes:

- La biblioteca de clases del Framework de .NET: provee tipos que son comunes a todos los lenguajes (CTS - Common Type System).
- El Common Language Runtime se compone de un cargador de clases que carga el código IL de un programa en el tiempo de ejecución, el cual compila el código IL a código nativo, ejecuta y gestionar el código para garantizar la seguridad y que los tipos estén asegurados (esto es, si se tiene un objeto y se declaran otros iguales no haya ambigüedad de cual es cual) y proporcionar soporte para threads.



La arquitectura tiene en su parte superior idiomas como VB .NET, C#, J #, C++, etc. Los desarrolladores pueden crear (utilizando cualquiera de los lenguajes anteriores), aplicaciones con tecnología para formularios Windows, formularios Web, servicios de Windows y servicios Web en XML. Las dos capas inferiores consisten en el Framework de .NET y el lenguaje común en tiempo de ejecución (CLR - Common Language Runtime).

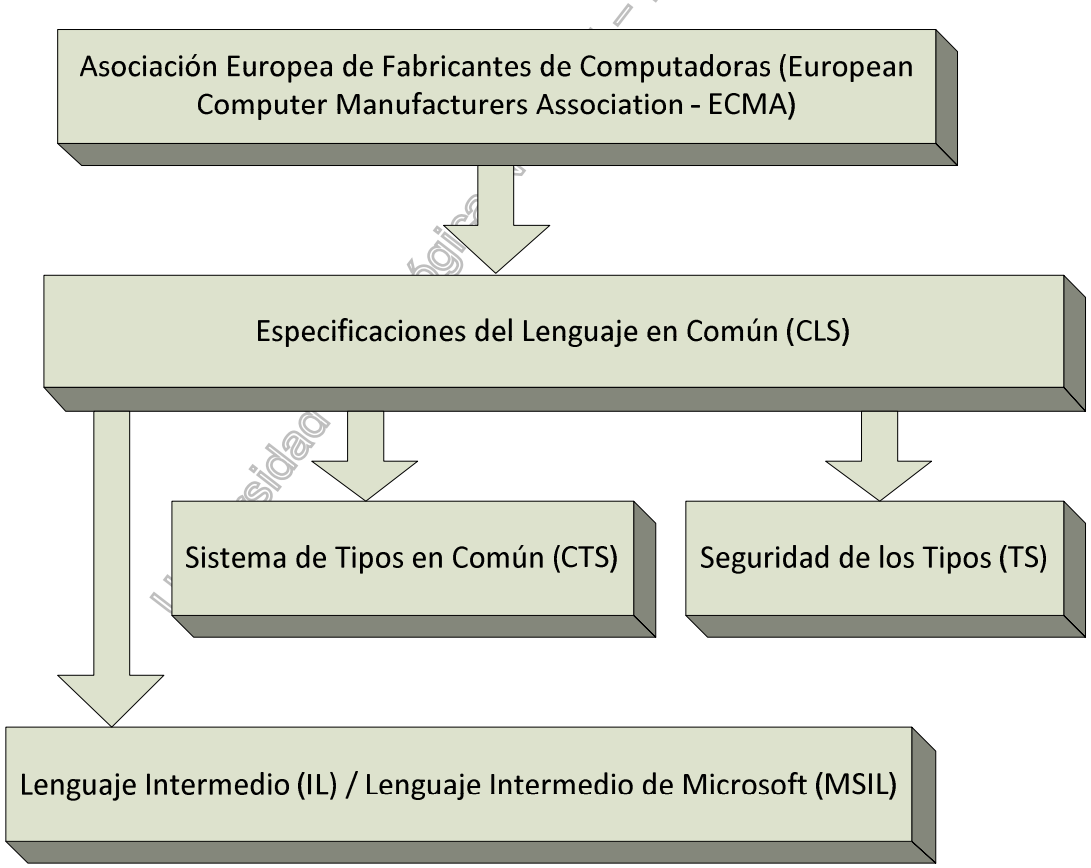
El Common Language Runtime es la base del Framework de .NET. CLR actúa como un agente que administra el código en tiempo de ejecución y brinda la prestación de servicios básicos tales como la gestión de memoria, el manejo de threads, y el acceso remoto, así como también fuerza la seguridad de estricta tipos. El concepto de gestión de código es un principio fundamental del CLR. ***El código que se dirige al CLR se conoce como código administrado o manejado, mientras que el código que no se dirige el CLR se conoce como código no administrado o no manejado***.

La biblioteca de clases, es un componente entero del Framework de .NET y se compone de una colección de clases reutilizables orientadas a objetos (los tipos asegurados) que se pueden utilizar para desarrollar programas que van desde las tradicionales aplicaciones de línea de comandos hasta cualquier interfaz gráfica de usuario (GUI) como los formularios Windows, ASP. NET, formularios Web, servicios de Windows y servicios Web XML.

El estándar de la European Computer Manufacturers Association (ECMA) ha definido la Common Language Specification (CLS). Esta especificación obliga a los lenguajes de desarrollo de software que se ajusten a ella seas compatibles entre si. El código escrito bajo la CLS debe ser compatible con el código escrito en otro lenguaje que se ajuste a la CLS. La razón es que el código que realizado en un lenguaje compatible con CLS debe ser compilado generando un lenguaje intermedio (IL). El motor del CLR ejecuta el código IL. Esto garantiza la interoperabilidad entre los lenguajes compatibles con CLS. El Framework de Microsoft .NET admite Idiomas como Microsoft Visual Basic .NET, Microsoft Visual C#, Microsoft Visual C++ .NET y Microsoft Visual J#. NET, para citar algunos.

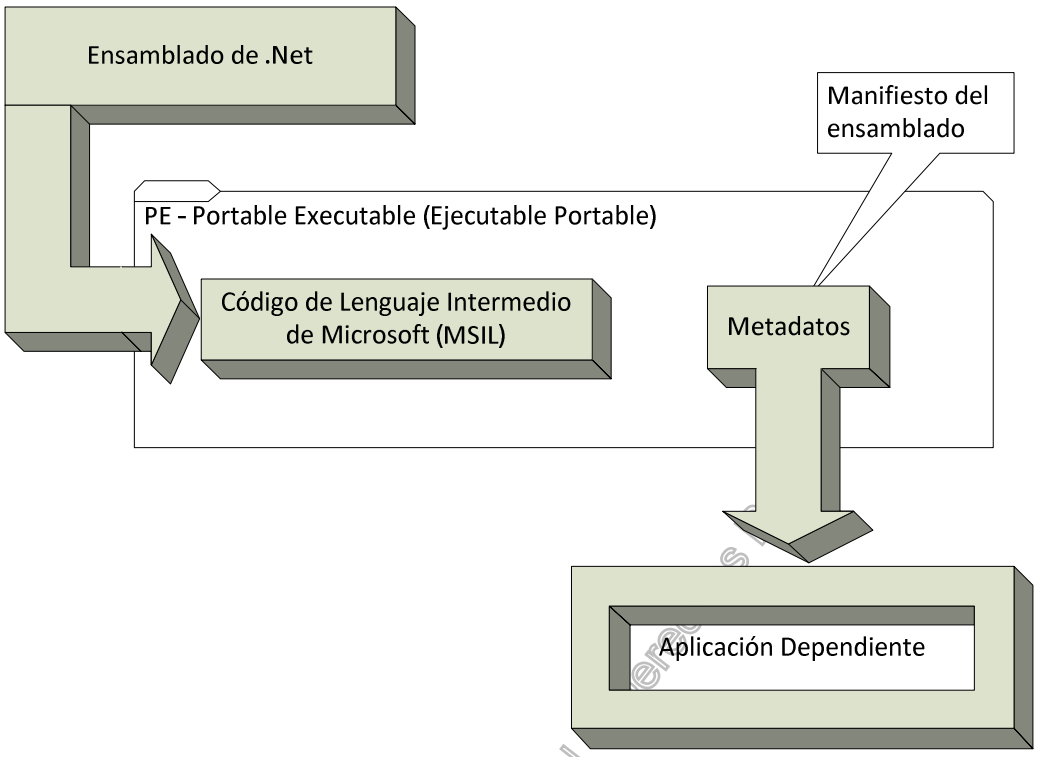
Los compiladores década lenguaje generan un código de lenguaje intermedio, llamado Microsoft Intermediate Language (MSIL), lo que hace que los programas escritos en cualquier lenguaje de .NET sean interoperables entre si.

El estándar ECMA, Common Language Infrastructure (CLI), define las especificaciones para la infraestructura que necesita el código IL para su ejecución. La CLI proporciona un sistema de tipos en común (CTS) y servicios, tales como la seguridad de tipos, ejecución de código manejado y ejecución lado a lado. La siguiente figura muestra el estándar de la ECMA para Microsoft.



El Framework de .NET proporciona la infraestructura y los servicios las cuales son las especificaciones de la Common Language Infrastructure (CLI). Estas incluyen:

- **Common Language Runtime**
  - El CLR incluye CLI
  - El CLR también proporciona el entorno de ejecución de las aplicaciones .NET.
- **Sistema de tipo en común**
  - Proporciona los tipos de datos, valores, tipos de objetos. Esto ayuda a los desarrolladores crear aplicaciones en diferentes lenguajes. En el lugar en el cual los lenguajes de .NET comparten el CTS significa que todos los tipos utilizados en las aplicaciones comparten los mismos tipos definidos en el CLI.
- **Seguridad de los tipos**
  - . NET Framework realiza operaciones sobre valores u objetos para los cuales requiere que cada valor u objeto tenga su tipo y quien hace referencia a ese tipo por valor u objeto.
- **La ejecución de código administrado**
  - El Framework de .NET administra el estado del objeto durante la ejecución de las aplicaciones. NET.
  - El Framework de .NET asigna automáticamente la memoria y proporciona un mecanismo de recolección de la basura para desasignar memoria.
- **Ejecución lado a lado**
  - El Framework de .NET permite diferentes versiones de la misma aplicación ejecutarse en la misma máquina mediante el uso de los ensamblados de las diferentes versiones. Los ensamblados consisten de código IL y metadatos, en donde los metadatos determinan las dependencias de la aplicación. Por este medio, en tiempo de ejecución el Framework de .NET ejecuta varias versiones de ensamblados para resolver de esta manera el problema crucial del entorno de desarrollo heredado. Esto es conocido como “el infierno de las DLL”. El siguiente gráfico muestra un diagrama de bloques de la ejecución lado a lado.



El CLR del Framework de .NET

El Common Language Runtime facilita lo siguiente:

- Un entorno para tiempo de ejecución
  - El CLR carga la aplicación dentro del entorno de ejecución, compila el código IL a código nativo y ejecuta el código.
- Servicios en tiempo de ejecución
  - Gestión de la memoria
  - Seguridad de los tipos
  - Fuerza la seguridad
  - Administración de excepciones.
  - Soporte de threads
  - Soporte de depuración

Arquitectura del CLR

Para ejecutar un programa y obtener todos los beneficios del entorno de ejecución administrado hay que escribir código en un lenguaje que sea compatible con la CLS (especificación común de lenguajes) es decir el Framework de .NET. El compilador de lenguaje compila el código fuente convirtiéndolo en código MSIL el cual es independiente de cualquier CPU, por lo tanto sus instrucciones son independientes de la plataforma. MSIL consiste en lo siguiente:

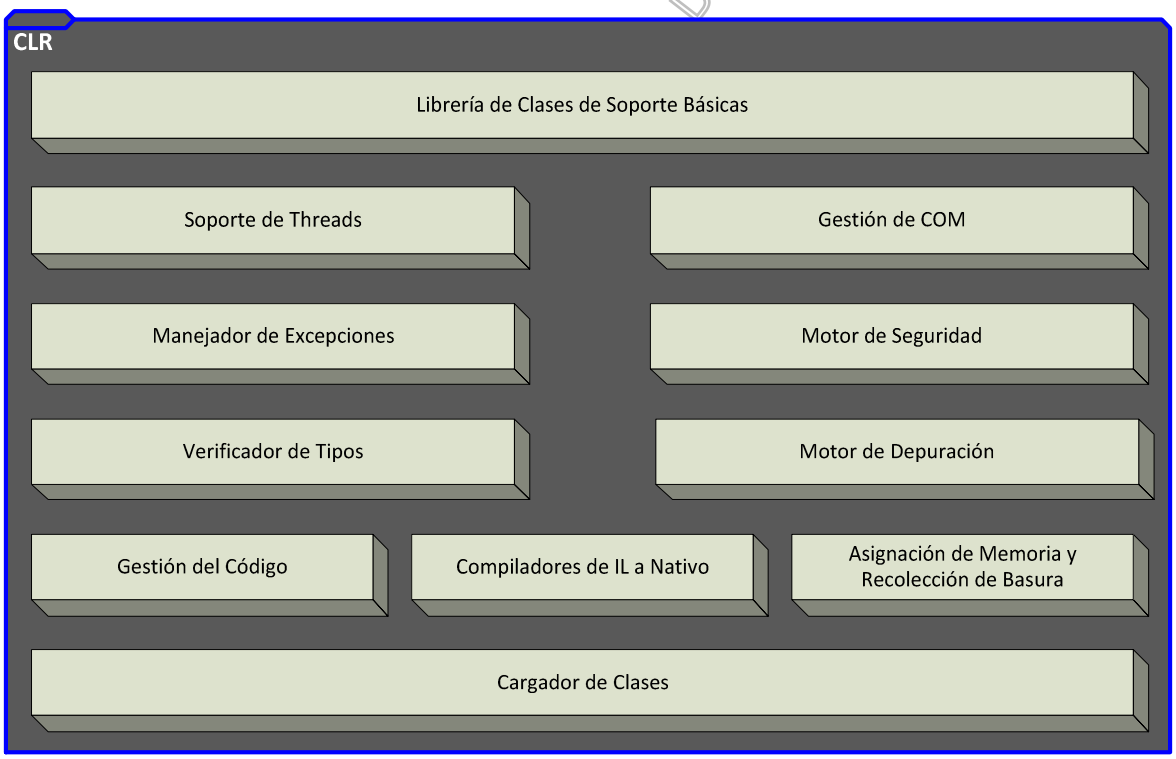
- Instrucciones que permiten realizar operaciones aritméticas y lógicas.

- Acceso directo a memoria.
- Control del flujo de ejecución.
- Manejo de excepciones.

El código MSIL se puede compilar con instrucciones específicas para una CPU antes de la ejecución, para lo cual el CLR requiere información sobre el código que no es otra cosa que los metadatos. Dichos metadatos describen el código y define los tipos que este contiene, así como también referencia a otros tipos que el código utiliza en tiempo de ejecución.

Un ensamblado consiste en un archivo ejecutable portable (PE). En el momento de la ejecución de archivo PE el cargador de clases carga el código MSIL y los metadatos del archivo portable ejecutable en la memoria en tiempo de ejecución.

Antes de la ejecución del archivo PE, pasa el código al compilador de código nativo para la compilación, de manera de realizar la compilación de IL a código nativo, lo cual lo realiza un compilador JIT (Just In Time Compiler). Para una arquitectura de CPU diferente cambian los compiladores para el código IL de manera de compilar adecuadamente a instrucciones nativas de esa plataforma en particular.



Donde:

- **Cargador de Clases**: tiene como función la carga las clases dentro del CLR.
- **Compiladores de IL Nativo**: traducción del código MSIL en código nativo.

- **Gestión del Código**: maneja el código durante la ejecución.
- **Asignación de Memoria y Recolección de Basura**: es el encargado de llevar a cabo la gestión automática de memoria.
- **Motor de Seguridad**: hace cumplir las restricciones de seguridad a nivel de código, como el nivel de seguridad en una carpeta y la seguridad a nivel de máquina usando las herramientas proporcionadas por Microsoft .NET y utilizando el ajuste para el Framework de .NET en el panel de control.
- **Verificador de Tipo**: impone una estricta comprobación de tipos.
- **Soporte de Threads**: proporciona soporte para las aplicaciones de subprocesos múltiples.
- **Manejador de Excepciones**: proporciona un mecanismo para gestionar el manejo de excepciones en tiempo de ejecución.
- **Motor de Depuración**: permite a los desarrolladores depurar los diferentes tipos de aplicaciones.
- **Gestión de COM**: permite a las aplicaciones. NET intercambiar datos con aplicaciones COM.
- **Librería de Clases de Soporte Básicas**: proporciona las clases (tipos) que las aplicaciones necesitan en tiempo de ejecución.

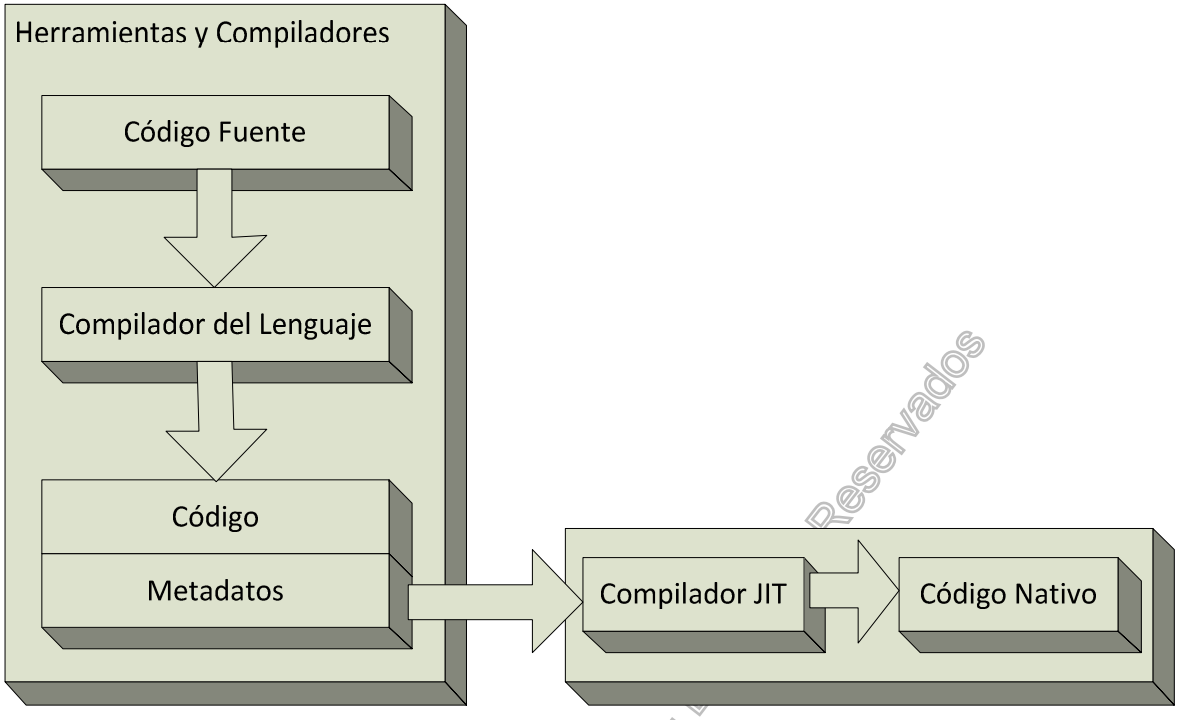
Compiladores en tiempo de ejecución (JIT – Just In Time)

Un compilador de tiempo de ejecución es una parte integral del CLR. Su función es la de compilar IL (Intermediate Language) por demanda, esto es, cuando se trata de ejecutar el código por primera vez, se compila y ejecuta. La siguiente vez que se requiera el código, si está compilado, no se realiza esta función incrementando la velocidad de carga y ejecución.

El cargador de clases del CLR carga el código MSIL y los metadatos en la memoria. El gestor de código llama al método de punto de entrada de programa, el cual es el método WinMain o DLLMain. El compilador JIT compila el método antes de la ejecución del punto de entrada. El gestor de código coloca los objetos en la memoria y controla la ejecución del código. El recolector de basura realiza controles periódicos sobre el heap administrado para identificar los objetos que no está en uso para la aplicación y liberar la memoria.

En el momento de la ejecución del programa el comprobador de tipos se asegura de que todos las las referencias de los objetos y los valores tiene un tipo válido. El verificador de tipos también se asegura de que sólo se realizan operaciones válidas en el código, en otro caso será lanzada una excepción. El código es controlado por CLR en tiempo de ejecución. CLR fuerza la seguridad de la siguiente manera:

- Para controlar y acceder a los recursos del sistema, como disco rígido.
- Para controlar y acceder a las conexiones de red.
- Para controlar y acceder a los recursos de hardware.



Ejecución del código administrado por el CLR

La ejecución del código administrado es el proceso ejecutado por el CLR cuando se corre una aplicación. Sus principales responsabilidades son:

- El CLR carga el MSIL y refiere los metadatos del programa a ejecutar.
- Se compila el código cargado a nativo.
- El CLR ejecuta el código nativo.
- El CLR proporciona gestión automática de memoria.
- La ejecución gestionada también lleva a cabo las compilaciones JIT, cuando sean necesarias, como la carga de una DLL en MSIL.
- Garantiza la seguridad de tipos.
- Fuerza la seguridad.
- Manejo de excepciones.

Proceso de la ejecución administrada

El código administrado es tal que se explica por sí mismo dándole información al CLR de los diferentes servicios en tiempo de ejecución del Framework de .NET que necesita para correr adecuadamente.

Esta información se almacena en código MSIL en forma de metadatos en el archivo PE. Dichos metadatos describen información acerca de los tipos de datos que el código contiene.

Los datos gestionados son asignados por el gestor de memoria y liberados de la memoria de forma automática por el recolector de basura. Los datos administrados pueden ser accesibles por el

código administrado, pero el código administrado puede ser accesible tanto por datos administrados como por no administrados. Esto es necesario por la compatibilidad con el código heredado que no se ejecuta como código administrado pero debe tener la posibilidad de interaccionar con el que si lo es.

Manejo automático de la memoria

Como no es necesario escribir explícitamente código para la gestión de la memoria, tanto cuando se deben alojar tipos como cuando se debe liberar el recurso solicitado, la gestión está automatizada. Esto libera tanto al desarrollador como a la aplicación misma de los errores derivados por el mal manejo de los recursos en memoria. El proceso del alojamiento en memoria implica:

- Alojamiento de memoria
  - Cuando un proceso se inicia, el módulo del tiempo de ejecución (la parte del CLR dedicada a este fin) reserva un espacio de direcciones contiguo sin asignar espacio de almacenamiento para él.
  - Este espacio de direcciones reservadas se denomina montón (heap) administrado. El heap administrado mantiene un puntero en el lugar donde se encontrará el siguiente objeto.
  - Cuando una aplicación utiliza el operador `new` o `New` para crear un objeto, el nuevo operador comprueba si la memoria requerida por el objeto está disponible en el heap.
  - Cuando se cree el siguiente objeto, el recolector de basura asigna memoria para el objeto en el heap administrado.
  - La asignación de memoria para objetos en un heap administrado lleva menos tiempo que la asignación de memoria no administrada.
  - En la memoria no administrada, los punteros a la memoria se mantienen en listas enlazadas de estructuras de datos. Por lo tanto, la asignación de memoria requiere navegar a través de la lista enlazada y encontrar un bloque de memoria grande para acomodar lo que se quiere alojar.
  - Puede acceder a los objetos en la memoria administrada más rápido que los objetos en la memoria no administrada, porque, en la asignación de memoria administrada, los objetos se crean de forma contigua en el espacio de direcciones administrada.
- Liberación de memoria
  - El recolector de basura periódicamente libera la memoria de los objetos que ya no son necesarios por la aplicación.
  - Cada aplicación tiene un conjunto de raíces. Las raíces apuntan a la ubicación de almacenamiento en el heap administrado. Cada raíz o bien se refiere a un objeto

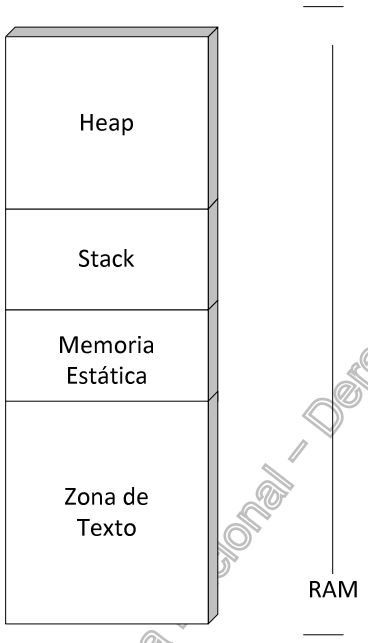
en el montón administrado, o se establece en `null`, `Nothing` o la definición de un valor nulo según el lenguaje

- Las raíces de una aplicación consisten en punteros de objetos globales y estáticos, variables locales y parámetros de referencia de objetos en una pila de threads.
- El compilador JIT y el gestor en tiempo de ejecución mantienen una lista de las raíces de la aplicación.
- El recolector de basura utiliza esta lista para crear un gráfico de objetos en el montón administrado que son accesibles desde la lista raíces.
- Cuando el recolector de basura comienza a ejecutarse, considera que todos los objetos en el heap administrado como basura.
- El recolector de basura navega a través de la lista de raíces de la aplicación e identifica los objetos que tienen referencias activas correspondientes en dicha lista y los marca como alcanzables (activos).
- El recolector de basura también considera a tales objetos como alcanzables.
- El recolector de basura también considera a los objetos inalcanzables en el heap administrado como basura.
- El recolector de basura lleva a cabo un proceso de recolección para liberar la memoria ocupada por los objetos considerados basura.
- El recolector de basura realiza la función de copia de memoria para comprimirla con los objetos en el heap administrado. Esto determina la optimización del espacio de memoria contigua para la asignación de la misma.
- El recolector de basura actualiza los punteros en la lista de raíces de la aplicación para que apunten correctamente a los objetos a los cuales apuntaban anteriormente.
- El recolector de basura utiliza un mecanismo altamente optimizado para llevar a cabo la recolección de basura. Se divide los objetos en el heap administrado en tres generaciones: 0, 1 y 2. Los de generación 0 son los objetos de creación reciente.
- El recolector de basura primero recoge los objetos inalcanzables en la generación 0. A continuación, el recolector de basura compacta de memoria y promueve los objetos accesibles a la generación 1.
- Los objetos que sobreviven el proceso de recolección son promovidos a generaciones más altas.
- Las búsquedas de objetos inalcanzables del recolector en las generaciones 1 y 2 sólo sucede cuando la memoria liberada por el proceso de recolección de objetos de la generación 0 es insuficiente para crear un nuevo objeto.
- El recolector de basura gestiona la memoria de todos los objetos administrados creados por la aplicación.

- La recolección de basura de forma explícita puede liberar estos recursos del sistema, proporcionando el código de limpieza en el método Dispose del objeto.
- Se debe que llamar explícitamente al método Dispose cuando se termine de trabajar con un objeto.

¿Cómo se maneja un programa en memoria?

La estructura de todo programa en memoria es la que muestra la siguiente figura



Donde:

- **Zona de Texto**: es el lugar donde se almacenan todas las instrucciones de programa
- **Memoria Estática**: es donde se almacenan las variables globales (según el lenguaje) y las estáticas.
- **Stack**: es el lugar donde se almacenan los parámetros y variables locales de un método. Se agranda o reduce dinámicamente en cada invocación a un método
- **Heap**: es la memoria asignada a un programa que no se encuentra en uso actualmente pero que se puede pedir dinámicamente.

¿Dónde se almacenan y qué son las variables?

El funcionamiento de la estructura básica que diseña un enlazador para un programa compilado y enlazado es emulado por el CLR.

Las variables son los espacios de almacenamiento en memoria que provee el lenguaje. Todas las variables declaradas en .Net dentro de los métodos, subprocesos o funciones se almacenan en el stack. Esto quiere decir que salvo dos excepciones, el único espacio de almacenamiento será en el stack.

Una excepción, como se verá posteriormente, son las variables que se encuentran dentro de los objetos fuera de los métodos, subprocesos o funciones. Todos los objetos en .Net se crean pidiendo memoria dinámicamente, por lo tanto, su espacio de almacenamiento es en el heap.

La segunda excepción son las variables estáticas, las cuales se almacenan en la memoria estática.

En un lenguaje de programación orientado a objetos, las variables son directamente asociadas a los atributos de las clases, aunque se utilicen en otros lugares que no sean directamente atributos, como los parámetros de un método o las variables locales a éstos. Por lo tanto en .Net hablar de variables o atributos es lo mismo salvo que estén en un método.

Nota: Visual Basic admite la declaración de variables globales en los módulos. Si bien es sintácticamente correcto, su uso no es recomendado y no va a ser tenido en cuenta a lo largo de futuras explicaciones, ya que se conserva esta capacidad en el lenguaje por cuestiones de compatibilidad respecto de versiones anteriores del mismo.

Variables

En .Net se puede hacer una primera clasificación de las variables según su procedencia en dos grupos:

- Tipos primitivos de .Net (Common Type System)
- Tipos Referenciados

La principal diferencia entre estos dos grupos es que el primero son los tipos de datos prexistentes definidos en cada lenguaje mientras que los segundos son aquellos que pueden almacenar las referencias que se obtienen al crear un objeto de cualquier clase que se haya definido.

Propiedades de las variables

Las variables en .Net poseen propiedades que permiten su control y manejo a lo largo de un programa. Las propiedades de una variable son las siguientes:

- Son espacios en memoria que almacenan datos
- Siempre tienen almacenado un valor desde su declaración
- Siempre deben ser declaradas por un tipo primitivo o referenciado
- Poseen visibilidad y alcance (lugares desde donde se pueden acceder y lugares desde los cuales no)
- Se pueden utilizar como parámetros de los métodos
- Se puede retornar el valor que almacenen desde un método

El manejo de estas propiedades es lo que garantiza el buen uso de las mismas, por lo tanto, si se puede asociar una variable con los atributos de una clase, es fundamental dominar el concepto para saber donde utilizarlas y como.

CTS (Common Type System)

El CTS agrupa tipos de datos tanto para los tipos por referencia o los tipos por valor. Las clases, interfaces y los delegados son tipos por referencia. Las declaraciones de tipos simples (como los tipos enteros o de punto flotante), las estructuras y las enumeraciones son tipos por valor.

Las variables que se basan en tipos por valor contienen directamente a los valores. La asignación de una variable tipo por valor a otra simplemente copia el valor del contenido. Esto difiere de la asignación de las variables de tipo por referencia, que copia una referencia al objeto, pero no el propio objeto.

Todos los tipos por valor se derivan implícitamente de System.ValueType .

A diferencia de los tipos por referencia, no es posible derivar una tipo existente en un nuevo tipo de un tipo por valor. Sin embargo, al igual que los tipos por referencia, las estructuras pueden implementar interfaces.

A diferencia de los tipos por referencia, no es posible para un tipo por valor contener un valor nulo. Sin embargo, existen formas de para que un tipo por valor acepte valores NULL, como se explicará posteriormente, pero se debe trabajar al tipo de una manera especial.

Cada tipo de valor tiene un constructor implícito predeterminado que inicializa el valor predeterminado para ese tipo.

Definición y uso de los tipos por valor

La mayoría de los lenguajes de programación tienen incorporados tipos de datos como números enteros y de punto flotante, que se copian por valor cuando se pasa a un método. En el Framework de. NET, estos son llamados tipos de valor.

El CLR admite dos grupos de tipos por valor:

- Tipos por valor simples (primitivos): son tipos como System.Int32 y System.Double son idénticos a los tipos de datos primitivos como `int` y `double` (en C #) e `Integer` y `Double` (en Microsoft Visual Basic).
- Tipos por valor definidos por el usuario: cada idioma del Framework de .NET proporciona palabras clave que permiten definir dos clases de tipos por valor definidos por el usuario: estructuras y enumeraciones.

Cuando se crea una instancia de un tipo por valor, se asigna la memoria correspondiente al lugar donde se lo declaró. Por ejemplo, si crea un objeto de tipo por valor como una variable local en un método, se asigna memoria en el stack y se libera automáticamente al final del ámbito o alcance que este posee.

Por el contrario, si se declara un objeto de tipo por valor en una clase, la memoria se asigna como parte del bloque de memoria para el objeto de la clase y se desasigna cuando el objeto es eliminado de la memoria por el recolector de basura.

Estructuras

Una estructura es un tipo por valor definido por el usuario que hereda directamente de la clase System.ValueType. Por lo general, se definen estructuras para representar tipos de datos numéricos que desean asignar y desasignar frecuentemente como valores de moneda, coordenadas o dimensiones.

Para definir un tipo estructura, utilizar `struct` (C #) o `Structure` (Visual Basic) palabras clave. Una estructura puede contener variables, métodos, propiedades y constructores con parámetros, pero no puede tener un constructor por defecto (sin parámetros).

Las estructuras pueden implementar cualquier número de interfaces, pero no puede heredar de forma explícita de una clase o ser utilizada como una clase base.

Los siguientes ejemplos de código muestran cómo definir una estructura para representar una coordenada. La estructura define las variables de instancia, propiedades y métodos. La estructura también define un constructor con parámetros. También remplaza el método ToString que hereda de System.Object por medio de la rescritura.

Ejemplo

C#

```
public struct Coordenadas
       {
       private int _x, _y;
       public Coordenadas(int x, int y) {
              _x = x;
              _y = y;
       }
       public int X { get { return _x; } set { _x = value; } }
       public int Y { get { return _y; } set { _y = value; } }
       public void MoverA(int x, int y)
       {
              _x = x;
              _y = y;
       }

       public override string ToString() {
            String res = ("x = " + _x + ", y = " + _y);
            return res;
        }
    }
```

VB

```
Public Structure Coordenadas
       Dim _x, _y As Integer
       Public Sub New(ByVal x As Integer, ByVal y As Integer)
              _x = x
              _y = y
       End Sub
```

```
        Public Property X() As Integer
                Get
                        Return _x
                End Get
                Set(ByVal value As Integer)
                        _x = value
                End Set
        End Property
        Public Property Y() As Integer
                Get
                        Return _y
                End Get
                Set(ByVal value As Integer)
                        _y = value
                End Set
        End Property

        Public Sub MoverA(ByVal x As Integer, ByVal y As Integer)
                _x = x
                _y = y
        End Sub

        Public Overrides Function ToString() As String
                Dim res As String = ("x = " + CStr(_x) + ", y = " + CStr(_y))
                Return res
        End Function
End Structure
```

Definición de tipos enumeración

Una enumeración es un tipo por valor definido por el usuario que hereda directamente de la clase System.Enum. Normalmente se utilizan los tipos de enumeración para representar un conjunto fijo de opciones tales como los días de la semana o los posibles estados de una conexión de base de datos.

Para definir un tipo de enumeración, utilizar la palabra clave `enum` (C #) o `Enum` (Visual Basic). Un tipo enumeración contiene mnemónicos que representan valores enteros, valores que comienzan en 0 y van aumentando de 1 en 1 por defecto. Las enumeraciones no pueden definir métodos, propiedades o constructores. Por otra parte, las enumeraciones no pueden implementar interfaces y no pueden heredar de forma explícita de una clase o ser utilizadas como una clase base.

Los ejemplos de código siguientes muestran cómo definir una enumeración para representar un punto cardinal. La enumeración define los miembros que representan a los cuatro puntos principales de la brújula.

Ejemplo

C#

```
    public enum PuntoCardinal
    {
        Norte,
```

```
        Sur,
        Este,
        Oeste
    }
```

VB

```
    Public Enum PuntoCardinal
        Norte
        Sur
        Este
        Oeste
    End Enum
```

Tipos por valor simples

Todo elemento que tenga las propiedades de una variable es un tipo. Por ejemplo, un objeto se puede acceder porque es un tipo referenciado (para acceder a un objeto se almacena una referencia en una variable del tipo de la clase del objeto a referenciar) y tiene las mismas propiedades de una variable.

Todos los tipos simples son alias de los tipos del Framework de .NET. Por ejemplo, `int` o `Integer` (para C# y VB respectivamente) es un alias de System.Int32.

Las expresiones constantes, cuyos operandos son constantes de tipo simple, se evalúan en tiempo de compilación.

Los tipos simples se pueden inicializar utilizando literales. Por ejemplo, 'A' es un literal del tipo `char` o `Character` y 2001 es un literal del tipo `int` o `Integer` (para C# y VB respectivamente).

Además de los tipos referenciados, existen los tipos primitivos, los cuales se pueden clasificar según su uso en los siguientes grupos:

- Tipos enteros

C#

Tipo	Rango	Tamaño
sbyte	-128 a 127	Entero de 8 bits con signo
byte	0 a 255	Entero de 8 bits sin signo
short	-32,768 a 32,767	Entero de 16 bits con signo
ushort	0 a 65,535	Entero de 16 bits sin signo
int	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
uint	0 a 4,294,967,295	Entero de 32 bits sin signo
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Entero de 64 bits con signo
ulong	0 a 18,446,744,073,709,551,615	Entero de 64 bits sin signo

VB

Tipo	Rango	Tamaño
SByte	-128 a 127	Entero de 8 bits con signo

Byte	0 a 255	Entero de 8 bits con signo
Short	-32,768 a 32,767	Entero de 16 bits con signo
UShort	0 a 65,535	Entero de 16 bits sin signo
Integer	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
UInteger	0 a 4,294,967,295	Entero de 32 bits sin signo
Long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Entero de 64 bits con signo
ULong	0 a 18,446,744,073,709,551,615	Entero de 64 bits sin signo

- Tipos de punto flotante

C#

Tipo	Rango aproximado	Tamaño	Precisión
float	±1.5e−45 a ±3.4e38	64 bits con signo	7 dígitos
double	±5.0e−324 a ±1.7e308	32 bits con signo	15-16 dígitos

VB

Tipo	Rango aproximado	Tamaño	Precisión
Single	±1.5e−45 a ±3.4e38	64 bits con signo	7 dígitos
Double	±5.0e−324 a ±1.7e308	32 bits con signo	15-16 dígitos

- Tipo texto

C#

Tipo	Rango	Tamaño
char	U+0000 a U+ffff	Caracter Unicode de 16 bits
string	0 a 2 mil millones de caracteres Unicode aproximadamente	2^{31} Caracteres Unicode

VB

Tipo	Rango	Tamaño
Char	U+0000 a U+ffff	Caracter Unicode de 16 bits
String	0 a 2 mil millones de caracteres Unicode	231 Caracteres Unicode

- Tipo lógico

C#

Tipo	Rango	Tamaño
bool	true o false	Depende de la plataforma en que se instala el CLR

VB

Tipo	Rango	Tamaño
Boolean	True o False	Caracter Unicode de 16 bits

- Tipo Decimal

C#

Tipo	Rango	Tamaño
decimal	0 a +/-79,228,162,514,264,337,593,543,950,335 sin punto decimal. 0 a +/-7.9228162514264337593543950335 con 28 lugares decimales	128 bits

VB

Tipo	Rango	Tamaño
Decimal	0 a +/-79,228,162,514,264,337,593,543,950,335 sin punto decimal. 0 a +/-7.9228162514264337593543950335 con 28 lugares decimales	128 bits

- Tipo Fecha

C#

Posee un tipo referenciado llamado DateTime para el manejo de fechas

VB

Tipo	Rango	Tamaño
Date	0:00:00 (medianoche) del 1 de Enero del 0001 a 11:59:59 PM del 31 de Diciembre de 9999	64 bits

Los tipos por valor simples y su correspondencia con los tipos del Framework de .Net

Para que todos los lenguajes que puedan correr en el CLR tengan una compilación en común se debieron definir tipos por valor correspondientes en el CLR para que se mapeen a los tipos específicos de cada lenguaje. Esta correspondencia es tan fuerte que se puede, inclusive, declarar dentro del código de un lenguaje en particular un tipo del CLR y el compilador aceptará la declaración como válida. La siguiente tabla muestra la correspondencia entre los tipos:

Tipo de C#	Tipo de VB	Tipo de .NET Framework
bool	Boolean	**System.Boolean**
byte	Byte	**System.Byte**
sbyte	SByte	**System.SByte**
char	Char	**System.Char**
decimal	Decimal	**System.Decimal**
double	Double	**System.Double**
float	Single	**System.Single**
int	Integer	**System.Int32**
uint	UInteger	**System.UInt32**
long	Long	**System.Int64**

Tipo de C#	Tipo de VB	Tipo de .NET Framework
ulong	ULong	**System.UInt64**
short	Short	**System.Int16**
ushort	UShort	**System.UInt16**
string	String	**System.String**

Comentarios

C#

Existen tres formas de poner comentarios dentro del código y son los siguientes

```
// La doble barra se usa si el comentario abarca una línea

 /* Comienza el comentario
  * La combinación barra – asterisco para comenzar y
  * asterisco - barra para finalizar se utiliza el
  * comentario abarca más de una línea
  * Finaliza el comentario
  */
```

La tercera alternativa sirve para documentar y excede los objetivos de la explicación actual. Sin embargo se introducirá el tema para conocer brevemente el mismo e.

Se puede agregar comentarios dentro de la estructura de código C # y verlas a través de informes Web de comentarios de código. Los comentarios y etiquetas XML mostradas en un informe web de comentario de código están precedidos por la sintaxis de comentario `///`. Normalmente, los comentarios se introducen antes de los tipos definidos por el usuario como una clase, estructura o interfaz. También anteceden a miembros como un atributo, evento, propiedad, método, o una declaración de espacio de nombres. Para utilizar estos comentarios se deben conocer las marcas que los definen, ya que las mismas caracterizan la información que contienen. El siguiente es un ejemplo simple de declaración

```
/// <summary>
///
/// </summary>
```

Un programa de utilidad lee en cometarios como el anterior, lo incluye en archivos HTML que conforman la documentación de la o las clases sobre las que se crearon reportes web.

VB

El lenguaje solo admite los comentarios que abarcan una línea y se señalan con una comilla simple al comienzo de la misma

```
'Esto es un comentario
```

Identificadores

Hay una regla muy simple para determinar si algo es un identificador:

> *Cuando en el código un programador debe decidir que nombre ponerle a un elemento, dicho elemento es un identificador*

Como se puede apreciar, dentro de esta característica entran varios elementos antes mencionados, como ser, nombres de clases, variables, métodos, etc...

Hay otros elementos de los lenguajes todavía no mencionados que son identificadores. Estos se irán descubriendo como tales a medida que se aprenda más del lenguaje siguiendo la simple regla antes mencionada.

> *Es preciso aclarar que cualquier palabra que el lenguaje defina como "reservada o clave", no podrá ser utilizada como identificador.*

Por otro lado, crear un identificador es asignarle un nombre a un elemento que permite definir un lenguaje de programación. Esta asignación de nombres debe seguir ciertas normas prestablecidas en el formato del mismo que variarán de lenguaje a lenguaje. Las reglas a seguir son las siguientes:

- El primer carácter de un identificador debe ser uno de los siguientes:
  - Una letra en mayúsculas (A~Z)
  - Una letra en minúsculas (a~z)
  - El caracter de subrayado (_)
- Del segundo carácter en adelante:
  - Cualquier elemento de los que sirve para el primer carácter
  - Caracteres numéricos (0~9)

Vale la pena mencionar que el espacio en blanco no es un carácter permitido como se puede apreciar, por lo tanto no debe utilizarse para nombrar un identificador.

Además existen ciertas normas estandarizadas para crear identificadores, que si bien no son obligatorias, han sido adoptadas por la mayoría de los programadores y seguirlas ayudan mucho a la legibilidad del código. Algunas de ellas son las siguientes

- Si es una clase, enumeración, evento, excepción, interfaz, espacio de nombres, método o propiedad, el nombre debe comenzar con mayúsculas
- Si es una variable o parámetro, el nombre debe comenzar con minúsculas.
- Si el nombre tiene más de una palabra, a partir de la segunda palabra separarlas sólo comenzando con mayúsculas (primera letra de cada palabra a partir de la segunda)
- Las constantes se escriben con mayúsculas y las palabras se separan con un símbolo de subrayado

No se debe olvidar el hecho de que C# es un lenguaje sensible al caso (diferencia entre mayúsculas y minúsculas), por lo tanto si dos identificadores son iguales en su significado a la lectura pero difieren tan sólo en el caso de una letra, el lenguaje los considerará identificadores distintos. Esto **NO** es cierto para VB porque no diferencia entre mayúsculas y minúsculas.

Literal de asignación

Cunado en el código se asignan valores a variables, constantes o atributos que se escriben como un número en el código, se denominan ***literales de asignación***. Estos se diferencian en la codificación porque cuando el compilador los encuentra, los "traduce" a un valor por defecto según el tipo de datos, antes de asignarlo al tipo declarado. Por lo tanto, para los valores enteros o de punto flotante, el compilador reconoce y traduce el literal de asignación a su tipo por defecto, lo cual se convierte en un error si el valor no se puede convertir automáticamente al tipo declarado (promoción automática). Esto determina que el programador **convierta explícitamente** el tipo del literal de asignación al del que se quiere asignar. El tema se trata con más profundidad posteriormente en este capítulo.

Convenciones para codificar

Si bien muchos temas se irán desarrollando posteriormente, para sentar bases, se pueden mostrar algunos ejemplos de las convenciones que normalmente se utilizan al codificar

Ejemplo

Espacios de nombres:

C#
```
namespace Estructuras
```

VB
```
Namespace Estructuras
```

Clases:

C#
```
class Empleado
```

VB
```
Class Empleado
```

Interfaces:

C#
```
interface IRotador
```

VB
```
Interface IRotador
```

Métodos:

C#
```
void Dibujar()
```

VB
```
Function AgregarInventario()
Sub Dibujar()
```

Variables:

C#
```
private int _alto;
```

VB
```
Private _alto As Integer
```

Constantes:

C#
```
const string MENSAJE = "hola mundo";
```

VB
```
Const MENSAJE As String = "hola mundo"
```

Tipos primitivos

Las declaraciones de las variables se realizan con el siguiente formato:

[<modificador>] *<tipo primitivo> <identificador> [= valor inicial];*

En otras palabras:

- Si aparece "<>", quiere decir "elegir uno entre los posibles"
- Si aparece "[]", quiere decir que es opcional
- Si una palabra o símbolo aparece sin ninguna otra cosa, indica que ponerlo es obligatorio

Este formato debe leerse de la siguiente manera:

- Modificador: Opcional. Es el modificador de visibilidad de la variable (indica como se lo puede utilizar y desde donde). Recordar que **no poner nada** se interpreta como el modificador por defecto (o sea, se declara un modificador implícitamente). Las posibilidades son:

C#	VB	Explicación

C#	VB	Explicación
`public`	`Public`	Especifica que los elementos se pueden acceder desde cualquier parte del código en el mismo proyecto, de otros proyectos que hacen referencia a este, así como de cualquier ensamblado. Se puede utilizar sólo a nivel de módulo, interfaz o espacio de nombres. Esto significa que se puede declarar un elemento público a nivel de un archivo de código fuente, espacio de nombres o dentro de una interfaz, módulo, clase o estructura, pero no en un método.
`private`	`Private`	Especifica que los elementos sólo se pueden acceder desde el interior, el mismo módulo de clase o estructura. Se puede utilizar sólo a nivel de módulo. Esto significa que puede declarar un elemento privado dentro de un módulo, clase o estructura, pero no a nivel de archivo fuente o espacio de nombres, dentro de una interfaz, o en un método. En VB, a nivel de módulo, la instrucción `Dim` sin ningún tipo de acceso específico es equivalente a una declaración privada. Sin embargo, se debe utilizar la palabra clave `Private` para que el código sea más fácil de leer e interpretar.
`protected`	`Protected`	Especifica que los elementos sólo se pueden acceder desde dentro de la misma clase o de una clase derivada de dicha clase. Se puede utilizar sólo a nivel de clase y cuando se declara un miembro de una clase. Esto significa que se puede declarar un elemento protegido en una clase, pero no a nivel de archivo fuente o espacio de nombres, o dentro de una interfaz, módulo, estructura o procedimiento.
`internal`	`Friend`	Especifica que los elementos se pueden acceder desde dentro del mismo ensamblado, pero no desde fuera de este. Se puede utilizar solamente a nivel de módulo, interfaz o espacio de nombres. Esto significa que se puede declarar un elemento de este tipo a nivel de archivo de código fuente, espacio de nombres o dentro de una interfaz, módulo, clase o estructura, pero no en un método.
`protected internal`	`Protected Friend`	Especifica que los elementos se pueden tener acceso desde las clases derivadas, desde dentro del mismo ensamblado o ambos. Se puede utilizar sólo a nivel de clase y cuando se declara un miembro de esta. Significa que se puede declarar un elemento en una clase, pero no a nivel de un archivo fuente, espacio de nombres, o dentro de una interfaz, módulo, estructura o método.

- Es obligatorio poner un tipo, pero se debe elegir uno de los posibles.
- Es obligatorio poner un identificador, se debe elegir el nombre a poner
- Se posee la opción de elegir poner un "=" y un valor si se desea que la variable tenga un valor inicial
- Siempre hay que finalizar la declaración con un ";" que indica fin de línea de programa

Tipos Primitivos: Lógico

Este tipo sólo puede almacenar dos valores: `true` o `false`. Los valores que almacenan son considerados palabras clave en el lenguaje y se deben utilizar para cualquier asignación a variables de este tipo. Generalmente este tipo es utilizado para la toma de decisiones.

Ejemplo

C#

```
bool verdadero = true;
```

VB

```
Dim verdadero As Boolean = true
```

Declara una variable llamada verdadero y le asigna el valor `true` o `True`.

Tipos Primitivos: `char` y `Char`

Este tipo se utiliza para almacenar un solo caracter. Tiene la capacidad de almacenar cualquier caracter en formato Unicode (formato estándar de cualquier caracter en .Net que ocupa 16 bits de almacenamiento). Se debe notar que sólo almacena un caracter, para almacenar una palabra o más de un caracter, se debe utilizar otro tipo de datos que provee el lenguaje, el tipo `string` o `String` (Vb o C#).

Otro punto importante es que cuando se asigna en el código un literal caracter, este debe ir rodeado de comillas simples, como se muestra a continuación.

Ejemplo

C#

```
char c = 'a';
```

VB

```
Dim c As Char = "a"
```

La capacidad de almacenamiento de un tipo `char` es de dos bytes. La diferencia fundamental respecto de los tipos enteros es que éste se maneja sin el signo como valor numérico, lo cual incrementa su capacidad de almacenamiento. Esto es lógico ya que si bien es como los otros tipos enteros internamente, su finalidad es representar caracteres Unicode. La siguiente tabla especifica su capacidad.

C#

Nombre del tipo	Longitud	Rango

char	16 bits	-2^{15} a $2^{15}-1$ ó 0 a 65,535

VB

Nombre del tipo	Longitud	Rango
Char	16 bits	-2^{15} a $2^{15}-1$ ó 0 a 65,535

Se puede utilizar la siguiente notación en los literales de asignación, pero sólo en C#:

Literal de Asignación	Descripción
'a'	La letra a
'\t'	Una tabulación
'\u????'	Un caracter Unicode específico con valor: ????

Ejemplo

- '\u03A6' Es la letra Griega Pi

Nota: Cuando se utiliza un literal de asignación con formato Unicode, el resultado final puede variar porque este puede tener asignado una página de códigos diferente a la propuesta en el ejemplo.

Tipos Primitivos: `string` y `String`

Los literales que lo definen deben ir encerrados entre comillas dobles (""), como se muestra a continuación:

```
"Este es un string"
```

Ejemplo

```
string saludo = "Hola !! \n";
string MensajeDeAlerta = "Usar un número!";
```

Nota: Este tipo tiene una serie de detalles en los cuales se irá profundizando más adelante. La gestión y manipulación de `string` exige un apartado separado el cuál se irá presentando en los próximos capítulos

Literales de caracter

Existen caracteres que tienen un significado especial ya sea si se los utiliza en una variable `char` o como parte de una cadena. Estos caracteres están asociados a valores de control que se pueden utilizar en las salidas o representan información en los ingresos. Como se hará un uso extensivo de los mismos se enumeran en la siguiente tabla

Caracter	Significado
\n	Salto de línea
\t	Tabulador

\r	Retorno de carro

Nota para VB: este lenguaje no maneja caracteres de escape como tales. En su lugar se puede utilizar la función Chr() con el número ASCII que corresponde al carácter cuando el mismo es un carácter que no se puede imprimir, como uno de control. En el caso de la nueva línea, se utilizan dos caracteres, el de alimentación de línea y el retorno de carro.

Para mostrar un ejemplo de utilización se usarán elementos todavía no explicados, las cadenas (`string`), la conversión de un tipo carácter a cadena y la concatenación de cadenas. Estos temas se aclararán posteriormente y se utilizan en este punto sólo a fin de demostrar el uso de estos caracteres especiales

Ejemplo

C#

```
        static void Main(string[] args)
        {
            char n = '\n';
            char t = '\t';
            char r = '\r';
            string cadena1 = "Hola Mundo\n" + n + "!!!!!!";
            string cadena2 = t + "\tHola Mundo";
            string cadena3 = "Hola Mundo\r" + r + "!!!!!!";

            Console.WriteLine(cadena1);
            Console.WriteLine(cadena2);
            Console.WriteLine(cadena3);
            Console.ReadKey();
        }
```

VB

```
    Sub Main()
        Dim lf As Char = Chr(10) 'alimentación de línea
        Dim t As Char = Chr(9) 'tabulación
        Dim r As Char = Chr(13) 'retorno de carro
        Dim cadena1 As String = "Hola Mundo" + lf + r + lf + r + "!!!!!!"
        Dim cadena2 As String = t + "Hola Mundo"
        Dim cadena3 As String = "Hola Mundo" + r + "!!!!!!"

        Console.WriteLine(cadena1)
        Console.WriteLine(cadena2)
        Console.WriteLine(cadena3)
        Console.ReadKey()
    End Sub
```

Este programa produce la siguiente salida, la cual esta afectada por el uso de los caracteres literales especiales

```
Hola Mundo
```

```
!!!!!!
                    Hola Mundo
!!!!!!undo
```

Caracteres de escape en C#

Las combinaciones de caracteres que constan de una barra inversa (\) seguido por una letra o por una combinación de dígitos se denominan "secuencias de escape". Para representar un carácter de nueva línea, comillas simples o ciertos otros caracteres en una constante de caracteres, se debe utilizar las secuencias de escape. Una secuencia de escape se considera como un único carácter, y es por lo tanto válido como carácter constante.

Las secuencias de escape se utilizan normalmente para especificar acciones tales como retornos de carro y los movimientos de tabulación en terminales e impresoras. También se utilizan para proporcionar representaciones literales de caracteres no imprimibles y caracteres que suelen tener un significado especial, como las comillas dobles ("). La siguiente tabla muestra las secuencias de escape ANSI y lo que representan.

Secuencia de escape C#	Representa
\a	Sonido (alerta)
\b	Retroceso
\f	Avance de hoja
\n	Nueva línea
\r	Retorno de carro
\t	tabulación horizontal
\v	tabulación vertical
\'	Comilla simple
\ "	Comilla doble
\\	Barra invertida
\?	Literal de signo de interrogación
\ ooo	Carácter ASCII en notación octal
\x hh	Carácter ASCII en notación hexadecimal
\x hhhh	Caracter Unicode en notación hexadecimal

Tipos Primitivos: Enteros

Todos los tipos enteros en .Net se rigen estrictamente por la aritmética de complemento a dos. Esto implica que son tipos con valores positivos y negativos y el bit más significativo en su formato interno indica el signo. Por lo tanto, sólo se ignora el bit más significativo como el signo del número cuando se realiza una declaración de tipo específica que considere al valor sin signo.

Los tipos enteros se diferencian en su capacidad de almacenamiento y signo como muestran las tablas.

C#

Tipo	Rango	Tamaño
sbyte	-128 a 127	Entero de 8 bits con signo
byte	0 a 255	Entero de 8 bits sin signo
short	-32,768 a 32,767	Entero de 16 bits con signo
ushort	0 a 65,535	Entero de 16 bits sin signo
int	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
uint	0 a 4,294,967,295	Entero de 32 bits sin signo
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Entero de 64 bits con signo
ulong	0 a 18,446,744,073,709,551,615	Entero de 64 bits sin signo

VB

Tipo	Rango	Tamaño
SByte	-128 a 127	Entero de 8 bits con signo
Byte	0 a 255	Entero de 8 bits con signo
Short	-32,768 a 32,767	Entero de 16 bits con signo
UShort	0 a 65,535	Entero de 16 bits sin signo
Integer	-2,147,483,648 a 2,147,483,647	Entero de 32 bits con signo
UInteger	0 a 4,294,967,295	Entero de 32 bits sin signo
Long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Entero de 64 bits con signo
ULong	0 a 18,446,744,073,709,551,615	Entero de 64 bits sin signo

Ejemplo

C#

```
int var1;
int var2 = 6;
```

VB

```
Dim var1 As Integer
Dim var2 As Integer = 6
```

En C#, el lenguaje por defecto convierte a todo literal de asignación a un tipo entero `int`, cunado no incluye un punto dicho literal, por lo tanto cuando se asigna un literal a otro tipo entero diferente se debe especificar con una letra o un molde que convierte el tipo (en inglés se denomina a dicha conversión "*cast*"), como se indica a continuación.

Ejemplo

C#

```
long p = 10L;
long n = (long)10;
short s = (short)5;
byte b = (byte)1;
```

Las formas utilizadas para p y n son análogas. La letra "L" utilizada en la asignación a la variable p es un camino corto que provee el lenguaje. En este caso en particular, como se asigna desde un espacio de almacenamiento menor a una mayor, el lenguaje provee una conversión automática llamada *promoción*.

Tipos Primitivos: Punto Flotante

Existen dos tipos primitivos de punto flotante y se diferencian por su capacidad de almacenamiento. Las variables de punto flotante en .Net pueden utilizar notación científica, por eso no es posible asignar un rango de valores que pueden almacenar. La siguiente tabla muestra los tipos

C#

Tipo	Rango aproximado	Tamaño	Precisión
float	±1.5e−45 a ±3.4e38	64 bits con signo	7 dígitos
double	±5.0e−324 a ±1.7e308	32 bits con signo	15-16 dígitos

VB

Tipo	Rango aproximado	Tamaño	Precisión
Single	±1.5e−45 a ±3.4e38	64 bits con signo	7 dígitos
Double	±5.0e−324 a ±1.7e308	32 bits con signo	15-16 dígitos

C# toma la asignación de un literal numérico en punto flotante por defecto como un double, por lo tanto, si la declaración es para un float, hay que especificarlo de una de las siguientes formas:

```
float f = 10.5F;
float g = (float)10.5;
```

Los tipos por valor en C# pueden incluir en los literales sufijos que sirven para convertir los datos como se muestran en la siguiente tabla:

Tipo por valor	Categoría	Sufijo del tipo
bool	Booleano	
byte	Sin signo, numérico, entero	
char	Sin signo, numérico, entero	
decimal	Numérico, decimal	M o m
double	Numérico, punto flotante	D o d
enum	Enumeración	
float	Numérico, punto flotante	F o f
int	Con signo, numérico, entero	
long	Con signo, numérico, entero	L o l

Tipo por valor	Categoría	Sufijo del tipo
sbyte	Con signo, numérico, entero	
short	Con signo, numérico, entero	
struct	Estructura definida por el usuario	
uint	Sin signo, numérico, entero	U o u
ulong	Sin signo, numérico, entero	UL o ul
ushort	Sin signo, numérico, entero	

Literal de Asignación	Descripción
3.14	Valor simple de punto flotante (un double)
6.02E23	Valor grande de punto flotante
2.718F	Valor del tamaño de un float
123.4E+306D	Valor grande de punto flotante con una D. Redundante pero correcta

Valores Iniciales de las Variables

Todas las variables declaradas, cuando son creadas y almacenadas en memoria, tienen un valor inicial. La siguiente tabla enumera dichos valores

Tipo C#	Tipo VB	Valor por Defecto
bool	Boolean	**false** o **False**
byte	Byte	0
char	Char	'\0'
decimal	Decimal	0.0M
double	Double	0.0D
enum	Enum	El valor producido por la expresión (E)0, donde E es el identificador de la enumeración.
float	Single	0.0F
int	Integer	0
long	Long	0L
sbyte	SByte	0
short	Short	0
struct	Struct	El valor producido al asignar todos los elementos de tipo base a sus valores por defecto y los referenciados a **null** o **Null**.
uint	UInteger	0
ulong	ULong	0
ushort	UShort	0

Inicialización de los tipos de datos

Un ejemplo de manejo de valores de asignación es el que se muestra a continuación

Ejemplo

C#

```
static void Main(string[] args)
{
        // declaración de variables enteras
        int x, y;

        // declaración y asignación de variables de punto flotante
        float z = 3.414f;
        Console.WriteLine(z);

        // declaración y asignación de double
        double w = 3.1415;
        Console.WriteLine(w);

        // declaración y asignación de boolean
        bool verdadero = true;
        Console.WriteLine(verdadero);

        // declaración de variable de caracter
        char c;

        // declaración de variable String
        String str;

        // declaración y asignación de String
        String str1 = "chau";
        Console.WriteLine(str1);

        // asignación de valores a un char
        c = 'A';
        Console.WriteLine(c);

        // asignación de valores a un String
        str = "Hola!";
        Console.WriteLine(str);

        // asignación de valores a int
        x = 6003334;
        y = 1000;
        Console.WriteLine(x);
        Console.WriteLine(y);
        Console.ReadKey();
 }
```

VB

```
Sub Main()
     ' declaración de variables enteras
     Dim x, y As Integer
```

```
        ' declaración y asignación de variables de punto flotante
        Dim z As Single = 3.41400003F
        Console.WriteLine(z)

        ' declaración y asignación de double
        Dim w As Double = 3.1415000000000002
        Console.WriteLine(w)

        'declaración y asignación de boolean
        Dim verdadero As Boolean = True
        Console.WriteLine(verdadero)

        ' declaración de variable de caracter
        Dim c As Char

        ' declaración de variable String
        Dim str As String

        ' declaración y asignación de String
        Dim str1 As String = "chau"
        Console.WriteLine(str1)

        ' asignación de valores a un char
        c = "A"
        Console.WriteLine(c)

        ' asignación de valores a un String
        str = "Hola!"
        Console.WriteLine(str)

        ' asignación de valores a int
        x = 6003334
        y = 1000
        Console.WriteLine(x)
        Console.WriteLine(y)
        Console.ReadKey()
    End Sub
```


Tipos referenciados

La mayoría de los tipos de datos que se definen y utilizan en una aplicación de. NET son tipos por referencia. El CLR es compatible con tres diferentes tipos por referencia:

- Clases
- Interfaces
- Delegados

Cuando se crea una instancia de un tipo por referencia, la memoria para el objeto se asigna en el heap (montón) administrado. El objeto permanece asignado hasta que la aplicación ya no tiene referencias al objeto. En ese momento, el objeto se convierte en disponible (también referido como seleccionable) para la recolección de basura. La próxima vez que el recolector de basura se

ejecuta, se compacta la memoria que utilizan los objetos no referenciados en el heap para esta quede disponible para su uso por otros objetos.

Clases

Las clases definen los datos y la funcionalidad relacionada para las principales entidades orientadas a objetos en una aplicación en el Framework de .NET.

Para definir una clase, utilizar la palabra clave `class` (C#) o la palabra clave `Class` (Visual Basic). Una clase puede contener variables, métodos, propiedades y constructores. También puede heredar de otra clase y se puede implementar cualquier número de interfaces.

Los siguientes ejemplos de código muestran cómo definir una clase para representar un empleado. La clase se define como una clase abstracta para indicar que no se pueden crear objetos del tipo Empleado directamente, sino que se debe definir una clase concreta que derive de esta. Las clases derivadas deben rescribir el método Promover y, opcionalmente, se puede remplazar el método CalcularBono.

Ejemplo

C#

```
public abstract class Empleado
 {
     // Estado.
     private string _nombre;
     private DateTime _fechaDeIngreso;
     private decimal _salario;
     // Constructores.
     public Empleado() { }
     public Empleado(string n, DateTime f, decimal s) {
      _nombre = n;
     _fechaDeIngreso = f;
     _salario = s;

     }
     // Propiedades.
     public string Nombre
     {
         get
         {
             return _nombre;
         }
         set
         {
             _nombre = value;
         }
     }
     public DateTime FechaDeIngreso
     {
         get
         {
```

```
                return _fechaDeIngreso;
            }
            set
            {
                _fechaDeIngreso = value;
            }
        }
        public decimal Salario
        {
            get
            {
                return _salario;
            }
            set
            {
                _salario = value;
            }
        }
        // Métodos.
        public abstract void Promover();
        public virtual decimal CalcularBono() { return 0.0M; }
    }
```

VB

```
Public MustInherit Class Empleado
    ' Estado.
    Private _nombre As String
    Private _fechaDeIngreso As DateTime
    Private _salario As Decimal
    ' Constructores.
    Public Sub New()

    End Sub
    Public Sub New(ByVal n As String, ByVal f As DateTime, ByVal s As Decimal)
        _nombre = n
        _fechaDeIngreso = f
        _salario = s

    End Sub
    ' Propiedades.
    Public Property Nombre() As String
        Get
            Nombre = _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property
    Public Property FechaDeIngreso() As DateTime
        Get
            FechaDeIngreso = _fechaDeIngreso
        End Get
        Set(ByVal value As DateTime)
            _fechaDeIngreso = value
        End Set
```

```
    End Property
    Public Property Salario() As Decimal
        Get
            Salario = _salario
        End Get
        Set(ByVal value As Decimal)
            _salario = value
        End Set
    End Property
    ' Métodos.
    Public MustOverride Sub Promover()

    Public Overridable Function CalcularBono() As Decimal
        Return 0.0
    End Function
End Class
```

Cada vez que se crea una clase se define un nuevo tipo, pero para poder hacer uso de él, se debe crear un objeto. Esto es similar a declarar una variable de un tipo primitivo con la diferencia que la memoria requerida para la operación será determinada por los elementos de dicha la clase.

Por otro lado, el lenguaje utiliza la definición de la clase como molde de la memoria a utilizar y en base a esto se realiza la creación del objeto. En .Net cuando se declara un objeto se utiliza un operador diseñado para este fin: la palabra clave `new` (C#) o la palabra clave `New` (Visual Basic). Es él quien toma la clase como molde o plantilla y reserva la memoria para la creación del objeto.

Sin entrar en detalles acerca de la manera en que lo realiza, el operador efectúa las operaciones necesarias en memoria y luego devuelve el valor de la dirección, o referencia, del lugar en el cual las realizó, de ahí el nombre de estas declaraciones como tipo referenciados.

Evidentemente el valor devuelto debe almacenarse en algún lugar para su posterior uso y ese lugar es una variable de tipo referencia. Como este tipo de variables almacenan la dirección de memoria donde se creo el objeto y como toda clase genera un nuevo tipo, los lenguajes son consistentes al especificar que la declaración deba efectuarse con la clase que define a dicha variable.

De esta manera, con el nombre de la clase seguido del identificador elegido, se crea la variable referencia.

Si se le desea dar una referencia inicial, se debe almacenar el valor devuelto por el operador `new` (C#) o `New` (Visual Basic). La declaración tendría un formato tal que al identificador lo seguirá un operador de asignación "=", el operador `new` (C#) o `New` (Visual Basic) y por último en nombre de la clase que indica el tipo de objeto a crear.

El formato es el siguiente:

C#

<nombre del tipo clase> <identificador>[= new nombre del tipo clase ()];

VB

Dim *<identificador>* As*<nombre del tipo clase> [=*New *nombre del tipo clase ()];*

Al igual que las variables, se pueden crear tipos referenciados sin necesidad de inicializarlos con un valor (referencia al objeto creado por new - C# - o New - Visual Basic -), pero se debe usar el operador posteriormente para crear el objeto cuando se desee utilizarlo.

Ejemplo:

CS

```
DateTime d = new DateTime(2012, 8, 21);
Empleado e1 = new Administrativo("Juan Perez", d, 5000);
Empleado e2 = new Desarrollador("Pedro Sanchez", d, 5000);
```

VB

```
Dim d As New DateTime(2012, 8, 21)
Dim e1 As Empleado = New Administrativo("Juan Perez", d, 5000)
Dim e2 As Empleado = New Desarrollador("Pedro Sanchez", d, 5000)
```

Dónde se almacenan y qué son los métodos

Casi todo código que se puede ejecutar en .Net estará escrito dentro de un elemento especial al que se denomina método.

El código de los métodos se almacena en el segmento de texto, por lo tanto ahí se encuentran todas las instrucciones a ejecutar.

En C# sólo se pueden declarar métodos dentro de clases, cualquier intento de declarar un método fuera de una clase es un error que se detecta en tiempo de compilación. Sin embargo, en VB se admite la declaración de métodos en los módulos. Estos para VB son los subprocesos y las funciones, mientras que para C# sólo son las funciones miembro.

Para acceder a un método con "notación de punto" desde afuera de la clase en la que se lo definió, debe estar declarado en su interfaz, por lo tanto deberá tener un modificador de tipo public o Public (C# o VB). Esto se explicará posteriormente.

Métodos

Los métodos, o como se los llama en otros lenguajes "funciones" o "subprocesos", son los elementos de una clase mediante los cuales se define una operación que un objeto de su tipo puede realizar. La declaración de un método puede adoptar el siguiente formato:

C#

```
[<modificador>] <tipo retornado> <identificador>([lista de argumentos])
{
        [declaraciones y / o métodos a utilizar]
        [return [valor retornado] ;]
}
```

VB

```
[<modificador>] Sub <identificador> ( [lista de argumentos] )
                [declaraciones y / o métodos a utilizar]
End Sub

[<modificador>] Function <identificador>([lista de argumentos]) As <tipo>
        [declaraciones y / o métodos a utilizar]
        [Return [valor retornado]]
End Function
```

El formato quiere decir lo siguiente:

- Modificador: Opcional. Es el modificador de visibilidad del método (indica como se lo puede utilizar y desde donde). Las posibilidades son:

C#	VB	Explicación
public	Public	Especifica que los elementos se pueden acceder desde cualquier parte del código en el mismo proyecto, de otros proyectos que hacen referencia a este, así como de cualquier ensamblado. Se puede utilizar sólo a nivel de módulo, interfaz o espacio de nombres. Esto significa que se puede declarar un elemento público a nivel de un archivo de código fuente, espacio de nombres o dentro de una interfaz, módulo, clase o estructura, pero no en un método.
private	Private	Especifica que los elementos sólo se pueden acceder desde el interior, el mismo módulo de clase o estructura. Se puede utilizar sólo a nivel de módulo. Esto significa que puede declarar un elemento privado dentro de un módulo, clase o estructura, pero no a nivel de archivo fuente o espacio de nombres, dentro de una interfaz, o en un método. En VB, a nivel de módulo, la instrucción Dim sin ningún tipo de acceso específico es equivalente a una declaración privada. Sin embargo, se debe utilizar la palabra clave Private para que el código sea más fácil de leer e interpretar.
protected	Protected	Especifica que los elementos sólo se pueden acceder desde dentro de la misma clase o de una clase derivada de dicha clase. Se puede utilizar sólo a nivel de clase y cuando se declara un miembro de una clase. Esto significa que se puede declarar un elemento protegido en una clase, pero no a nivel de archivo fuente o espacio de nombres, o dentro de una interfaz, módulo, estructura o procedimiento.

C#	VB	Explicación
`internal`	`Friend`	Especifica que los elementos se pueden acceder desde dentro del mismo ensamblado, pero no desde fuera de este. Se puede utilizar solamente a nivel de módulo, interfaz o espacio de nombres. Esto significa que se puede declarar un elemento de este tipo a nivel de archivo de código fuente, espacio de nombres o dentro de una interfaz, módulo, clase o estructura, pero no en un método.
`protected internal`	`Protected Friend`	Especifica que los elementos se pueden tener acceso desde las clases derivadas, desde dentro del mismo ensamblado o ambos. Se puede utilizar sólo a nivel de clase y cuando se declara un miembro de esta. Significa que se puede declarar un elemento en una clase, pero no a nivel de un archivo fuente, espacio de nombres, o dentro de una interfaz, módulo, estructura o método.

- **Tipo retornado**: se debe elegir entre un tipo primitivo o uno referenciado. Si el método no devuelve ningún valor se puede indicar poniendo en este lugar la palabra clave `void` en C# o utilizar un subproceso (`Sub`) en VB.
- **Identificador**: es el nombre que se le asignará al método y mediante el cual se lo invocará
- **Lista de argumentos**: en este lugar se indican los parámetros que recibirá el método. Los argumentos se declaran igual que las declaraciones de variables con la diferencia que no se pone el ";" final. Si el método posee varios argumentos, se deben separar unos de otros con el operador de continuación de declaración (","). Si el método no recibe argumentos, se dejan sólo los paréntesis del mismo.
- **Comienzo del bloque de sentencias**: obligatoriamente poner la llave de apertura de bloque de sentencias en C#. En VB, los bloques de los métodos comienzan con las declaraciones `Sub` y `Function`.
- **Sentencias**: opcionalmente agregar declaraciones e invocaciones a métodos. Existe la posibilidad de no poner nada, a lo que se llama método de cuerpo vacío.
- **Valor retornado**: opcionalmente poner una sentencia `return` o `Return` (C# o VB) para terminar la ejecución del método. Si se indico que el método retorna un valor (recordar que puede ser `void` o un subproceso), este se debe poner a continuación
- **Fin del bloque de sentencias:** Poner obligatoriamente la llave de cierre en C# y `End Sub` o `End Function` según sea un subproceso o función

Ejemplo

C#

```
public string RetornaFecha(string dia, string mes, string anio) {
       string fecha = dia + "-" + mes + "-" +anio;
       return fecha;
}
```

VB

```
Public Function RetornaFecha(ByVal dia As String, _
```

```
                                ByVal mes As String, ByVal anio As String) As String
        Dim fecha As String = dia + "-" + mes + "-" + anio
        Return fecha
    End Function
End Class
```

Este método es público, retorna un `string` y recibe tres `string` como argumentos

Clases

Tienen el siguiente formato:

C#

```
<modificador> class <identificador>
{
        [declaraciones de variables, propiedades y métodos]
}
```

VB

```
<modificador> Class <identificador>
        [declaraciones de variables, subprocesos, propiedades y métodos]
End Class
```

Donde este tipo de declaración se debe leer de la siguiente manera:

- Si aparece “<>”, quiere decir “elegir uno entre los posibles”
- Si aparece “[]”, quiere decir que es opcional
- Si una palabra o símbolo aparece sin ninguna otra cosa, indica que ponerlo es obligatorio

Ejemplo

C#

```
public class Ejemplo
{
        private int var1;
        private int var2;

        public int Var1
        {
                get { return var1; }
                set { var1 = value; }
        }

        public int Metodo()
        {
                return 20;
        }
}
```

VB

```
Public Class Ejemplo
```

```
        Private _var1 As Integer
        Private _var2 As Integer

        Public Property Var1() As Integer
            Get
                Return _var1
            End Get
            Set(ByVal value As Integer)
                _var1 = value
            End Set
        End Property

        Public Function Metodo() As Integer
            Return 20
        End Function
    End Class
```

No se debe olvidar que cuando se declara una clase se crea un nuevo tipo, en este caso el tipo es **Ejemplo**. Las clases son sólo moldes para los objetos de su tipo que se creen. Se puede apreciar en ella la interfaz declarada con el modificador `public` o `Public`. Esto indica que el método `Metodo` será accesible por notación de punto

Uso de Bloques, Espacios en Blanco y Finalización de Sentencia

C#

Una sentencia se compone de una o más líneas terminadas con un punto y coma (;):

```
totales = a + b + c
        + d + e + f;
```

El compilador separa las sentencias o llamados a función por cada “;” que encuentre.

Por otra parte, las sentencias del lenguaje se colocan dentro de bloques, los cuales se definen con un par de llaves. Por lo tanto, se puede definir un bloque como una colección de sentencias limitadas por la apertura y cierre de llaves:

```
{
        x = y + 1;
        y = x + 1;
}
```

Otra característica de los bloques es que definen sus propias **visibilidades**, por lo tanto las variables declaradas dentro de ellos tienen alcance del bloque.

Los bloques se pueden anidar, por lo tanto, cuando se anidan bloques los que están anidados ven las variables de los bloques que los contienen.

Los bloques se utilizan para separaciones sintácticas, como por ejemplo, el contenido de una clase:

Ejemplo

C#

```
public class Fecha
    {
        private int dia;
        private int mes;
        private int anio;
    }
```

Se pueden utilizar los espacios en blanco que se necesiten sin que afecten el código.

VB

Cada sentencia se deberá escribir en un único renglón. Si se desea escribir en un nuevo renglón, por cada instrucción sin terminar, agregar un espacio y un carácter de subrayado. Esto indica continuación de la declaración

En VB los bloques de sentencias están divididos entre instrucciones que demarcan el bloque. Por ejemplo, si se declara una subrutina, el bloque que la determina estará demarcado por las instrucciones `Sub` … `End Sub`. De esta manera, a cada declaración, por ejemplo `Function` o `While`, le corresponde un `End` que demarca el bloque.

Ejemplo

```
Public Class Fecha
    Private dia As Integer
    Private mes As Integer
    Private anio As Integer
End Class
```

Módulos en VB

Se debe tener en cuenta que VB cuenta con un tipo declarativo que C# no posee, el módulo. En él se pueden declaran tanto funciones como subprocesos pero los mismos sólo pertenecerán al módulo y al n o estar definido en ninguna clase son accesibles por cualquier código. Se debe evitar lo máximo posible las declaraciones en módulos, aunque sea necesario en algunas construcciones del lenguaje.

La función de comienzo de programa en VB por defecto se coloca en un módulo, como se explicará posteriormente.

Constructores

Además, las clases tienen un método especial que se denomina constructor. Cuando se declara un objeto a través de crear una instancia de una clase, luego que se reserva en memoria los lugares de almacenamiento necesarios para dicho objeto, se ejecuta siempre el constructor. Cómo es un método especial se diferencia de los otros por las siguientes dos características:

- Se llaman igual que la clase

- Nunca se le pone el valor retornado

Si la clase no posee uno, como en el ejemplo anterior, se utiliza uno por defecto que el lenguaje provee más allá que no se especifique en el código. Además, si se lo desea, se puede poner más de uno.

Cuando se declara un constructor este debe tener el mismo nombre de la clase en C# o el nombre prefijado `New` en VB.

Los constructores permiten que se les pase parámetros, pero cuando se declara un objeto, se le deben pasar tantos parámetros como los que figure en alguno de sus constructores o la declaración será un error. Así si se le quiere poner a la clase anterior un constructor que reciba un entero, quedaría con el siguiente formato.

Ejemplo

C#

```
public class Ejemplo
{
        public Ejemplo(int v) { var1 = v; }
        private int var1;
        private int var2;
        public int getVar1() { return var1; }
}
```

VB

```
Public Class Ejemplo
    Private var1 As Integer
    Private var2 As Integer

    Public Sub New(ByVal v As Integer)
        var1 = v
    End Sub

    Public Function getVar1() As Integer
        Return var1
    End Function
End Class
```

Y la declaración de un objeto de esta clase sería

C#

```
Ejemplo obj = new Ejemplo(8);
```

VB

```
Dim obj As New Ejemplo(8)
```

Un resumen de lo que ocurre cuando se declara un objeto de una clase que tiene constructor es el siguiente:

- Se reserva el espacio en memoria para el objeto
- Se le pasa el parámetro 8 al constructor
- Se empieza a ejecutar el constructor y se asigna el valor del parámetro a la variable var1
- Se termina la ejecución del constructor
- Se devuelve la referencia al objeto y se almacena en la variable obj

Objetos y Mensajes

Para utilizar los servicios que presta un objeto se invocan los elementos declarados en su interfaz pública. En la clase Ejemplo el único elemento en dicha interfaz es el método getVar1. Para poder invocarlo se debe utilizar la notación de punto y la referencia almacenada en la variable obj, ya que esta indica el lugar de almacenamiento del mismo. El código en un programa se vería de la siguiente manera:

Ejemplo

C#

```
int aux;
aux = obj.getVar1();
```

VB

```
Dim aux As Integer
aux = obj.getVar1()
```

En este código se declara la variable entera aux y luego se almacena en ella el valor retornado por el método getVar1.

Cada vez que se utiliza un elemento declarado en la interfaz de la clase (parte pública) se le manda un ***mensaje al objeto***.

Strings

Una cadena (string) es una colección secuencial de caracteres Unicode que se utiliza para representar texto. Un objeto `String` es una colección secuencial de objetos del tipo System.Char que representan una cadena. El valor de un objeto del tipo `String` es el contenido de la colección secuencial y su valor es inmutable (es decir, que es de sólo lectura). La inmutabilidad de las cadenas se tratará posteriormente. El tamaño máximo de un objeto `String` en la memoria es de 2 GB, o alrededor de 1 mil millones de caracteres. Las cadenas de caracteres o “strings” en .Net se manejan con una clase interna del lenguaje, por lo tanto, siempre será un tipo referenciado.
El lenguaje permite dos formas de asignar una cadena de caracteres: cuando se crea el objeto y mediante el operador de asignación.

Se pueden crear instancias de un objeto de tipo `String` de la siguiente manera:

Al asignar un literal de cadena a una variable `string`. Este es el método más comúnmente utilizado para la creación de una cadena. El ejemplo siguiente utiliza la asignación para crear varias

cadenas. Tener en cuenta que en C #, ya que la barra invertida (\) es un carácter de escape, los literales de barras invertidas en una cadena se deben usar con el carácter de escape o la cadena completa deberá ser antecedida por el caracter @.

Ejemplo

C#

```
        static void Main(string[] args)
        {
            string string1 = "Esta es una cadena creada por asignación.";
            Console.WriteLine(string1);
            string string2a = "Este es un path C:\\Documentos\\Reporte1.doc";
            Console.WriteLine(string2a);
            string string2b = @"Este es el mismo path C:\\Documentos\\Reporte1.doc";
            Console.WriteLine(string2b);
            Console.ReadKey();
        }
```

El resultado obtenido por consola es

```
Esta es una cadena creada por asignación.
Este es un path C:\Documentos\Reporte1.doc
Este es el mismo path C:\\Documentos\\Reporte1.doc
```

Sin embargo, en VB, como no existen los caracteres de escape, las cadenas se interpretan como se escriben y el carácter @ no es reconocido

Ejemplo

VB

```
    Sub Main()
        Dim string1 As String = "Esta es una cadena creada por asignación."
        Console.WriteLine(string1)
        Dim string2a As String = "Este es un path C:\\Documentos\\Reporte1.doc"
        Console.WriteLine(string2a)
        Dim string2b As String = "Este es el mismo path C:\Documentos\Reporte1.doc"
        Console.WriteLine(string2b)
        Console.ReadKey()
    End Sub
```

El resultado obtenido por consola es

```
Esta es una cadena creada por asignación.
Este es un path C:\\Documentos\\Reporte1.doc
Este es el mismo path C:\Documentos\Reporte1.doc
```

Nota: los String en .Net son inmutables, lo cual implica que si se asigna una nueva cadena de caracteres a una variable de referencia, como la del ejemplo llamada nombre, se crea un nuevo objeto y se deja al anterior para que sea recolectado como basura

Ocultamiento de la información

Para proteger los datos que se almacenan dentro de un objeto, la clase utiliza la declaración de visibilidad `private` para que no se pueda acceder la variable, ya que esta declaración indica que la variable no pertenece a la interfaz de la clase. Cuando un elemento no pertenece a la interfaz no se puede acceder por notación de punto una vez creado un objeto, sólo lo podrán acceder aquellos elementos que pertenezcan a la clase.
Por lo tanto, se diseñan una serie de métodos como interfaz pública de la clase para acceder a los valores de las variables privadas. De esta manera se gana control sobre el acceso a las mismas para evitar que código fuera de la clase pueda modificar sus valores y causar un cambio en el comportamiento de los servicios que esta brinde. A esos métodos se los suele llamar "accessors" y "mutators" porque permiten el acceso y el cambio de valores pero bajo el control del código que el diseñador de la clase determine.

Ejemplo

C#

```
   public class Persona
    {
        private String primerNombre;
        private String segundoNombre;
        private String apellido;
        private String documento;

        public Persona()
        {
        }

        public String getApellido()
        {
            return apellido;
        }

        public String getDocumento()
        {
            return documento;
        }

        public String getPrimerNombre()
        {
            return primerNombre;
        }

        public String getSegundoNombre()
        {
            return segundoNombre;
        }

        public void setApellido(String str)
        {
            apellido = str;
        }
```

```
        public void setDocumento(String str)
        {
            documento = str;
        }

        public void setPrimerNombre(String str)
        {
            primerNombre = str;
        }

        public void setSegundoNombre(String str)
        {
            segundoNombre = str;
        }
    }
```

VB

```
Public Class Persona
    Private primerNombre As String
    Private segundoNombre As String
    Private apellido As String
    Private documento As String

    Public Sub New()

    End Sub

    Public Function getApellido() As String
        Return apellido
    End Function

    Public Function getDocumento() As String
        Return documento
    End Function

    Public Function getPrimerNombre() As String
        Return primerNombre
    End Function

    Public Function getSegundoNombre() As String
        Return segundoNombre
    End Function

    Public Sub setApellido(ByVal str As String)
        apellido = str
    End Sub

    Public Sub setDocumento(ByVal str As String)
        documento = str
    End Sub

    Public Sub setPrimerNombre(ByVal str As String)
        primerNombre = str
    End Sub
```

```
    Public Sub setSegundoNombre(ByVal str As String)
        segundoNombre = str
    End Sub
End Class
```

Encapsulado

Como se mostró con anterioridad, en toda clase existe una parte pública y una privada. La primera define los elementos de la clase que son accesibles a través de su interfaz. Muchas veces se hace referencia a este hecho como "accesible por el mundo exterior o el universo". Esta frase puede simplificarse de la siguiente manera:

> *Los elementos públicos de una clase serán accesibles por notación de punto una vez creado un objeto de su tipo*

En cambio, la parte privada, define todo lo contrario.

> *Los elementos declarados como privados en una clase no serán accesibles por ningún elemento salvo que este se encuentre en la misma clase.*

Este último concepto es el que permite separar los servicios que brinda un objeto de la forma en que lo hace. Por lo tanto, en otras palabras, un objeto debe verse como una serie de servicios que presta a través de su interfaz y la forma en que lo hace se oculta del mundo exterior porque no es accesible. Cuando una clase oculta una serie de operaciones (métodos y variables de la clase que éstos usan) declarándolos privados y los utiliza posteriormente para brindar un servicio, se dice que este servicio esta encapsulado dentro de la clase.

Se puede decir que el ocultamiento de la información y los métodos privados son en conjunto conocidos como "encapsulado", estén presente uno de ellos o ambos.

Un ejemplo claro de esto es cuando en una clase se oculta la información pero se quiere brindar la posibilidad de acceder a los datos almacenados, ya sea para guardar valores como para leerlos. En este caso se deben crear métodos públicos que cumplan ese rol, como se mostró en el ejemplo anterior.

Propiedades

Los entornos de desarrollo como el Visual Studio suelen poseer diseñadores que interaccionan con el código. Por lo general ese código es provisto por el mismo diseñador aunque el programador puede crear el camino inverso, código que el diseñador reconozca y se pueda interaccionar con el desde el diseñador. Se suelen llamar a las clases desarrolladas de esta manera componentes.

Un componente debe definir propiedades en lugar de campos públicos porque los diseñadores visuales como los del Visual Studio pueden gestionar propiedades por pantalla, pero no los campos (variables miembro de la clase), en el explorador de propiedades.

Las propiedades son como campos inteligentes. Una propiedad en general, tiene un miembro de datos privado acompañado de funciones de acceso y se utilizan sintácticamente como el campo de una clase (aunque las propiedades pueden tener diferentes niveles de acceso, la discusión se centra en el caso más común de acceso público). Las propiedades han estado disponibles en varias ediciones de Visual Basic.

Una definición de la propiedad en general consta de las siguientes dos partes:

- Definición de un miembro de datos privado.
- Definición de una propiedad pública utilizando la sintaxis de declaración de propiedad. Esto asocia la sintaxis del miembro de datos privado con una propiedad pública a través de las funciones de acceso y mutación get y set.

Ejemplo

C#

```
class Entero
{
        private int numero = 0;

        public int Numero
        {
                get { return numero; }
                set { numero = value; }
        }
}
```

VB

```
Public Class Entero
    Private _numero As Integer = 0
    Public Property Numero() As Integer
        Get
            Return _numero
        End Get
        Set(ByVal value As Integer)
            _numero = value
        End Set
    End Property
End Class
```

El término `value` es una palabra clave en la sintaxis de la definición de propiedad. El valor de la variable se asigna a la propiedad en el código de llamada. El tipo por valor debe ser el mismo que el tipo declarado en la propiedad a la que se le asigna.

La manera de utilizar la propiedad es simplemente a través de su nombre. Esto simplifica el código al no ser necesario invocar los métodos `get` ó `Get` y `set` ó `Set` con notación de punto, como se muestra a continuación.

Ejemplo

C#

```
static void Main(string[] args)
 {
     Entero ejemplo = new Entero();
     // Asignar a la propiedad.
     ejemplo.Numero = 5;
     // Obtener de la propiedad.
     int unNumero = ejemplo.Numero;
 }
```

VB

```
     Entero ejemplo = new Entero();
     // Asignar a la propiedad.
     ejemplo.Numero = 5;
     // Obtener de la propiedad.
     int unNumero = ejemplo.Numero;
```


Mientras que una definición de propiedad incluye generalmente un miembro de datos privado, esto no es necesario. El descriptor de acceso `get` ó `Get` podría devolver un valor sin tener acceso a un miembro de datos privado. Un ejemplo es una propiedad cuyo método `get` ó `Get` devuelve la hora del sistema. Las propiedades permiten ocultar datos y los métodos de acceso ocultan la implementación de la propiedad.

Hay algunas diferencias en la sintaxis de la propiedad entre los diferentes lenguajes de programación. Por ejemplo, el término `Property` no es una palabra clave en C #, pero es una palabra clave en Visual Basic.

Los métodos `get` y `set` (`Get` y `Set`) en general no son diferentes de otros métodos. Pueden realizar cualquier lógica del programa, lanzar excepciones, ser sobrescritos y se declara con los modificadores permitidos por el lenguaje de programación. Tener en cuenta, sin embargo, que las propiedades también pueden ser declaradas como estáticas. Si una propiedad es estática, hay limitaciones en lo que los métodos `get` y `set` (`Get` y `Set`) pueden hacer. Su comportamiento es igual a la de los métodos estáticos y este comportamiento se analizará posteriormente

El tipo de una propiedad puede ser un tipo primitivo, una colección de tipos primitivos, un tipo definido por el usuario o una colección de tipos definidos por el usuario. Para todos los tipos primitivos, el Framework de .NET ofrece convertidores de tipos que implementan conversiones de cadena a valor.

Cuando el tipo de datos de una propiedad es una enumeración, un entorno de desarrollo como Microsoft Visual Studio mostrará la propiedad como una lista desplegable en la ventana de Propiedades. Si el tipo de datos de una propiedad es una clase que tiene propiedades, estas propiedades se denominan sub propiedades de la propiedad definitoria. En la ventana

Propiedades de Visual Studio, el usuario puede expandir una propiedad para mostrar sus sub propiedades.

Las propiedades son una herramienta útil más allá que la clase se diseñe como componente o no y favorecen el ocultamiento de la información, razón por la cual se deben utilizar cada vez que sea posible.

Diferencias entre propiedades y variables

Tanto las variables como las propiedades representan valores que se pueden acceder. Sin embargo, hay diferencias en el almacenamiento e implementación.

Variables: Una variable corresponde directamente a una ubicación de memoria. Se define una variable con una instrucción de declaración individual. Una variable puede ser local, definida dentro de un método y disponible dentro de dicho método, o puede ser una variable miembro, de módulo, clase o estructura. Una variable miembro se llama también un campo o atributo cuando se encuentra en una clase, estructura o enumeración.

Propiedades: Una propiedad es un elemento de datos definido en un módulo, clase o estructura. Se define una propiedad con un bloque de código que contiene los métodos `get` y `set` (`Get` y `Set`). A estos métodos se los llaman métodos de propiedad o acceso de propiedades. Además de recuperar o almacenar el valor de la propiedad, también pueden realizar acciones personalizadas, mediante instrucciones de programa regulares conolcadas en el interior de cada `get` y `set` (`Get` y `Set`), como actualizar un contador de acceso.

Las diferencias

Punto de diferencia	Variable	Propiedad
Declaración	Instrucción de declaración individual	Una serie de declaraciones en un bloque de código
Implementación	Ubicación individual en el almacenamiento	El código ejecutable (métodos de propiedad)
Almacenamiento	Directamente asociado con el valor de la variable	Normalmente tiene almacenamiento interno no está disponible fuera de la clase que contiene la propiedad o el módulo
		El valor de la propiedad podría o no existir como un elemento almacenado [1]
Código ejecutable	Ninguno	Debe tener al menos un método
Acceso a lectura y escritura	Lectura / escritura o de sólo lectura	Lectura / escritura, solo lectura o sólo escritura
Las acciones personalizadas (además de aceptar o retornar valores)	No es posible	Puede ser realizado como parte de la configuración o la recuperación del valor de la propiedad

[1]A diferencia de una variable, el valor de una propiedad no se corresponde directamente con un solo elemento de almacenamiento. El almacenamiento puede ser dividido en piezas por comodidad o seguridad, o puede estar almacenado en forma cifrada. En estos casos el método `get` (`Get`) ensambla las piezas o las descifra el valor almacenado, y el método `set` (`Set`) cifra el nuevo valor o lo divide en los distintos almacenamientos que lo constituyen. Un valor de propiedad puede ser efímero, como la hora del día, en cuyo caso el método `get` (`Get`) calcula sobre la marcha cada vez que acceda a la propiedad.

¿Dónde empiezan los programas y por qué?

En .Net, el comienzo de un programa se coloca dentro de una clase en un método para C#, pero puede estar también en un módulo para VB. A diferencia de otros métodos escritos por el programador este tiene un nombre prestablecido: `main ó Main (C# ó VB)`. Este método es el punto de entrada para el comienzo de un programa.

Como no esta definido que un programa deba comenzar en una clase en particular o módulo, esto indica que puede haber muchas clases que posean el método `main`, pero sólo puede haber un método de este tipo por clase..

Ejemplo

C#

```
class Program
{
        static void Main(string[] args)
        {
                Persona p = new Persona();
        }
}
```

VB

```
Module Module1
        Sub Main()
                Dim p As New Persona
        End Sub
End Module
```

Se debe tener en cuenta que sólo puede existir un método `main ó Main` por clase.

Nota: VB admite colocar la función `Main` dentro de una clase. Sin embargo, para que el programa comience en dicha función se debe asignar a las propiedades del proyecto como lugar de inicio

Espacios de nombre

Un espacio de nombres crea un ámbito o visibilidad que identifica unívocamente los elementos declarados en él. Dentro de ellos se pueden declarar uno o varios tipos por valor y referenciados como los que se muestran a continuación:

- Otro espacio de nombres (anidamiento)
- Una clase
- Una interfaz
- Una estructura
- Una enumeración
- Un delegado

Cuando se crea un proyecto en Visual Studio, se crea un espacio de nombres por defecto igual al nombre del proyecto. En C# se puede apreciar la declaración en el archivo fuente que muestra por defecto en entorno (Program.cs). El espacio de nombres por defecto del proyecto se puede cambiar desde las propiedades del proyecto, un cuadro de diálogo específico que asigna valores al proyecto para determinar el entorno del mismo. En VB por otro lado, el entorno genera por defecto un archivo que no es una clase, sino un módulo. Estos también pertenecen a un espacio de nombres, el que se genera por defecto para el proyecto, que si bien no se ve la declaración en el código, la misma se encuentra en las propiedades del proyecto.

Ya sea o no que se declare explícitamente un espacio de nombres en un archivo de código fuente de C #, el compilador agrega un espacio de nombres predeterminado. Este espacio de nombres sin nombre, a veces llamado el espacio de nombres global o por defecto, está presente en cada archivo. Cualquier identificador en el espacio de nombres global está disponible para su uso en cualquier espacio de nombres resolviendo adecuadamente la visibilidad.

Ejemplo

C#

```
using EspaciosDeNombres.Anidado;

namespace EspaciosDeNombres
{
    class Program
    {
        static void Main(string[] args)
        {
            Espacios2 e2 = new Espacios2();
        }
    }
}
```

VB

```
Imports EspaciosDeNombres.Anidado
```

```
Module Module1

    Sub Main()
        Dim e2 As Espacios2 = New Espacios2()
    End Sub

End Module
```

Los espacios de nombres tienen implícitamente acceso público y no es modificable.

Se pueden definir más de un espacio de nombres en un mismo archivo e inclusive anidar las declaraciones. Sin embargo, esta no es una práctica aconsejada ya que disminuye la legibilidad del código. ***Se recomienda ampliamente que las declaraciones de espacios de nombres rodeen al elemento que se desea declarar y que dichos elementos estén contenidos en un archivo con el mismo nombre con el que se identifica al tipo. Esto es, una clase, interfaz, enumeración, estructura o delegado por archivo. Si bien se puede ser flexible con los últimos tres elementos, no se lo debe ser con las clases e interfaces.***

Los espacios de nombre son los lugares lógicos (no físicos, o sea, no tienen relación con un directorio del mismo nombre en el sistema de archivos local) donde se almacenan los tipos por valor o referencia. Se definen antes de la declaración de éstos y cada uno deberá pertenecer a uno o se ubicará en el espacio de nombres por defecto (lo cual no es aconsejable porque acarrea problemas de visibilidades).

Cuando se define un espacio de nombres no es sólo un lugar de almacenamiento virtual, sino que también es un espacio que define una visibilidad. Por lo tanto, si se quiere utilizar una clase que se encuentra en un espacio de nombres, se debe definir explícitamente.

Para acceder a los elementos de un espacio de nombres se debe declarar el acceso mediante la palabra clave `using` ó `Imports` (C# ó VB). Esta tiene dos usos principales

- Como directiva, cuando se utiliza para crear un alias para un espacio de nombres o para importar tipos definidos en otros espacios de nombres.
- Como una declaración, cuando se define una visibilidad en un bloque de sentencias en la que se declara un objeto que va a ser eliminado cuando termina el bloque asociado a `using` ó `Using` (***Notar que a diferencia de C#, en VB no se utiliza la misma palabra clave***).

Cuando los espacios de nombres se encuentran anidados se crean relaciones jerárquicas de tipo padre a hijo. Sin embargo, resolver la visibilidad de acceso al espacio de nombres padre no da acceso a los elementos que contenga sus hijos ni a ningún otro nivel de anidamiento.

Para acceder a una clase, por ejemplo, declarada en el espacio de nombres del ejemplo anterior primero se define que se utiliza el espacio de nombres donde se encuentra la misma. Si el espacio de nombres se encuentra dentro de otro, se debe resolver ***específicamente***. El siguiente ejemplo muestra el uso de una clase en un espacio de nombres anidado.

Ejemplo

C#

```
using espaciosDeNombres.Anidado;

namespace espaciosDeNombres
{
    class Program
    {
        static void Main(string[] args)
        {
            Espacios2 e2 = new Espacios2();
        }
    }
}
```

VB

```
Imports EspaciosDeNombres.Anidado

Module Module1

    Sub Main()
        Dim e2 As Espacios2 = New Espacios2()
    End Sub

End Module
```

Utilizar `using` ó `Imports` (C# ó VB) como directiva para crear un alias se simplifica la escritura de notación de puntos para el acceso a un espacio de nombres. Otra funcionalidad interesante es que permite el manejo de nombres ambiguos, como se muestra en el siguiente ejemplo.

Ejemplo

C#

```
namespace espacio1
{
    class Clase1
    {
        public string RetornaString()
        {
            return "Lamado a espacio1.Clase1";
        }
    }
}

namespace espacio2
{
    class Clase1
    {
        public string RetornaString()
        {
            return "Lamado a espacio2.Clase1";
        }
```

```
    }
}

// Utilizado como directiva.
using System;
// Utilizado como alias para una clase.
using AliasAClase1EnEspacio1 = espacio1.Clase1;
using AliasAClase1EnEspacio2 = espacio2.Clase1;

namespace alias
{
    // Utilizado como directiva:
    using espacio1;
    // Utilizado como directiva:
    using espacio2;

    class MainClass
    {
        static void Main()
        {
            AliasAClase1EnEspacio1 obj = new AliasAClase1EnEspacio1();
            AliasAClase1EnEspacio2 obj2 = new AliasAClase1EnEspacio2();
            Console.WriteLine(obj.RetornaString());
            Console.WriteLine(obj2.RetornaString());
            Console.ReadKey();
        }
    }
}
```

VB

```
Namespace Espacio1
    Public Class Clase1
        Public Function RetornaString() As String
            Return "Lamado a espacio1.Clase1"
        End Function
    End Class
End Namespace

Namespace Espacio2
    Public Class Clase1
        Public Function RetornaString() As String
            Return "Lamado a espacio2.Clase1"
        End Function
    End Class
End Namespace

' Utilizado como directiva.
Imports System
' Utilizado como directiva:
Imports [Alias].Espacio1
' Utilizado como directiva:
Imports [Alias].Espacio2
' Utilizado como alias para una clase.
Imports AliasAClase1EnEspacio1 = [Alias].Espacio1.Clase1
Imports AliasAClase1EnEspacio2 = [Alias].Espacio2.Clase1
```

```
Module Module1
    Sub Main()
        Dim obj As AliasAClase1EnEspacio1 = New AliasAClase1EnEspacio1()
        Dim obj2 As AliasAClase1EnEspacio2 = New AliasAClase1EnEspacio2()
        Console.WriteLine(obj.RetornaString())
        Console.WriteLine(obj2.RetornaString())
        Console.ReadKey()
    End Sub
End Module
```

La otra manera de utilizar `using` ó `Using` (C# ó VB) es generando un bloque de visibilidad de recursos que libera a estos cuando el bloque se termina.

El CLR (Common Language Runtime), libera automáticamente la memoria utilizada para almacenar objetos que ya no son necesarios. La liberación de la memoria no es determinista, y la memoria es liberada cuando el CLR decide llevar a cabo la recolección de basura. Sin embargo, por lo general es la mejor manera de liberar recursos limitados, tales como identificadores de archivos y conexiones de red, tan pronto como sea posible.

El uso de declaración permite al programador especificar que los objetos que utilizan los recursos deben liberarlos. El objetivo previsto para el uso de la declaración debe implementar la interfaz `IDisposable`. Esta interfaz proporciona el método Dispose, el cual es el encargado de liberar los recursos del objeto.

Una declaración `using` ó `Using` puede terminar cuando se alcanza el final de la declaración o si se produce una excepción y el control del programa abandona el bloque de instrucciones antes del final de la declaración.

Ejemplo

C#

```
class Program
{
        static void Main(string[] args)
        {
                using (Horario h = new Horario(1, 7, 30, 11, 00, 4))
                {
                        Console.WriteLine(h.Dia);
                        Console.ReadKey();
                }
        }
}
```

VB

```
Module Module1
        Sub Main()
        Using h As Horario = New Horario(1, 7, 30, 11, 0, 4)
                Console.WriteLine(h.Dia)
        End Using
```

```
        Console.ReadKey()
        End Sub
        End Module
```

Construcción e Inicialización de Objetos

Los objetos, como se mencionó anteriormente, se construyen e inicializan para poder utilizarlos. Para comprender mejor este concepto, se analizará el siguiente ejemplo.

Ejemplo

C#

```
    public class Horario
    {
        private int dia = 1;
        private int horaComienzo = 9;
        private int minutosComienzo = 30;
        private int horaFin = 18;
        private int minutosFin = 30;
        private int turnosPorHora = 4;

        public Horario(int d, int hc, int mc, int hf, int mf, int tph)
        {
            dia = d;
            horaComienzo = hc;
            minutosComienzo = mc;
            horaFin = hf;
            minutosFin = mf;
            turnosPorHora = tph;
        }

        public Horario(Horario f)
        {
            dia = f.dia;
            horaComienzo = f.horaComienzo;
            minutosComienzo = f.minutosComienzo;
            horaFin = f.horaFin;
            minutosFin = f.minutosFin;
            turnosPorHora = f.turnosPorHora;
        }

        public Horario agregar(int masDias)
        {
            Horario nuevaFecha = new Horario(this);
            nuevaFecha.dia = nuevaFecha.dia + masDias;
            return nuevaFecha;
        }

        public void imprimir()
        {
            Console.WriteLine("Horario: ");
            Console.WriteLine("Día: " + dia);
            Console.WriteLine("Hora de comienzo: " + horaComienzo);
            Console.WriteLine("Minutos de comienzo: " + minutosComienzo);
```

```
            Console.WriteLine("Hora de fin: " + horaFin);
            Console.WriteLine("Minutos de fin: " + minutosFin);
            Console.WriteLine("Turnos por hora:" + turnosPorHora);
        }

        public int Dia
        {
            get { return dia; }
        }

        public int HoraComienzo
        {
            get { return horaComienzo; }
        }

        public int MinutosComienzo
        {
            get { return minutosComienzo; }
        }

        public int HoraFin
        {
            get { return horaFin; }
        }

        public int MinutosFin
        {
            get { return minutosFin; }
        }

        public int TurnosPorHora
        {
            get { return turnosPorHora; }
        }
    }
```

VB

```
Public Class Horario
    Private _dia As Integer = 1
    Private _horaComienzo As Integer = 9
    Private _minutosComienzo As Integer = 30
    Private _horaFin As Integer = 18
    Private _minutosFin As Integer = 30
    Private _turnosPorHora As Integer = 4

    Public Sub New(ByVal d As Integer, ByVal hc As Integer, ByVal mc As Integer,
ByVal hf As Integer, ByVal mf As Integer, ByVal tph As Integer)
        _dia = d
        _horaComienzo = hc
        _minutosComienzo = mc
        _horaFin = hf
        _minutosFin = mf
        _turnosPorHora = tph
    End Sub
```

```
    Public Sub New(ByVal f As Horario)
        _dia = f._dia
        _horaComienzo = f._horaComienzo
        _minutosComienzo = f._minutosComienzo
        _horaFin = f._horaFin
        _minutosFin = f._minutosFin
        _turnosPorHora = f._turnosPorHora
    End Sub

    Public Function agregarDias(ByVal masDias As Integer) As Horario
        Dim nuevaFecha As Horario = New Horario(Me)
        nuevaFecha._dia = nuevaFecha._dia + masDias
        Return nuevaFecha
    End Function

    Public Sub imprimir()
        ' Se usa CStr para convertir a String
        Console.WriteLine("Horario: ")
        Console.WriteLine("Día: " + CStr(_dia))
        Console.WriteLine("Hora de comienzo: " + CStr(_horaComienzo))
        Console.WriteLine("Minutos de comienzo: " + CStr(_minutosComienzo))
        Console.WriteLine("Hora de fin: " + CStr(_horaFin))
        Console.WriteLine("Minutos de fin: " + CStr(_minutosFin))
        Console.WriteLine("Turnos por hora:" + CStr(_turnosPorHora))
    End Sub

    Public ReadOnly Property Dia() As Integer
        Get
            Return _dia
        End Get
    End Property

    Public ReadOnly Property HoraComienzo() As Integer
        Get
            Return _horaComienzo
        End Get
    End Property
    Public ReadOnly Property MinutosComienzo() As Integer
        Get
            Return _minutosComienzo
        End Get
    End Property
    Public ReadOnly Property HoraFin() As Integer
        Get
            Return _horaFin
        End Get
    End Property
    Public ReadOnly Property MinutosFin() As Integer
        Get
            Return _minutosFin
        End Get
    End Property
    Public ReadOnly Property TurnosPorHora() As Integer
        Get
            Return _turnosPorHora
```

```
        End Get
    End Property

End Class
```

Y una clase cliente de la anterior que utiliza uno de sus servicios

C#

```
namespace objetos
{
    class Program
    {
        static void Main(string[] args)
        {
            Horario h = new Horario(2, 8, 45, 12, 45, 3);
            h.imprimir();
            Console.WriteLine("--------------");
            Horario nuevoHorario = h.agregar(3);
            nuevoHorario.imprimir();
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim h As New Horario(2, 8, 45, 12, 45, 3)
        h.imprimir()
        Console.WriteLine("--------------")
        Dim nuevoHorario As Horario = h.agregarDias(3)
        nuevoHorario.imprimir()
        Console.ReadKey()
    End Sub

End Module
```

Una declaración del tipo:

C#

```
            Horario h = new Horario(2, 8, 45, 12, 45, 3);
```

VB

```
            Dim h As New Horario(2, 8, 45, 12, 45, 3)
```

Aloja espacio en memoria para un nuevo objeto de la siguiente manera:

1. Se reserva el espacio en memoria para el nuevo objeto
2. Todas las variables de instancia son inicializadas a sus valores por defecto
3. Los atributos que tienen asignados valores explícitamente son inicializados

4. Se ejecuta el constructor
5. Se le asigna el valor a la variable de referencia

Alojamiento en Memoria

Una declaración aloja espacio en memoria sólo para la variable de referencia. En el ejemplo anterior previa ejecución del operador `new` ó `New` se aloja espacio solo para h. Por lo tanto, si la declaración se realizó en un método, la variable h se alojará en el stack. Si la declaración se hizo como una variable de instancia de la clase, h se alojará en el heap. En el ejemplo anterior como h se encuentra dentro de `main` o `Main`, se alojará en el espacio dedicado para `main` o `Main` en el stack.
Un ejemplo de cómo es el proceso de construcción de un objeto, más allá de la ubicación de la variable h (heap o stack) es el siguiente:

Nota: esta sección muestra en código en C# por simplicidad y espacio. Los conceptos son los mismos para ambos lenguajes y el código utilizado es análogo a excepción de las instrucciones en particular

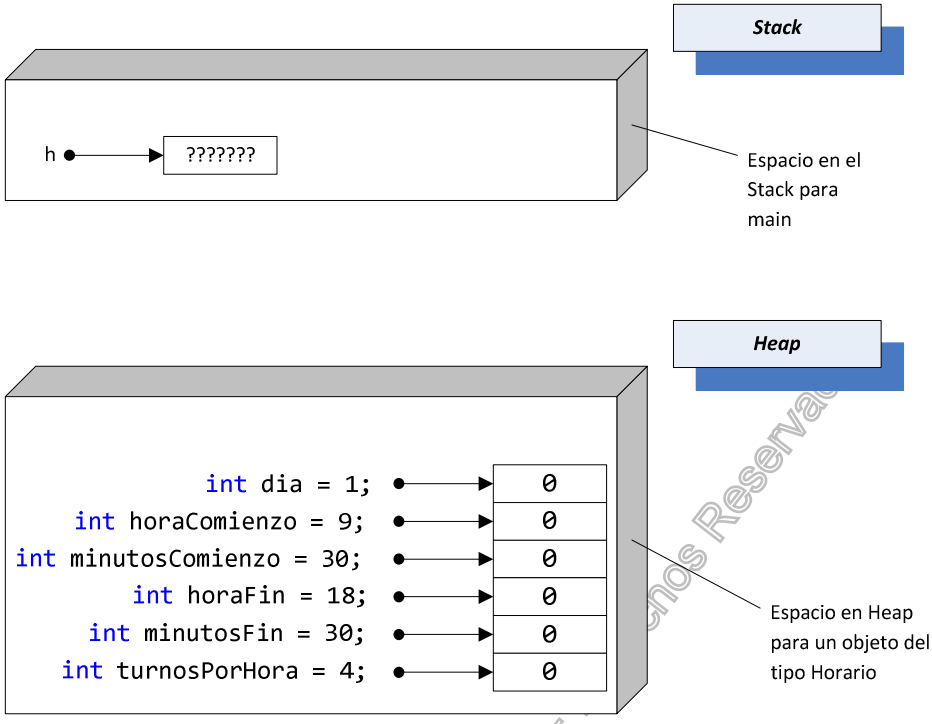
1. Aloja espacio en el stack sólo para la variable de referencia

```
Horario h = new Horario(2, 8, 45, 12, 45, 3);
```

Cuando se resuelve la declaración de la referencia, en memoria sólo se tiene

2. Cuando se ejecuta el operador `new`, lo primero que se hace es alojar espacio para el objeto en el heap.

```
Horario h = new Horario(2, 8, 45, 12, 45, 3);
```



3. El próximo paso en la construcción del objeto, es la inicialización de los atributos. Como la declaración que crea el objeto indica que se le pasan valores al constructor, éste intentará asignarlos según un orden de inicialización prestablecido. El lugar mediante el cual se le pasan los valores al constructor es entre los paréntesis que están a continuación del nombre de la clase a la derecha de la declaración:

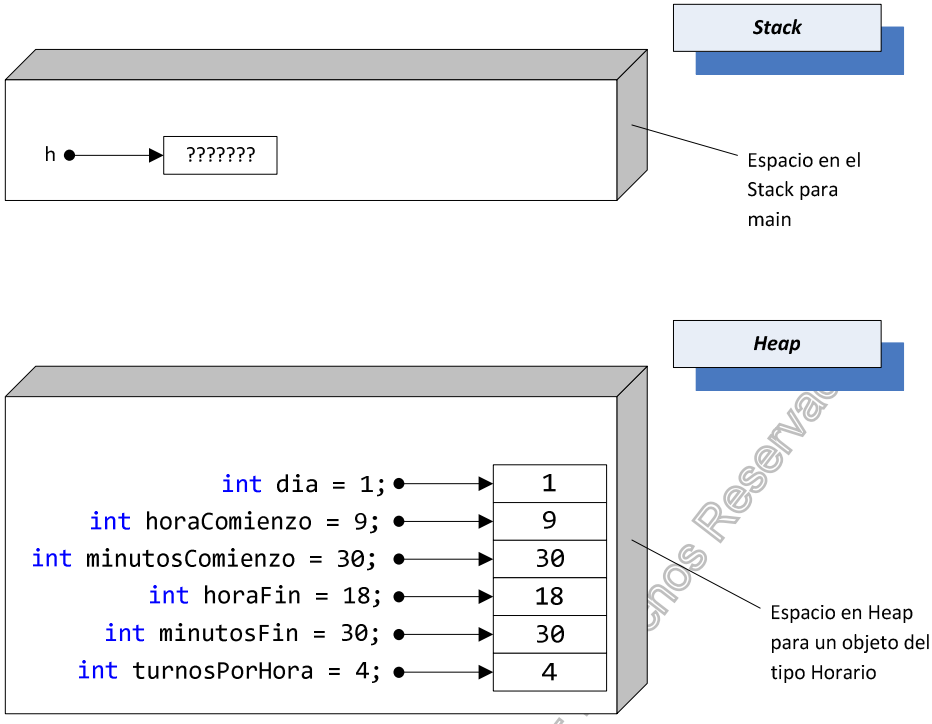
```
Horario h = new Horario(2, 8, 45, 12, 45, 3);
```

4. Cómo la clase declara variables de instancia con valores iniciales, el próximo paso es inicializar los valores declarados. Estos fueron declarados en la clase de la siguiente manera:

```
private int dia = 1;
private int horaComienzo = 9;
private int minutosComienzo = 30;
private int horaFin = 18;
private int minutosFin = 30;
private int turnosPorHora = 4;
```

Por lo tanto, se asignan los valores de inicialización declarados en la clase para los atributos (llamados valores por defecto)

El siguiente gráfico muestra la realización de estos pasos



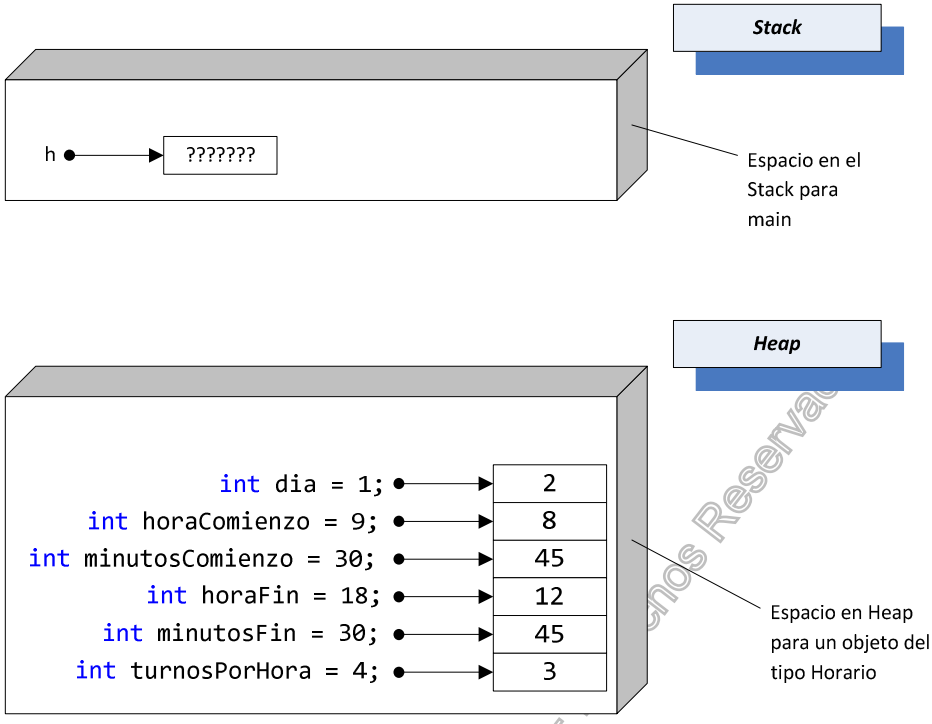
5. Una vez inicializados los valores de las variables de instancia, se ejecuta el conjunto de sentencias que se encuentran dentro del constructor que corresponda, por ejemplo:

```
Horario h = new Horario(2, 8, 45, 12, 45, 3);
```

Deberá pasar los parámetros 2, 8, 45, 12, 45, 3 y empezar a ejecutar lo que está declarado dentro del constructor como se muestra a continuación

```
public Horario(int d, int hc, int mc, int hf, int mf, int tph)
{
    dia = d;
    horaComienzo = hc;
    minutosComienzo = mc;
    horaFin = hf;
    minutosFin = mf;
    turnosPorHora = tph;
}
```

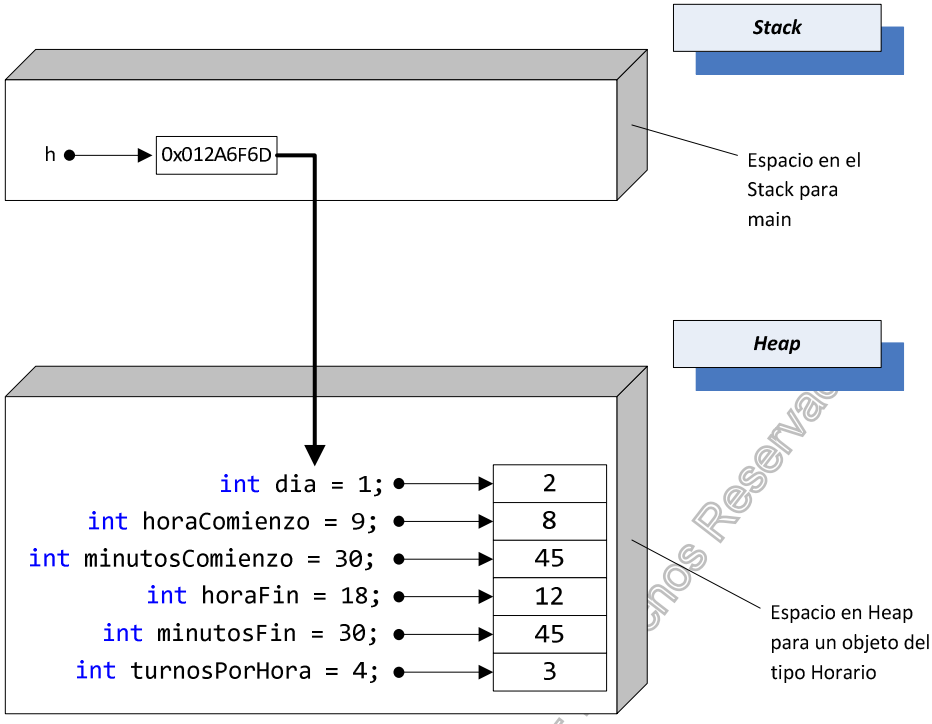
El siguiente gráfico muestra el estado de las variables de instancia una vez terminada la ejecución de sentencias del constructor de la clase



6. Cuando se termina de ejecutar el constructor, el objeto se encuentra creado en el heap. Lo único que queda por hacer es retornar la referencia que permite luego encontrarlo cuando se opera con él. Por lo tanto, se asigna el objeto que acaba de ser creado a la variable de referencia h

```
Horario h = new Horario(2, 8, 45, 12, 45, 3);
```

El siguiente gráfico ilustra el proceso



Asignación de tipos referenciados

Como se demostró anteriormente, un objeto se crea mediante el operador `new` y su referencia se almacena en la variable declarada del tipo de la clase. Esta variable es tan sólo una referencia, ***no el objeto en sí mismo***, el cual se encuentra almacenado en el heap.

Si, por ejemplo, se tiene el siguiente código:

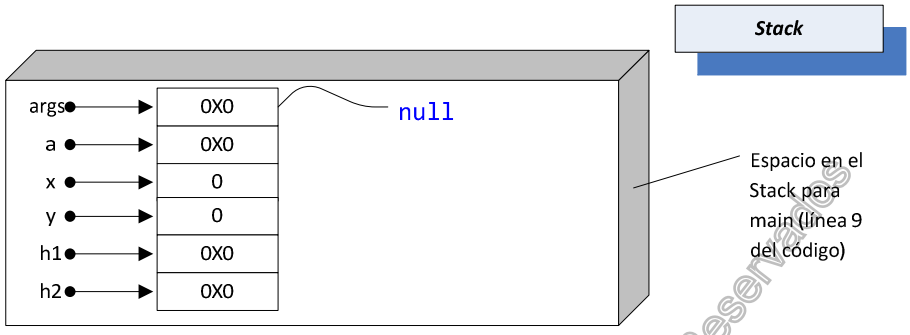
```
1.      public class AsignacionReferencias{

2.          public void metodo(Horario h){
3.                  int x = 7;
4.                  int y = x;
5.                  Horario h1 = new Horario(2, 8, 45, 12, 45, 3);
6.                  Horario h2 = h;
7.                  h2 = new Horario(2, 8, 45, 12, 45, 3);
8.              }

9.          static void Main(string[] args){
10.             AsignacionReferencias a = new AsignacionReferencias();
11.                 int x = 7;
12.                 int y = x;
13.                 Horario h1 = new Horario(2, 8, 45, 12, 45, 3);
14.                 Horario h2 = h1;
15.                 h2 = new Horario(2, 8, 45, 12, 45, 3);
16.                 a.metodo(h2);
17.         }
18.     }
```

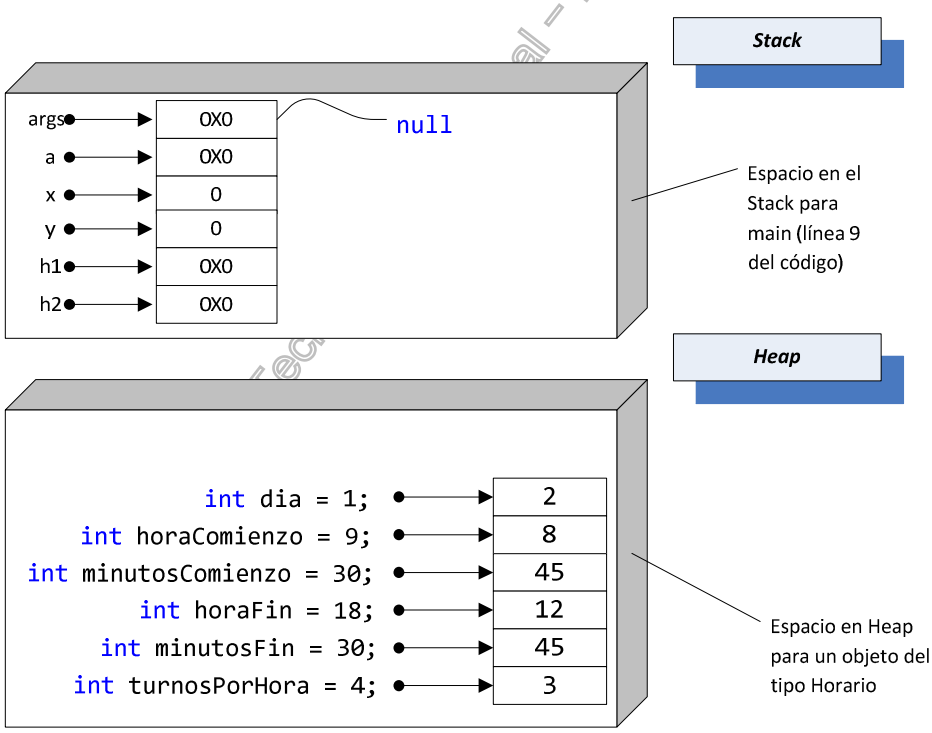
Se analizará el programa por cada línea de ejecución para comprender el manejo de memoria que se realiza a medida que se ejecuta

```
9.        public class AsignacionReferencias{
```



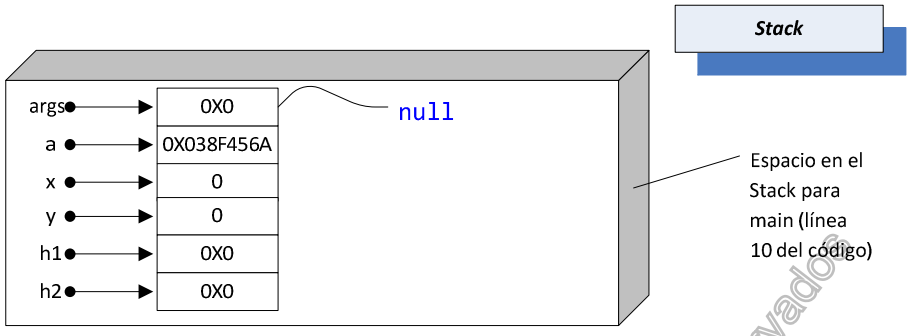
Cuando comienza el programa se reserva espacio en el stack para almacenar las variables locales a **main**. Como no recibe ningún argumento por línea de comandos en la variable args (esto se explicará posteriormente) el valor almacenado que queda es 0 en hexadecimal (`null`)

```
10.             AsignacionReferencias a = new AsignacionReferencias();
```



Se ejecuta el operador `new` el cual reserva y asigna valores de memoria en el heap según se explicó anteriormente. Como en este caso en particular la clase AsignacionReferencias no tiene variables de instancia, se reserva un lugar en el heap que representa el espacio de memoria asignado al objeto, pero no ocurre ningún proceso de reserva de memoria, inicialización o asignación de

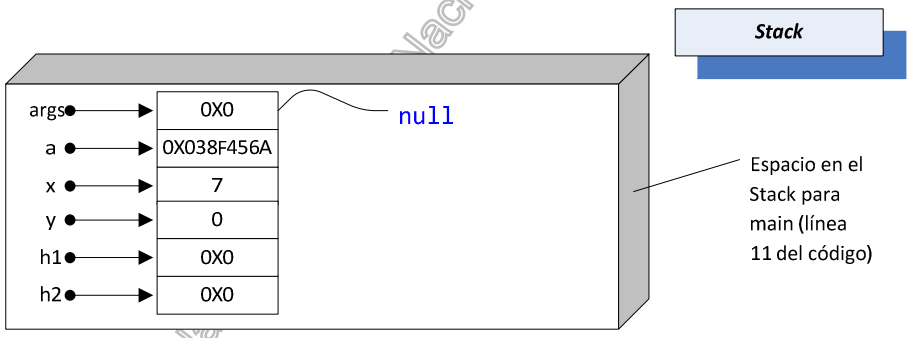
valores porque no hay atributos. El valor de la dirección de memoria, sin embargo, igualmente lo retorna el operador `new` y se asigna a la variable de referencia **a** que se encuentra en el stack.



Nota: Los valores hexadecimal que se presentan como representaciones de espacios de memoria son a fines didácticos y se deben considerar como suposiciones. El algoritmo interno que se dedica a establecer las direcciones de asignación en la máquina virtual es interno y se llama gestor de heap (heap manager). Éste es quien determina la dirección de memoria donde se almacenará el objeto.

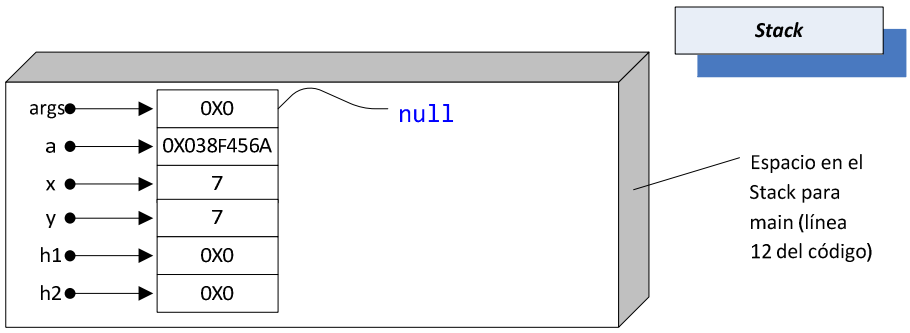
```
11.             int x = 7;
```

Se inicializa la variable x en el stack asignándole el valor 7

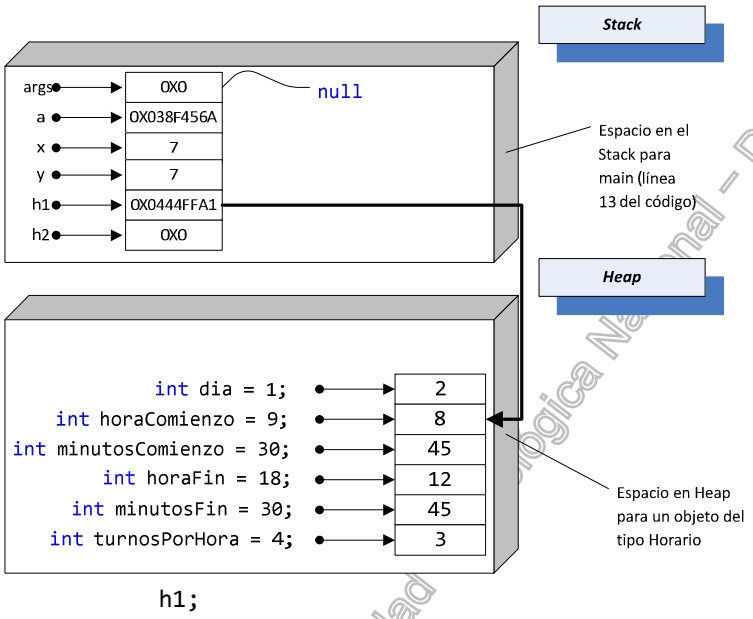



```
12.             int y = x;
```

Se inicializa la variable y en el stack asignándole el valor que almacena la variable x

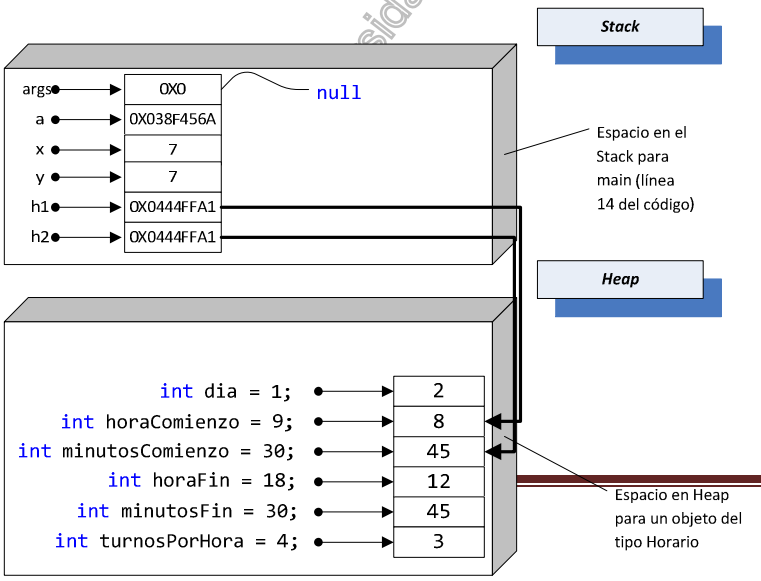


13. `Horario h1 = new Horario(2, 8, 45, 12, 45, 3);`



Se ejecuta el operador `new` el cual reserva y asigna valores de memoria en el heap según se explicó anteriormente. Como en este caso en particular la clase Horario tiene variables de instancia, se asigna un lugar en el heap que representa el espacio de memoria asignado al objeto, y ocurre el proceso de reserva de memoria, inicialización o asignación de valores. El valor de la dirección de memoria lo retorna el operador `new` y se asigna a la variable de referencia **h** que se encuentra en el stack.

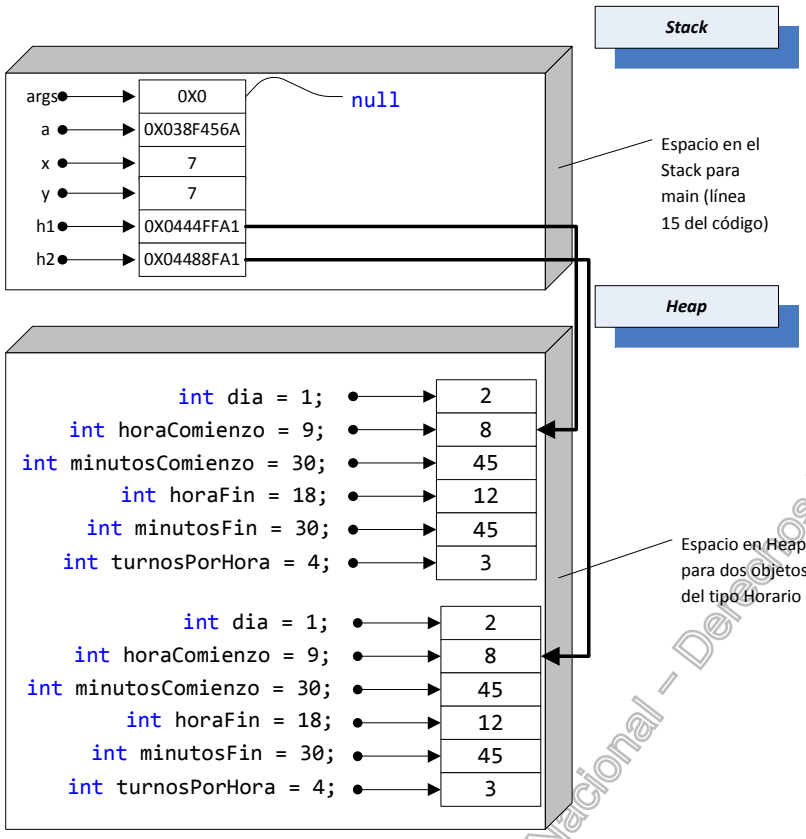
14. `Horario h2 = h1;`



El resultado de la ejecución de esta línea es que se asignan dos referencias al mismo objeto del tipo Horario, por lo tanto dos variables de referencia a objetos del tipo Horario apuntan al mismo lugar de memoria en el heap. Por lo tanto, es muy importante no confundir el concepto de “variable de referencia” con “objeto”. La variable

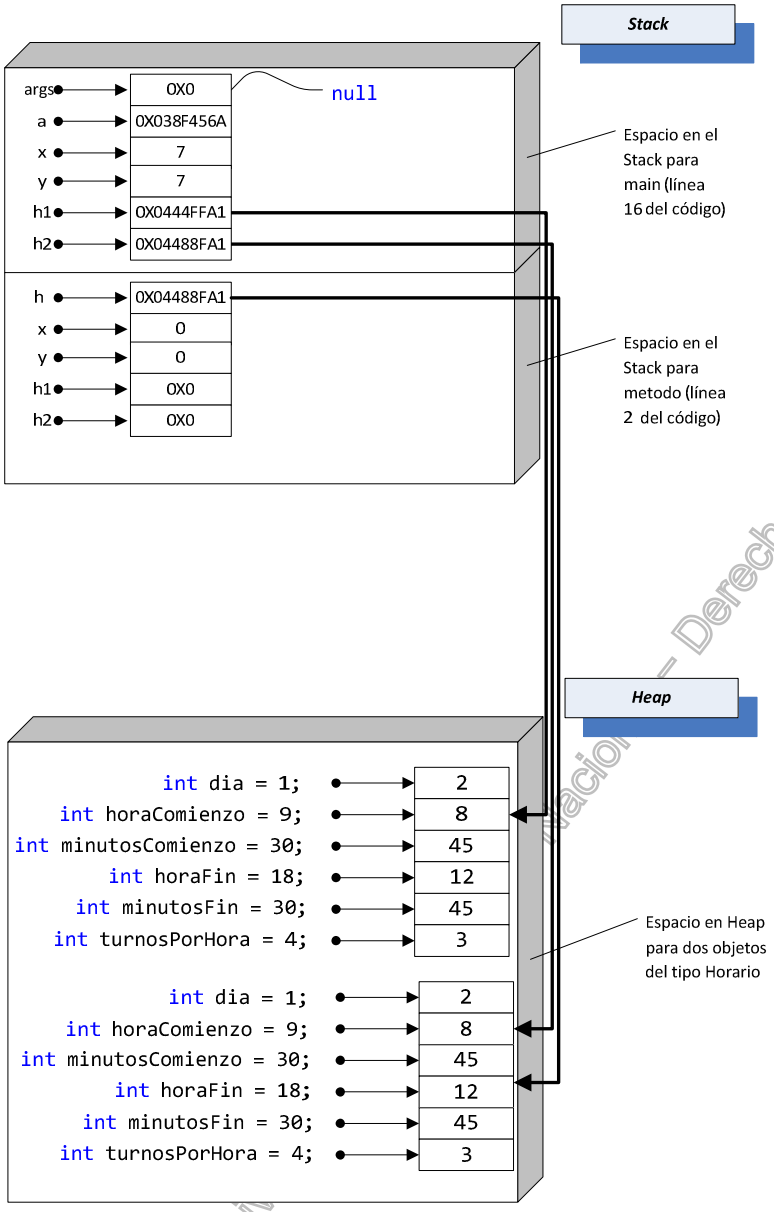
de referencia tan sólo indica el lugar donde se almacena el objeto.

```
15.             h2 = new Horario(2, 8, 45, 12, 45, 3);
```



Para tener dos objetos diferentes con los mismos atributos debe ejecutarse `new` dos veces con los mismos valores como parámetros para el constructor. Sino, tendrán cada uno sus valores iniciales

```
16.              a.metodo(h2);
```



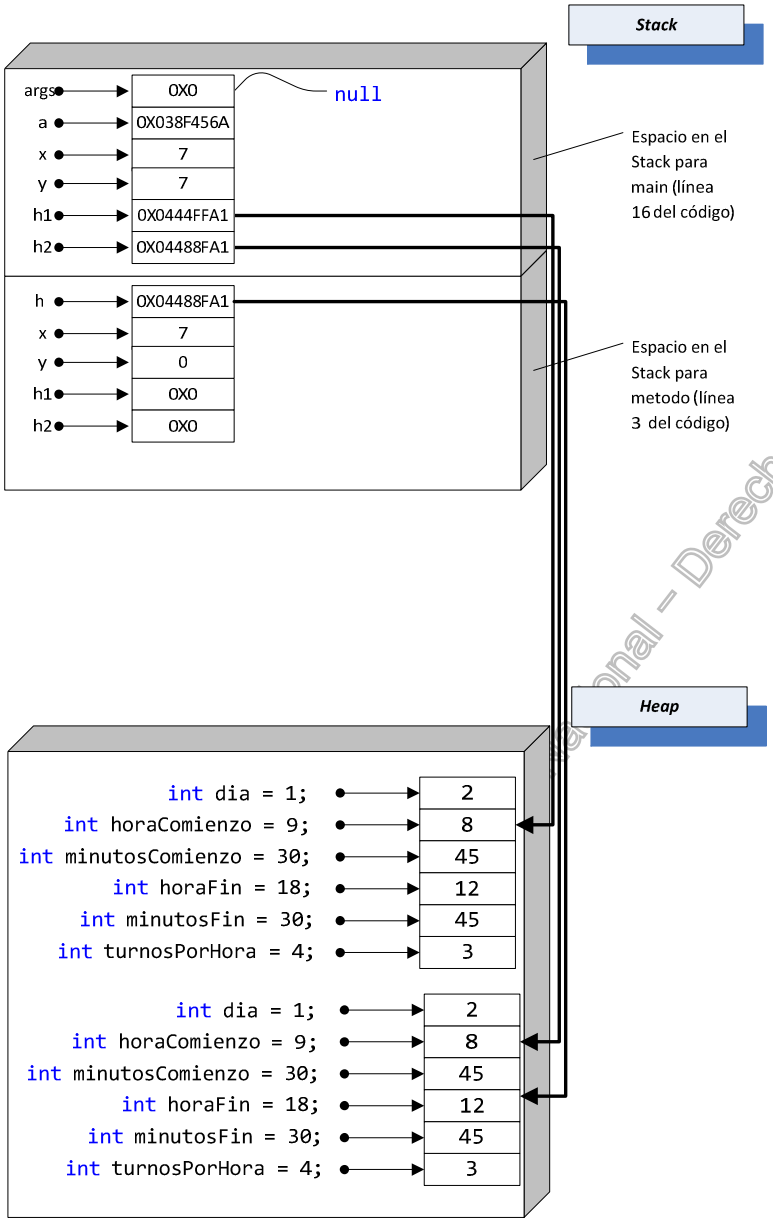
Se invoca al método del objeto **a** que se define en la línea 2. Esto causa que se reserve espacio en el stack para el nuevo método a ejecutar y se almacene en dicho espacio los parámetros y variables locales del método.

```
2.  public void metodo(Horario h){
```

Al empezar la ejecución del método, se encuentra reservado el espacio en el stack y las variables locales se inicializan a sus valores por defecto. El parámetro recibe un valor con la dirección de un objeto de tipo Horario (el que se encuentra almacenado en la variable **h2**) y comienza la ejecución con dicho valor asignado. En este punto hay dos referencias apuntando a un mismo objeto.

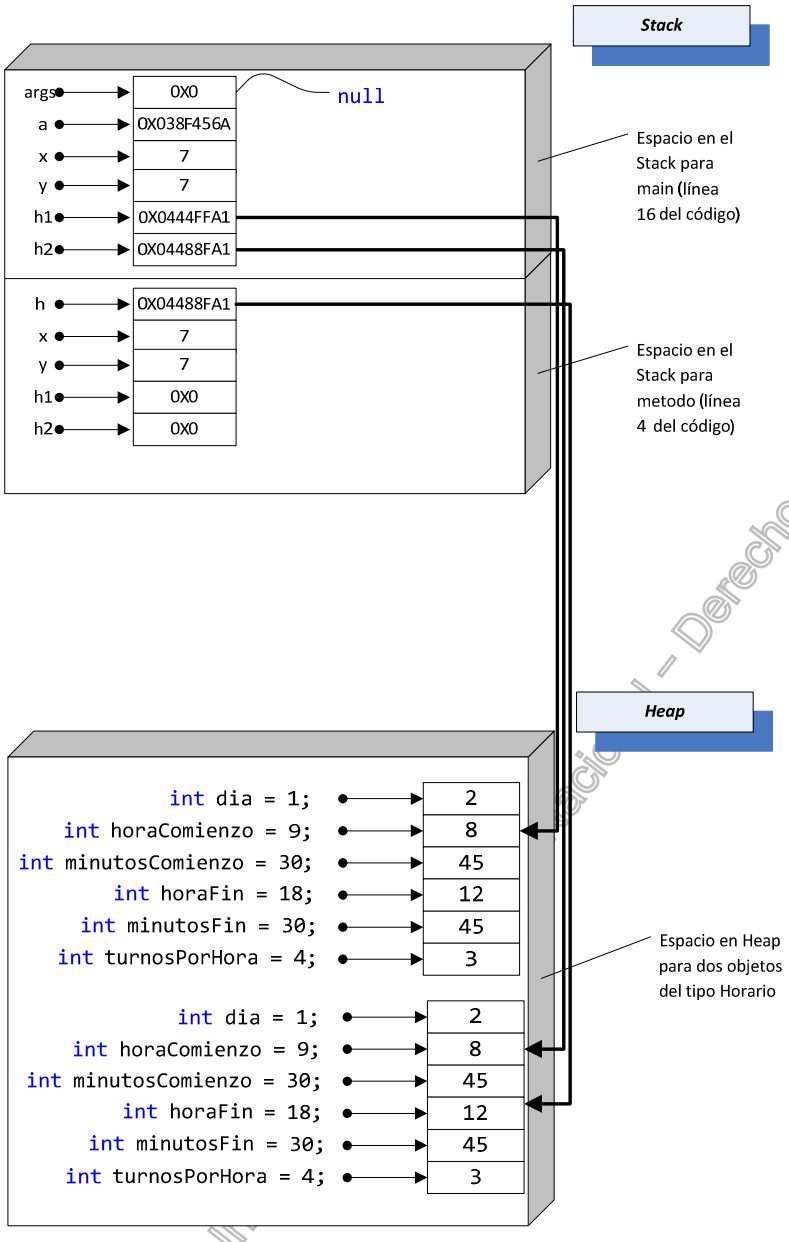
Notar que existen dos referencias a un mismo objeto. Una se encuentra almacenada en el espacio reservado en el stack para **Main** y la otra se encuentra en el espacio reservado para **método**.

```
3.              int x = 7;
```



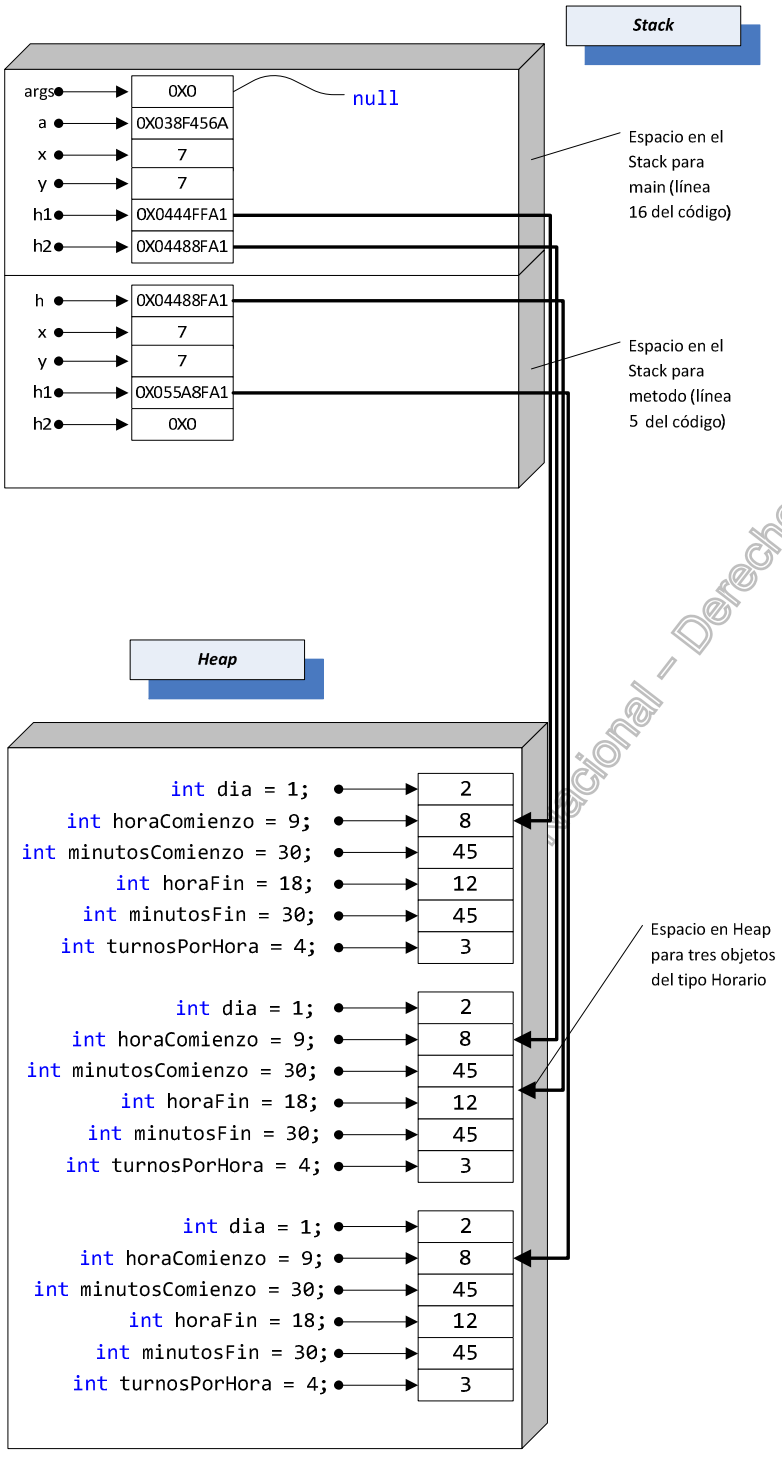
Se asigna en el espacio del stack el valor 7 a la variable **x**. Notar que no sólo esta variable no tiene nada que ver con la variable definida en **Main** sino que se encuentra en un espacio de memoria totalmente diferente

```
4.              int y = x;
```



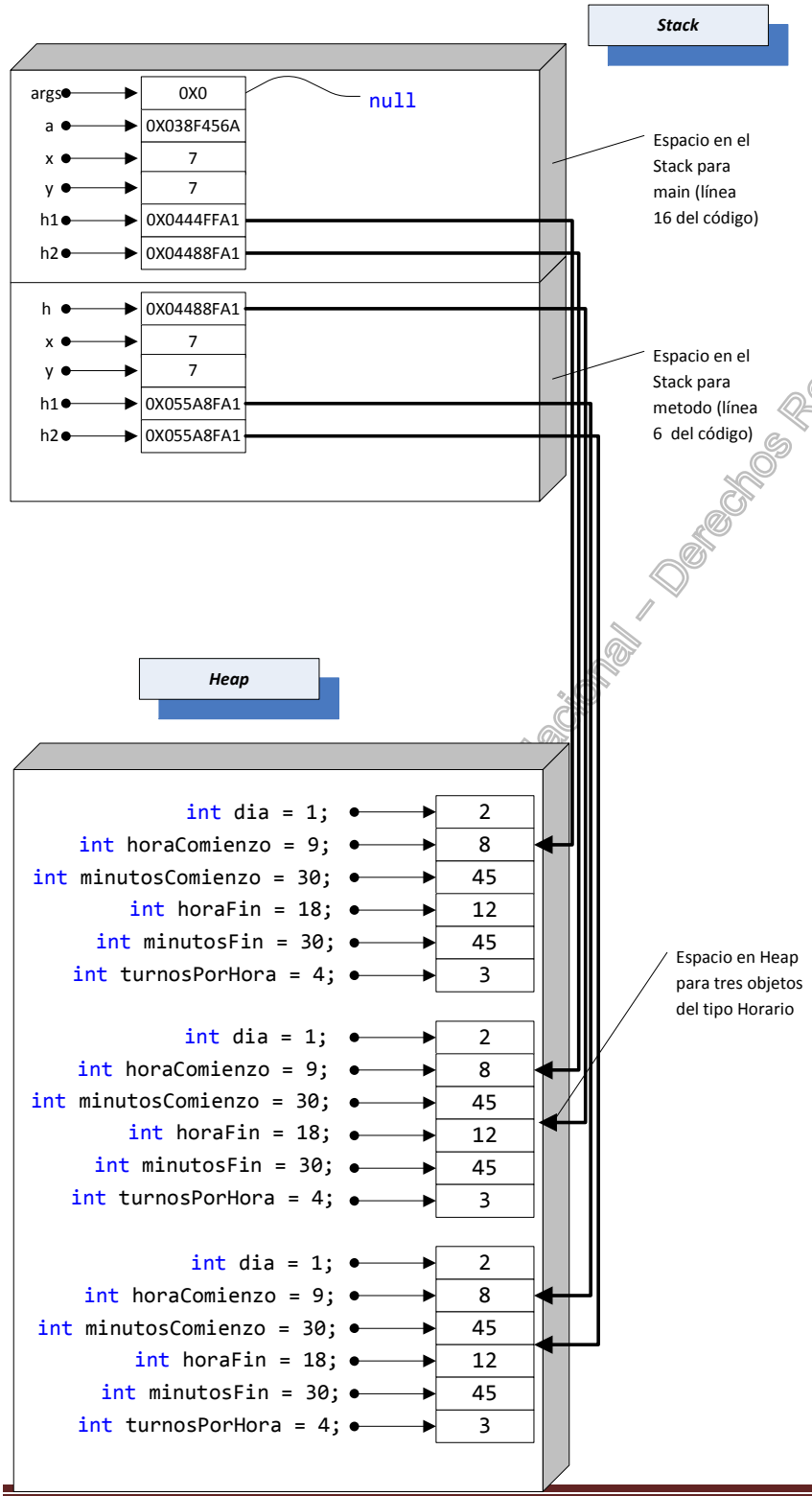
Al igual que en M**ain**, se le asigna el valor que almacena **x** a la variable **y**. Notar nuevamente los distintos lugares de memoria ocupados.

```
5.              Horario h1 = new Horario(2, 8, 45, 12, 45, 3);
```



Se asigna a la variable de referencia **h1** la dirección de un nuevo objeto del tipo Horario que se encuentra en el heap. En este punto el stack y el heap quedan en el estado que muestra el gráfico.

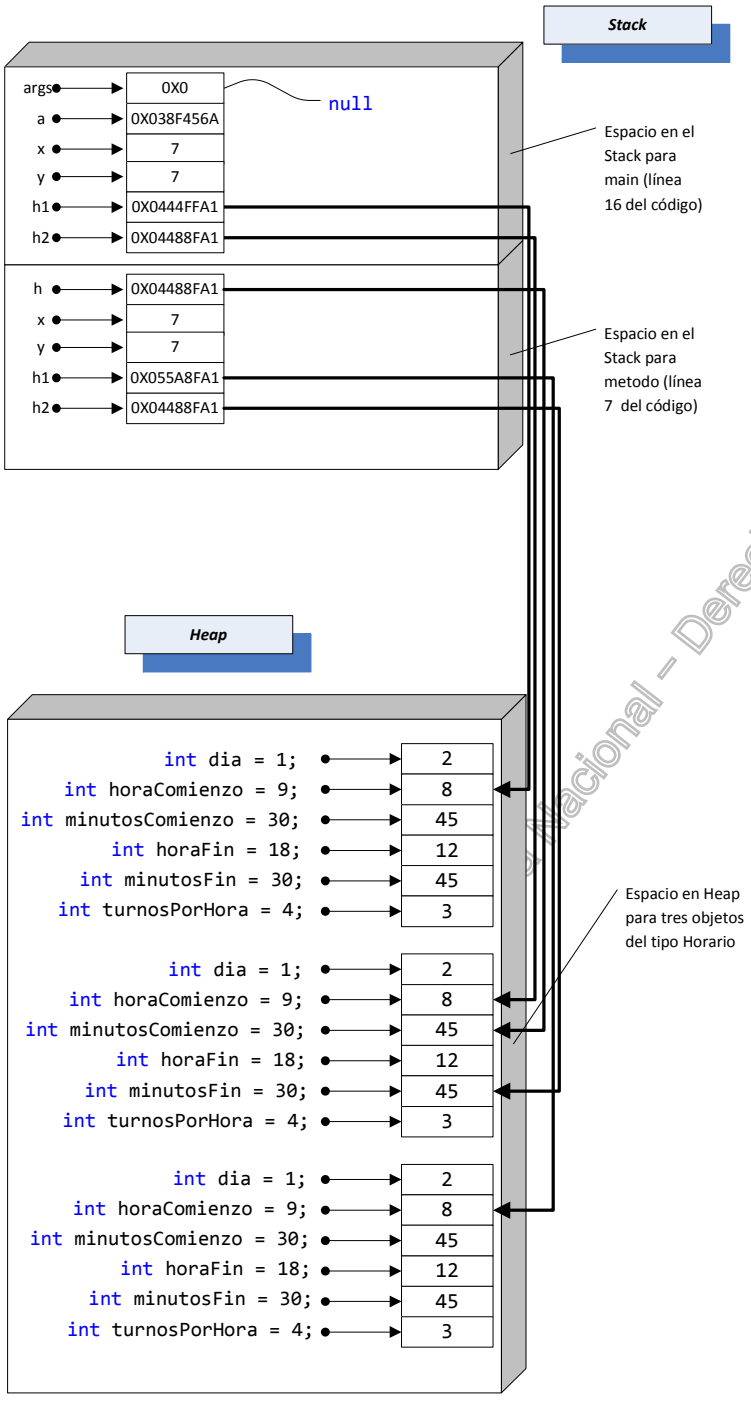
```
6.              Horario h2 = h;
```



Nuevamente hay dos variables de referencia apunta a un mismo objeto. La diferencia es que estas se encuentran en espacio del stack reservado para **método**, por lo tanto no tienen nada que ver con las que se encuentran en el espacio reservado para **main**.

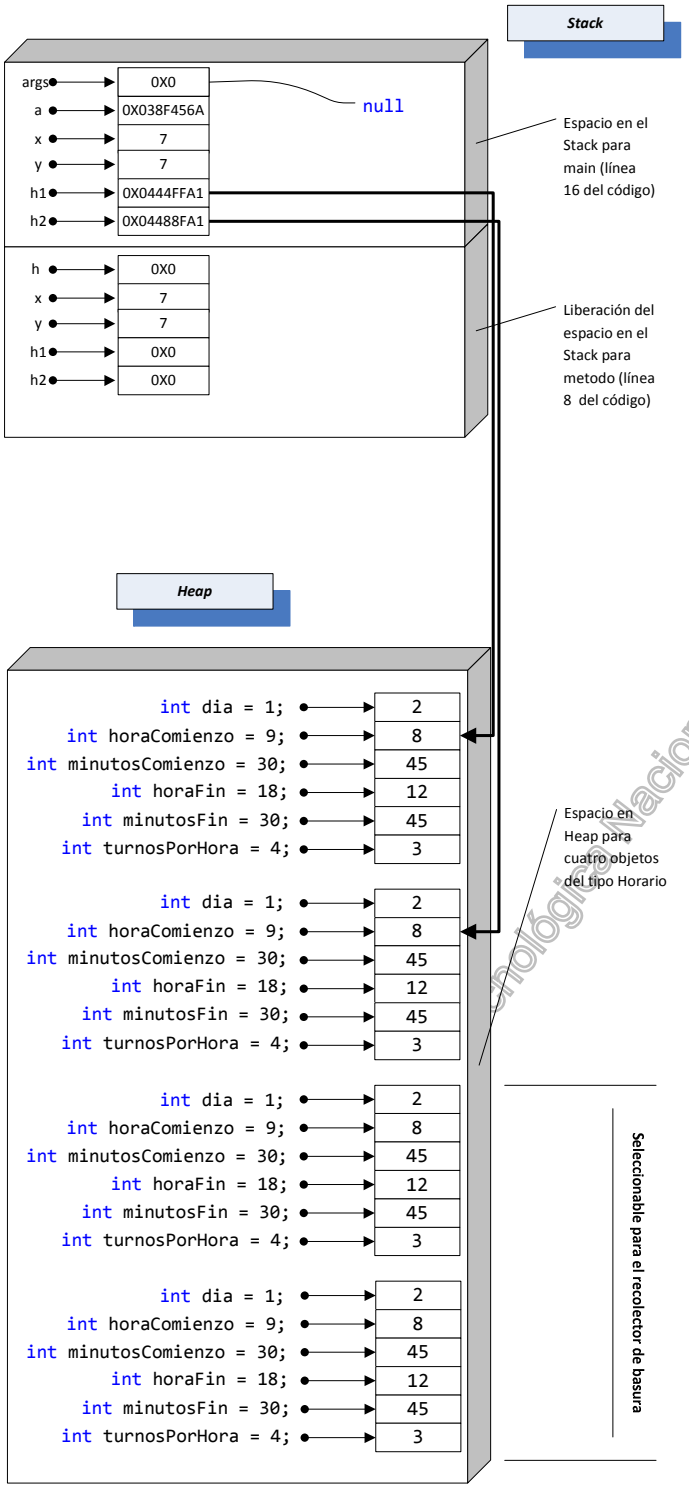
Recapitulando, la variable **h2** que se encuentra en el espacio del stack para **main** y el parámetro que recibe **metodo** apuntan a un mismo objeto en el heap. Las variables **h1** y **h2** que pertenecen a **metodo** apuntan también a un mismo objeto diferente al que apunta el parámetro y la variable de **main**, *pero todos son objetos del tipo Horario* (mismo tipo, objetos diferentes).

```
7.              h2 = new Horario(2, 8, 45, 12, 45, 3);
```



Una vez más se crea un nuevo objeto en el heap, esta vez para asignarlo a la variable de referencia **h2** de **metodo**. En este punto existen cuatro objetos en el heap y cinco referencias activas en el stack.

```
8.      }
```

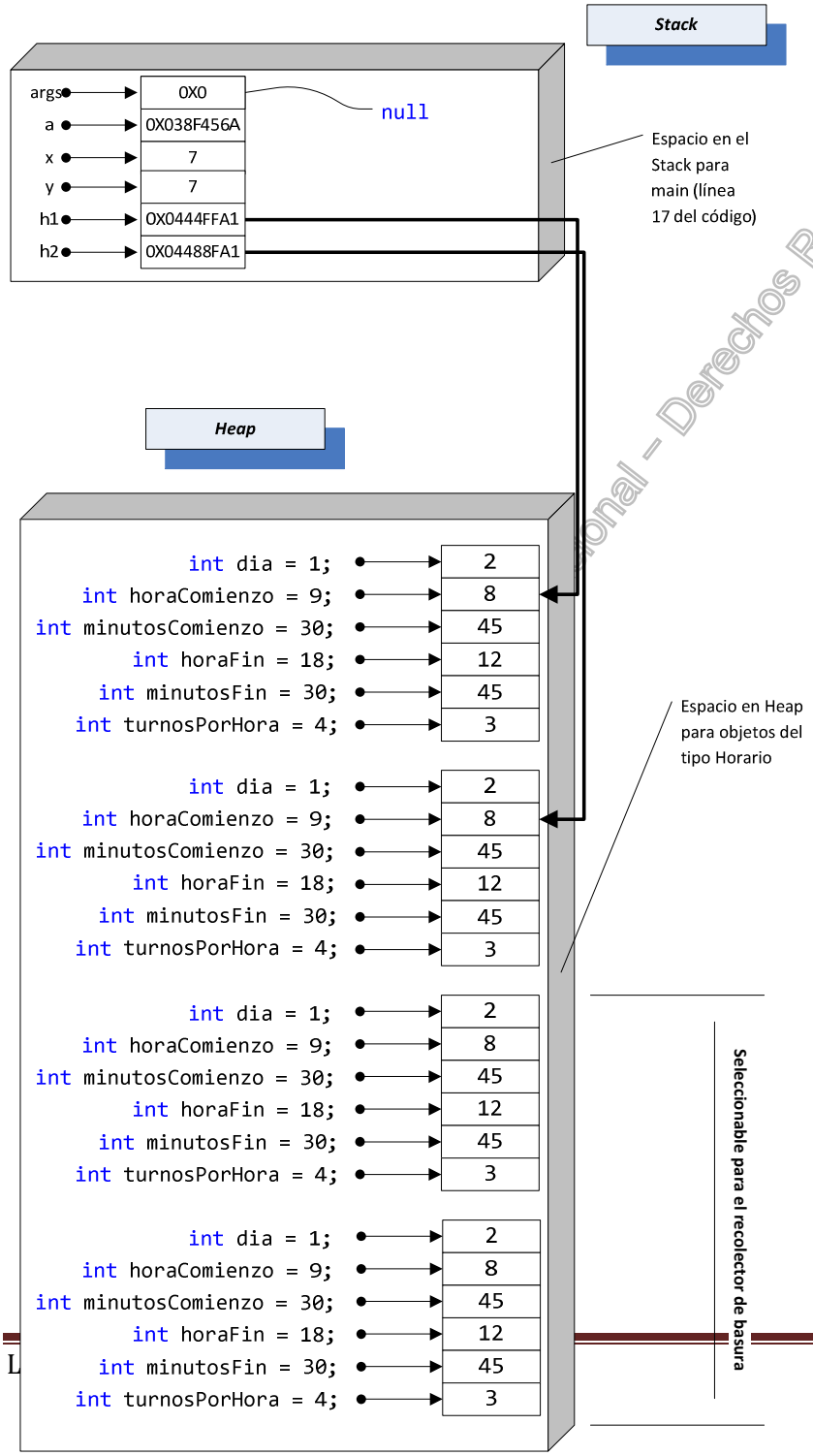


Esta línea indica el fin del método, lo cual acarrea como consecuencia la liberación de los recursos que estaban siendo utilizados por el método invocado. Las consecuencias son las siguientes:

- Toda referencia al heap se libera, lo cual implica que si un objeto que se encuentra en el heap **no tiene una referencia activa que apunte hacia él en algún lugar del código que se sigue ejecutando en el programa, el mismo es elegible para que lo libere el recolector de basura (garbage collector)**. Los objetos siguen estando en el heap, pero no son accesibles y el recolector liberará el espacio que ocupan en un momento en el cual el algoritmo que lo gestiona considere oportuno (la máquina virtual posee un criterio para liberar objetos a medida que transcurre el tiempo. Cuanto más tiempo estuvo en memoria sin referencia activa, más rápido lo elimina).
- Se libera el espacio asignado al método en el stack, con lo cual dicho espacio se puede asignar a cualquier nuevo método que lo requiera.
- El control del código

retorna a la siguiente instrucción posterior al lugar donde se realizó la invocación del método

```
17.       }
```



Justo antes de ejecutar esta línea que indica el fin de **main**, y por lo tanto del programa, la memoria queda en el estado que muestra el gráfico. Al ejecutar esta línea, se liberan todos los recursos que aún están tomados, tanto en el heap como el stack, y la memoria queda "limpia".

Pasajes de parámetros

Pasaje de parámetros por valor

*En .Net si los pasajes de parámetros **no** tienen un modificador que indique lo contrario en la declaración del mismo siempre se realizan por valor*. Esto quiere decir que cuando se invoca a un método, el parámetro se copia en el espacio del stack que corresponde a dicho método.

Sin embargo, se debe prestar especial atención cuando el valor pasado como parámetro es una referencia, ya que esta apunta al heap y si se modifican valores estos se reflejaran en el objeto al cual referencia la variable.

El siguiente ejemplo demuestra como varían los valores según se utilizan, se rescriben o se pasan a otros lugares de memoria.

Nota: en VB es opcional la declaración `ByVal` para los pasajes de parámetros por valor

Ejemplo

C#

```
namespace Argumentos
{
    class PasajeDeArgumentos
    {
        public void CambiarEntero(int valor)
        {
            valor = 55;
        }

        public void CambiarRefObjeto(Horario referencia)
        {
            referencia = new Horario(1, 7, 30, 11, 00, 4);
        }

        public void CambiarAtributoObjeto(Horario referencia)
        {
            referencia.agregarDias(4);
        }
        static void Main(string[] args)
        {
            PasajeDeArgumentos p = new PasajeDeArgumentos();
            Horario h;
            int val;
            // Asignarle un valor al entero
            val = 11;
            // Intento de cambiarlo
            p.CambiarEntero(val);
            // ¿Cuál es el valor actual?
            Console.WriteLine("El valor del entero es: " + val);
            Console.WriteLine("--------------------");
```

```
            // Asignar un objeto del tipo Horario
            h = new Horario(2, 8, 45, 12, 45, 3);
            // Intento por cambiarlo
            p.CambiarRefObjeto(h);
            // ¿Cuál es el valor actual?
            h.Imprimir();
            Console.WriteLine("--------------------");

            // Cambiando el atributo día
            // a través de la referencia al objeto
            p.CambiarAtributoObjeto(h);
            // ¿Cuál es el valor actual?
            h.Imprimir();
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Public Sub CambiarEntero(ByVal valor As Integer)
        valor = 55
    End Sub

    Public Sub CambiarRefObjeto(ByVal referencia As Horario)
        referencia = New Horario(1, 7, 30, 11, 0, 4)
    End Sub

    Public Sub CambiarAtributoObjeto(ByVal referencia As Horario)
        referencia.agregarDias(4)
    End Sub
    Sub Main()
        Dim h As Horario
        Dim val As Integer
        ' Asignarle un valor al entero
        val = 11
        ' Intento de cambiarlo
        CambiarEntero(val)
        ' ¿Cuál es el valor actual?
        Console.WriteLine("El valor del entero es: " + CStr(val))
        Console.WriteLine("--------------------")

        ' Asignar un objeto del tipo Horario
        h = New Horario(2, 8, 45, 12, 45, 3)
        ' Intento por cambiarlo
        CambiarRefObjeto(h)
        ' ¿Cuál es el valor actual?
        h.Imprimir()
        Console.WriteLine("--------------------")

        ' Cambiando el atributo día
        ' a través de la referencia al objeto
        CambiarAtributoObjeto(h)
        ' ¿Cuál es el valor actual?
        h.Imprimir()
```

```
        Console.ReadKey()
    End Sub

End Module
```

La salida producida es la siguiente:

```
El valor del entero es: 11
-------------------
Horario:
Día: 2
Hora de comienzo: 8
Minutos de comienzo: 45
Hora de fin: 12
Minutos de fin: 45
Turnos por hora:3
-------------------
Horario:
Día: 2
Hora de comienzo: 8
Minutos de comienzo: 45
Hora de fin: 12
Minutos de fin: 45
Turnos por hora:3
```

Pasaje por referencia

La declaración de parámetros puede tener modificadores que afectan al espacio de memoria utilizado para manejar el mismo. Cuando un parámetro se declara como referencia, indica que el lugar de almacenamiento utilizado en el stack es el del método que realiza la invocación. Esto quiere decir que se recibe la referencia al espacio de almacenamiento como argumento en la invocación un lugar de memoria que le pertenece a otro método y lo único que almacena el método actual es la referencia a dicho espacio de memoria. ***Cabe destacar que los pasajes por referencia se utilizan con tipos por valor***, ya que utilizar una referencia como argumento puede ocasionar resultados inesperados.

En C# se pueden utilizar dos modificadores diferentes, `ref` y `out`. El primero obliga a que el parámetro posea una referencia a un elemento no nulo al momento de la invocación (esto quiere decir que la referencia apunte a un valor en memoria obligatoriamente), mientras que el segundo puede ser nulo.

En VB sólo existe un modificador y es `ByRef`, el cual debe ser distinto de nulo (a menos que se lo declare como opcional, lo cuál se verá posteriormente).

Ejemplo

C#

```
namespace ArgumentosPorReferencia
{
    public class Referencias
```

```
    {
        private void PruebaRef(ref int uno, int dos)
        {
            // Esta asignación afectará al parámetro
            uno = uno + dos;
            // Esta no afectará al valor usado como segundo argumento
            dos = 999;
        }

        public  void ProbandoRef()
        {
            int uno = 5;
            int dos = 2;
            Console.WriteLine("uno= {0}, dos = {1}", uno, dos);
            PruebaRef(ref uno, dos);
            Console.WriteLine("uno= {0}, dos = {1}", uno, dos);
        }
    }
}
```

VB

```
Public Class Referencias
    Private Sub PruebaRef(ByRef uno As Integer, ByVal dos As Integer)

        ' Esta asignación afectará al parámetro
        uno = uno + dos
        ' Esta no afectará al valor usado como segundo argumento
        dos = 999
    End Sub

    Public Sub ProbandoRef()

        Dim uno As Integer = 5
        Dim dos As Integer = 2
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos)
        PruebaRef(uno, dos)
        Console.WriteLine("uno= {0}, dos = {1}", uno, dos)
    End Sub
End Class
```

Para utilizar la clase y comprobar el efecto de los pasajes por referencia de argumentos en los métodos, se utiliza el siguiente código.

Ejemplo

C#

```
namespace ArgumentosPorReferencia
{
    class Program
    {
        static void Main(string[] args)
        {
            Referencias r = new Referencias();
```

```
            r.ProbandoRef();
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Sub Main()
        Dim r As Referencias = New Referencias()
        r.ProbandoRef()
        Console.ReadKey()
    End Sub
End Module
```

Notar en la salida obtenida el cambio de valor en el parámetro uno del método `Main`

uno= 5, dos = 2
uno= 7, dos = 2

Determinación del mecanismo de pasaje de argumentos a utilizar

A la hora de elegir el mecanismo de pasaje de argumentos a utilizar (por valor o referencia) hay que tomar decisiones respecto del diseño del código. Se debe elegir minuciosamente el mecanismo para pasar argumentos en cada parámetro declarado.

- **Protección**. A la hora de elegir uno de los dos mecanismos que existen para pasar argumentos, el criterio más importante que hay que tener en cuenta es la exposición al cambio de las variables de llamada. La ventaja de pasar un argumento con ref ó out en C# o `ByRef` en VB, es que el método puede devolver un valor al código de llamada por medio del argumento. La ventaja de pasarlo por valor (usando o no `ByVal` en VB) es que protege a la variable de los cambios que sobre ella pueda efectuar el método invocado.
- **Rendimiento**. Aunque el mecanismo que se utilice para pasar argumentos también puede afectar al rendimiento del código, la diferencia suele ser insignificante. Existe una excepción: cuando se pasa un tipo por valor. En este caso, tanto C# como VB copian todo el contenido de los datos del argumento. Por lo tanto, para un tipo por valor grande, como una estructura, lo más eficiente es transferirlo por referencia. En los tipos por referencia, sólo se copia el puntero de los datos (cuatro bytes en plataformas de 32 bits y ocho bytes en plataformas de 64 bits).

¿Cuándo se debe pasar un argumento por valor?

- Si la parte del código que realiza la llamada al método usa como argumento un elemento no modificable, declarar el parámetro correspondiente `ByVal` (VB - opcional) o sin ningún modificador de tipo (C# y VB). Ningún código puede cambiar el valor de un elemento no modificable.
- Si la parte del código que realiza la llamada al método usa como argumento un elemento modificable, pero no se desea que el método pueda modificar el valor, declarar el

parámetro como `ByVal` (VB - opcional) o sin ningún modificador de tipo (C# y VB). Sólo el código de llamada puede cambiar el valor de un elemento modificable transferido por valor.

¿**Cuándo se debe pasar un argumento por referencia**?

- Si la parte del código que realiza la llamada al método necesita realmente modificar el valor en el código que realiza la llamada, declarar el parámetro correspondiente como `ref` ó `out` (C#), `ByRef` (VB).
- Si la ejecución correcta del código depende del método que cambia el elemento pasado como parámetro en el código que realiza la llamada, declarar el parámetro `ref` ó `out` (C#), `ByRef` (VB). Si se lo transfiere por valor o si el código de llamada en lugar de utilizar el mecanismo para pasar argumentos `ref` ó `out` (C#), `ByRef` (VB) incluye el argumento sólo entre paréntesis, la llamada al procedimiento podría producir resultados inesperados.

Los argumentos opcionales y nombrados

Los argumentos con nombre permiten especificarlos para un parámetro en particular mediante la asociación del argumento con el nombre del parámetro en lugar de la posición del mismo en la lista de parámetros. Los *argumentos opcionales* permiten omitirlos en algunos parámetros. Ambas técnicas se pueden utilizar con los métodos, indizadores, constructores y delegados.

Al utilizar argumentos opcionales y nombrados, estos se evalúan en el orden en el que aparecen en la lista de argumentos, no a la lista de parámetros (o sea, en la invocación).

Cuando se usan juntos ambos tipos de argumentos, permiten proporcionarlos para sólo algunos parámetros de una lista de parámetros opcionales.

Argumentos nombrados: al usarlos no es necesario recordar o buscar el orden de los parámetros en las listas de parámetros de los métodos llamados. El argumento para cada parámetro puede ser especificado por medio del nombre de parámetro. Por ejemplo, una función que calcula el índice de masa corporal (IMC) puede ser llamado en la forma estándar mediante el envío de argumentos para el peso y la altura, a través de la posición en el orden definido en el método.

Ejemplo

```
CalcularIMC(123, 64)
```

Si no se recuerda el orden de los parámetros, pero se conocen sus nombres, se pueden enviar los argumentos en cualquier orden, primero el peso o la altura.

Ejemplo

C#

```
CalcularIMC(peso: 123, altura: 64)
CalcularIMC(altura: 64, peso: 123)
```

VB

```
CalcularIMC(peso:=123, altura:=64)
CalcularIMC(altura:=64, peso:=123)
```

Los argumentos con nombre también mejoran la legibilidad del código mediante identificación de lo que representa cada argumento.

Un argumento con nombre se puede colocar a continuación de los argumentos con posición, como se muestra en el siguiente código.

Ejemplo

C#

```
CalcularIMC(123, altura: 64)
```

VB

```
CalcularIMC(123, altura:=64)
```

Sin embargo, un argumento posicional no puede colocarse a continuación de un argumento con nombre. La siguiente declaración se produce un error del compilador.

Ejemplo

C#

```
// Los argumentos posicionales no pueden seguir los argumentos con nombre.
// La siguiente sentencia provoca un error del compilador.
// Console.WriteLine(CalcularIMC(peso: 123, 64));
```

VB

```
' Los argumentos posicionales no pueden seguir los argumentos con nombre.
' La siguiente sentencia provoca un error del compilador.
' Console.WriteLine(CalcularIMC(peso:= 123, 64))
```

Argumentos opcionales: La definición de un método, constructor, indizador o delegado puede especificar que sus parámetros son necesarios u opcionales. Cualquier llamada debe proporcionar argumentos para todos los parámetros requeridos, pero puede omitir los argumentos para los parámetros opcionales.

Cada parámetro opcional tiene un valor por defecto como parte de su definición. Si no hay ningún argumento enviado a ese parámetro, se utiliza el valor predeterminado. Este debe ser uno de los siguientes tipos de expresiones:

- Una expresión constante.
- Una expresión de la forma VALTYPE `new` () , donde VALTYPE es un tipo por valor, como una enumeración o una estructura.
- Una expresión de la forma default(VALTYPE) , donde VALTYPE es un tipo por valor.

Los parámetros opcionales se definen al final de la lista de parámetros, después de los parámetros requeridos. Si el código que realiza la invocación proporciona un argumento para cualquier parámetro opcional dentro de una serie de estos, debe proporcionar argumentos para todos los parámetros opcionales que le preceden. No son permitidos los lugares vacíos separados por comas en la lista de argumentos. Por ejemplo, en el siguiente código, el método de instancia `MetodoDeEjemplo` se define con un parámetro obligatorio y dos opcionales.

Ejemplo

C#

```
public void MetodoDeEjemplo(int requerido,
      string cadenaOpcional = "cadena por defecto",
      int enteroOpcional = 10)
  {
      Console.WriteLine("{0}: {1}, {2}, and {3}.",
        _nombre, requerido, cadenaOpcional, enteroOpcional);
  }
```

VB

```
Public Sub MetodoDeEjemplo(ByVal requerido As Integer,
  Optional ByVal cadenaOpcional As String = "cadena por defecto",
   Optional ByVal enteroOpcional As Integer = 10)
    Console.WriteLine("{0}: {1}, {2}, and {3}.", _nombre, requerido,
          cadenaOpcional, enteroOpcional)
End Sub
```

La siguiente llamada a `MetodoDeEjemplo` provoca un error de compilación, porque un argumento se proporciona para el tercer parámetro, pero no para el segundo.

Ejemplo

```
// No se puede dejar un vacío en los argumentos proporcionados.
//e1.MetodoDeEjemplo(3, ,4);
```

Sin embargo, si se conoce el nombre del tercer parámetro, se puede utilizar un argumento con nombre para llevar a cabo la tarea.

Ejemplo

C#

```
// Se puede utilizar un parámetro nombrado para hacer
// funcionar la sentencia anterior
e1.MetodoDeEjemplo(3, enteroOpcional: 4);
```

VB

```
' Se puede utilizar un parámetro nombrado para hacer
' funcionar la sentencia anterior
e1.MetodoDeEjemplo(3, enteroOpcional:=4)
```

Cantidad variable de argumentos

Por lo general, no se puede llamar a un método con más argumentos que la declaración del mismo especifica como lista de parámetros. Cuando se necesita un número indefinido de argumentos, se puede declarar un vector de parámetros, lo que permite que un método acepte un vector de valores como argumento. No se tiene que saber el número de elementos en él cuando se lo define en el método. Su tamaño se determina individualmente por cada llamada al método.

Se utiliza la palabra clave `params` ó `ParamArray` (C# ó VB) para referirse a un vector de parámetros. Se aplican las siguientes reglas:

- Un método sólo puede tener un vector de parámetros, y debe ser el último argumento en la definición del procedimiento.
- Un vector de parámetros debe pasarse por valor. Es buena práctica de programación incluir explícitamente la palabra clave `ByVal` en VB en la definición del método.
- El código dentro del procedimiento debe tratar al vector de parámetros como de una dimensión, cada elemento de los cuales es el mismo tipo de datos con el que se declaró `params` ó `ParamArray`.
- La matriz de parámetros de forma automática opcional. Su valor por defecto es un vector vacío unidimensional del tipo de elementos con el que se definió el vector de parámetros.
- Todos los argumentos que preceden a la matriz de parámetros deben ser parámetros requeridos. El vector de parámetros debe ser el único argumento opcional.

Cuando se llama a un procedimiento con un argumento de vector de parámetros, puede pasar por cualquiera de los siguientes casos para el vector de parámetros:

- `Nothing` o `null` - es decir, se puede omitir el argumento `params` ó `ParamArray`. En este caso, se pasa al método un vector vacío. También se puede pasar la palabra clave `Nothing` o `null`, con el mismo efecto.
- Una lista de una cantidad indefinida de argumentos, separados por comas. El tipo de datos de cada argumento debe ser implícitamente convertible al tipo del elemento declarado para el `params` ó `ParamArray`.
- Un vector con el mismo tipo de elementos que posee el vector de parámetros.

El siguiente ejemplo muestra cómo se puede definir un método con un vector de parámetros.

Ejemplo

C#

```
namespace ArgumentosVariables
{
    class CantidadVariablesDeArgumentos
    {
        public void ParametrosVariables(params int[] vec)
        {
            for (int i = 0; i < vec.Length; i++)
```

```
            {
                Console.WriteLine(vec[i]);
            }
            Console.WriteLine();
        }

        public void ParametrosVariables2(params object[] vec)
        {
            for (int i = 0; i < vec.Length; i++)
            {
                Console.WriteLine(vec[i]);
            }
            Console.WriteLine();
        }

        static void Main(string[] args)
        {
            CantidadVariablesDeArgumentos v = new CantidadVariablesDeArgumentos();
            v.ParametrosVariables(1, 2, 3);
            v.ParametrosVariables2(1, 'a', "test");

            // También se pueden pasar un conjunto de objetos ,
            // siempre y cuando el tipo del vector coincida con
            // el del método que se llama.
            int[] vector = new int[3] { 10, 11, 12 };
            v.ParametrosVariables(vector);
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Public Sub ParametrosVariables(ByVal ParamArray vec() As Integer)
        For i As Integer = 0 To vec.Length - 1
            Console.WriteLine(vec(i))
        Next i
        Console.WriteLine()
    End Sub

    Public Sub ParametrosVariables2(ByVal ParamArray vec() As Object)
        For i As Integer = 0 To vec.Length - 1
            Console.WriteLine(vec(i))
        Next i
        Console.WriteLine()
    End Sub

    Sub Main()
        ParametrosVariables(1, 2, 3)
        ParametrosVariables2(1, "a", "test")

        ' También se pueden pasar un conjunto de objetos ,
        ' siempre y cuando el tipo del vector coincida con
        ' el del método que se llama.
        Dim vector() As Integer = New Integer() {10, 11, 12}
```

```
        ParametrosVariables(vector)
        Console.ReadKey()
    End Sub
End Module
```

Inferencia de tipos en las variables locales

La inferencia de tipos se refiere al momento en el cual el compilador tiene la responsabilidad de determinar el tipo de una variable porque este no fue indicado en el código.

Se utiliza de manera diferente dependiendo de las capacidades del lenguaje, por lo tanto se va a explicar su funcionamiento en base a cada uno de ellos.

C#

Las variables locales se pueden declarar como un "tipo" inferido mediante la palabra clave `var` en lugar de un tipo explícito. Esta palabra clave indica al compilador que infiera el tipo de la variable de la expresión en el lado derecho de la instrucción de inicialización. El tipo inferido puede ser un tipo incorporado, un tipo anónimo, un tipo definido por el usuario, o un tipo definido en el Framework de .NET.

Los siguientes ejemplos muestran distintas maneras en que las variables locales pueden ser declaradas con `var`.

Ejemplo

C#

```
        static void Main(string[] args)
        {
            // i se compila como un int
            var i = 5;

            // s se compila como un string
            var s = "Hola";

            // v se compila como un int[]
            var v = new[] { 0, 1, 2};

            // Esta línea es un error de compilación
            // var v = new[] { 0, 1, 2, "Hola" };

            // obj se compila como un objeto del tipo Horario
            var obj = new Horario(1, 7, 30, 11, 00, 4);

            Console.WriteLine(i);
            Console.WriteLine(s);
            Console.WriteLine(v[1]);
            Console.WriteLine(obj.HoraComienzo);
            Console.ReadKey();
        }
    }
```

Es importante entender que la palabra clave `var` no significa "variant" y no indica que la variable no tiene un tipo definido o que el tipo será determinado en tiempo de ejecución. Sólo significa que el compilador determina y asigna el tipo más adecuado. Si el compilador no puede determinar un tipo apropiado lo indicará como un error en tiempo de compilación

Ejemplo

C#

```
            // Esta línea es un error de compilación
            // var v = new[] { 0, 1, 2, "Hola" };
```

VB

El compilador de Visual Basic utiliza la inferencia de tipos para determinar los tipos de datos de las variables locales declaradas sin una cláusula `As`. El compilador deduce el tipo de la variable mediante el tipo de la expresión de inicialización. Esto permite declarar variables sin establecer explícitamente un tipo, tal como se muestra en el siguiente ejemplo.

Ejemplo

VB

```
Module Module1
    Sub Main()
        ' i se compila como un int
        Dim i = 5

        ' s se compila como un string
        Dim s = "Hola"

        ' v se compila como un int[]
        Dim v = New Integer() {0, 1, 2}

        ' Esta línea es un error de compilación
        ' var v = New() { 0, 1, 2, "Hola" }

        ' obj se compila como un objeto del tipo Horario
        Dim obj = New Horario(1, 7, 30, 11, 0, 4)

        Console.WriteLine(i)
        Console.WriteLine(s)
        Console.WriteLine(v(1))
        Console.WriteLine(obj.HoraComienzo)
        Console.ReadKey()
    End Sub
End Module
```

La inferencia de tipo local se aplica a nivel de procedimiento. No se puede utilizar para declarar variables a nivel del módulo (dentro de una clase, estructura, módulo o interfaz, pero si dentro de un procedimiento o bloque). Si una variable es un campo de una clase en lugar de una variable local en un procedimiento, la declaración causa un error con `Option Strict On`, y si no está activado, interpreta a la variable que es un atributo como un objeto. Del mismo modo, la

inferencia de tipo de variable local no se aplica a las variables de procedimiento (variables locales) si el mismo se declara como estático (Shared).

Si el compilador no puede determinar un tipo apropiado lo indicará como un error en tiempo de compilación.

Ejemplo

VB

```
' Esta línea es un error de compilación
' var v = New() { 0, 1, 2, "Hola" }
```

Palabras clave

Las palabras clave en son las que se muestran a continuación:

C#

case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

VB

AddHandler	AddressOf	Alias	And
AndAlso	Ansi	As	Assembly
Auto	Boolean	ByRef	Byte
ByVal	Call	Case	Catch
CBool	CByte	CChar	CDate
CDec	CDbl	Char	CInt
Class	CLng	CObj	Const
CShort	CSng	CStr	CType
Date	Decimal	Declare	Default
Delegate	Dim	DirectCast	Do
Double	Each	Else	ElseIf
End	Enum	Erase	Error
Event	Exit	False	Finally
For	Friend	Function	Get
GetType	GoSub	GoTo	Handles
If	Implements	Imports	In
Inherits	Integer	Interface	Is
Let	Lib	Like	Long
Loop	Me	Mod	Module
MustInherit	MustOverride	MyBase	MyClass
Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object
On	Option	Optional	Or
OrElse	Overloads	Overridable	Overrides
ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	RemoveHandler	Resume
Return	Select	Set	Shadows
Shared	Short	Single	Static
Step	Stop	String	Structure
Sub	SyncLock	Then	Throw
To	True	Try	TypeOf
Unicode	Until	Variant	When
While	With	WithEvents	WriteOnly
Xor	#Const	#ExternalSource	#If...Then...#Else
#Region	-	&	&=
*	*=	/	/=
\	\=	^	^=

+	+=	=	-=