Ingeniería en Desarrollo de Software

6º semestre

Programa de la unidad didáctica:

Programación .NET II

Unidad 1. Desarrollo de objetos en la plataforma .NET

Clave:

150930934

Universidad Abierta y a Distancia de México UnADM

México, Ciudad de México, febrero de 2025

DCEIT

Índice

Unidad 1. Desarrollo de objetos en la plataforma .NET

Presentación de la unidad

Bienvenido(a) a la primera etapa de la unidad didáctica Programación .Net II. Es una materia seriada, y en este sentido será necesario que apliques los conocimientos adquiridos en Programación .Net I del quinto semestre, bloque 1. Ahora sabes que en la programación .Net uno de los lenguajes utilizados es CSharp, y se aplica en la solución de problemas orientados a objetos.

Con esta unidad se inicia tu aprendizaje de los conceptos de programación orientada a objetos utilizando la tecnología de .Net, específicamente el lenguaje CSharp. Para ello, recordarás conceptos como *propiedades*, *métodos miembros*, *constructores*, *destructores* y *sobrecarga*, todos ellos desde el punto de vista de Microsoft y .Net.

Aprenderás nuevos conceptos tales como los que se refieren a colecciones, arreglos y manejo de errores. Elementos que son empleados en diversos lenguajes orientados a objetos, y de gran utilidad en el desarrollo de aplicaciones en el sector productivo.

Logros

Al término de esta unidad lograrás:

- Entender el concepto de *encapsulación*. Comprender la utilidad que para este fin tienen los métodos junto con los datos miembros de una clase, y su forma de construir clases encapsuladas.
- Identificar en qué consiste la encapsulación, la utilidad de los métodos y datos miembro de una clase.
- Manejar los constructores y destructores dentro de una clase, así como los diferentes tiempos en los que estos métodos son utilizados.
- Comprender y utilizar la sobrecarga de un objeto, aplicarla en el desarrollo de éste, y a la vez entender los beneficios de la sobrecarga de métodos, constructores y operadores.

Unidad 1. Desarrollo de objetos en la plataforma .NET

Competencia específica

- Declara clases y aplicar la sobrecarga de métodos y operadores, garantizando la integridad de los datos con el encapsulamiento y los métodos especiales constructor y destructor, mediante la codificación de programas orientados a objetos en el lenguaje de programación C# de la plataforma .NET.

1.1. Encapsulación

La historia de la programación ha sido marcada por una serie de propuestas que están pensadas en guiar al desarrollador para que éste desempeñe su trabajo siguiendo dos principios:

Primero: reutilizar el trabajo ya realizado. En otras áreas de la ingeniería no se parte de cero para la construcción del siguiente paso de la tecnología, normalmente se toma como punto de inicio lo ya creado. En el área de sistemas este concepto anteriormente era casi desconocido, la programación orientada a objetos intenta agregar este proceso a la forma de desarrollar objetos de software.

Segundo: el mundo real puede ser interpretado, de manera más fácil, si todo se representa como objetos y con la manera en que éstos se congregan para formar entidades más complejas. La programación orientada a objetos sigue este pensamiento, mientras busca que el desarrollo sea un proceso acumulativo de objetos computacionales basados en los anteriores.

La programación orientada a objetos se basa en conceptos considerados fundamentales de esta metodología de programación, tales como *encapsulación*, *herencia* y *polimorfismo* (Sharp, 2010). Para iniciar el estudio de la unidad se presenta en este tema la definición y características del concepto de encapsulación.

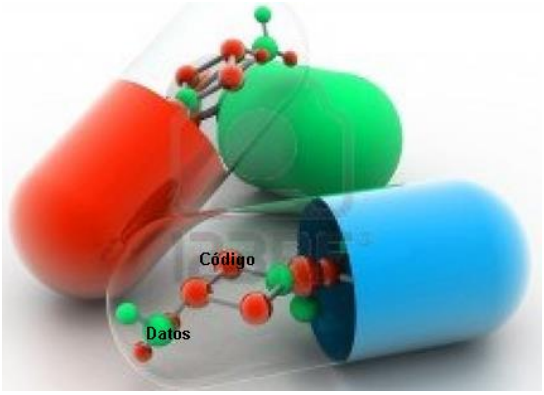
La **encapsulación** es un mecanismo que consiste en organizar datos y métodos de una estructura para evitar que ellos sean consultados y modificados por un medio distinto a

los especificados. Por lo tanto, la encapsulación permite garantizar la integridad de los datos que contiene un objeto (Ferguson *et al*., 2010). Recuerda que la **integridad de los datos** es un concepto fundamental dentro de la programación orientada a objetos, por lo que debes asegurarte de que una clase sea la única responsable del acceso a sus métodos y atributos. Esto está definido así en la programación orientada a objetos.

Además de envolver datos y métodos en una sola unidad, la encapsulación también oculta los detalles internos de datos y el comportamiento de un objeto (Michaells, 2010).

El usuario de una clase en particular no necesita saber cómo están estructurados los datos dentro de ese objeto, es decir, un usuario no necesita saber la forma en que fueron programados. Mediante la encapsulación los datos se ocultan, "se encapsulan" dentro de una clase, y la clase implementa un diseño que permite que otras partes del código accedan de esos datos de forma eficiente. Imagina la encapsulación como un envoltorio protector que rodea a los datos de las clases de C# (Ferguson *et al*., 2010).

De acuerdo con lo anterior, es posible comparar la clase con una **cápsula**, puesto que cada objeto es una estructura en cuyo interior hay **datos** y un **código** que los manipula, todos ellos relacionados entre sí, como si estuvieran encerrados en una cápsula, de ahí el término de **encapsulación**.



Moléculas dentro de la cápsula.
http://us.123rf.com/400wm/400/400/cooldesign/cooldesign1107/cooldesign110700072/9974518-moleculas-dentro-de-la-capsula.jpg

Encapsulación significa que un grupo de propiedades relacionadas, métodos y otros miembros, se tratan como una sola **unidad** u **objeto** (MSDN, 2013 e). Se conocen 4 alcances (MSDN, 2013 e):

1. **Público**: en el que cualquier función de cualquier objeto puede acceder a los datos o métodos de una clase que se define con este nivel de acceso. Este es el nivel de protección de datos más bajo.
2. **Internal**: el acceso a los datos sólo está permitido al proyecto actual, es decir, a la clase que se está compilando.
3. **Protegido**: es un acceso a los datos restringido, se hereda pero no se puede manipular desde afuera.
4. **Privado**: el acceso a los datos está restringido, sólo pueden ser modificados por los métodos de la clase que se está compilando. Este es el alcance más alto de protección de datos.

El siguiente ejemplo te aclarará el tema de los diferentes alcances.

Imagina que te solicitan crear la clase “Entrada” que representa en la vida real la entrada a una casa, a un edificio, etcétera. Todas las entradas deben tener las siguientes características:

alto, ancho y una propiedad que indique si está abierta o cerrada.

Estos datos (alto y ancho) pueden ser representados por una variable de tipo **double**, y la propiedad que indique si está cerrada o abierta de tipo **boolean**, **true** si está abierta y **false** si está cerrada.

Estas propiedades deberán ser de los siguientes tipos según el escenario que se presente:

Internal	Los datos miembro sólo serán manipulados por el mismo objeto, y éste sólo será manipulado por la misma clase que lo define. El objeto no tendrá una jerarquía de herencia.
	
	En el ejemplo los datos miembro tendrán la siguiente descripción:
	
	`double alto;`
	`double largo;`
	`bool entrada;`

Privado	Los datos miembro sólo serán manipulados por el mismo objeto, y éste será manipulado por otras clases pero no deben tener acceso a los datos miembro. El objeto no tendrá una jerarquía de herencia.
	
	En el ejemplo los datos miembro tendrán la siguiente descripción:
	
	`private double alto;`
	`private double largo;`
	`private bool entrada;`

Protegido	Los datos miembro sólo serán manipulados por el mismo objeto, y éste será manipulado por otras clases pero no deben tener acceso a los datos miembro. El objeto tendrá una jerarquía de herencia. Por ejemplo:
	
	`protected double alto;`
	`protected double largo;`
	`protected bool entrada;`

Público	Los datos miembro serán manipulados por el mismo objeto, y otras clases deben tener acceso a dichos datos. El objeto puede o no tener una jerarquía de herencia (puede o no tener hijos). Por ejemplo:
	
	`public double alto;`
	`public double largo;`
	`public bool entrada;`

En los siguientes subtemas abordarás los **métodos miembro,** que son una de las funciones que permiten dar integridad a los datos miembro. Cabe mencionar que los métodos miembro aseguran el proceso de encapsulación al ser las únicas funciones válidas que pueden manipular los datos.

1.1.1. Métodos miembro

Los métodos miembro son funciones que pertenecen a un objeto (son miembros de la clase, de ahí su nombre). En este subtema se explicará de una manera más profunda cómo se definen y sus propiedades.

Un método miembro es una función asociada a un objeto. Desde el punto de vista del comportamiento del objeto, es lo que el objeto puede hacer. Por ejemplo, si se tiene una clase llamada **CuentaBancariaDeDebito** y uno de sus datos es **saldo,** el comportamiento de esa clase es el de revisar que su saldo no sobrepase a un valor negativo; es decir, evitar, informar o estar pendiente de que no se retire más de lo que se tenga de saldo.

Como dice Marck Michaells en el libro *Essential C#* (2010) “un método es un medio de agrupación de una secuencia de instrucciones que realizan una determinada acción o calculan un resultado determinado, esto proporciona mayor estructura y organización de las normas que conforman el programa” (p. 220).

Todas las clases tendrán cuando menos un método, como ya pudiste constatarlo en la unidad didáctica *Programación .Net I*. En esta plataforma se cuenta con el método llamado **Main**(), que es el punto de entrada de la clase. Pero una clase también se compone de muchos otros métodos que, según lo propuesto por Deitel (2007), pueden clasificarse en:

- Constructores
- Destructores
- *Setters* y *getters*, también llamados propiedades
- Métodos de uso general

A continuación, se explica la forma en que se aplican los **métodos de uso general**.

Cuando un método es invocado (ejecutado), es conceptualmente lo mismo que enviar un mensaje a una clase, por ello algunos autores les llaman **interfaces**; porque son el medio de comunicar al objeto con su ambiente exterior.

Existen dos **funcionalidades básicas de un método**:

1. Manipular los datos miembro del mismo objeto.
2. Comunicarse con otros objetos, que son los llamados interfaces, porque tienen la capacidad de enviar y recibir mensajes de otros objetos.

Un método se define de acuerdo con la siguiente nomenclatura:

```
[alcance] valorDeRetorno nombreDelMétodo ([ Tipo1 Parámetro1, Tipo2 Parámetro2, ...])
//Declaración o firma del método
{
    Definición de variables locales;   //Sólo accesibles desde el método
    Conjunto de sentencias;

    [return [(] expresión [)]];  //No es necesario que sea la última
                                 //Puede aparecer más de una vez en el método pero
                                 //en todo caso indica que en ese punto termina el
                                 //método
}
```

Basado en MSDN, 2013a.

En la que se puede apreciar que:

[alcance]: se refiere al grado de encapsulación, que como sabes puede ser público, internal, protegido y privado. El estar entre corchetes [ ] significa que lo que está adentro no es necesario, es opcional, y, si no se pusieran, será de tipo **internal**.

Tipo: tipo del valor de retorno. Un método puede devolver un valor a quien lo llama o no devolver nada. El valor devuelto por un método puede ser de un tipo primitivo de datos o una referencia, pero nunca puede devolver más de un valor. Si el método no devuelve nada, el tipo devuelto por el método es el tipo **void** y debe indicarse utilizando esa palabra.

nombreDelMetodo: existe un acuerdo propuesto por Microsoft de que los nombres de los métodos comienzan con minúscula. Si el nombre de ese método es un nombre compuesto cada nueva palabra empieza con mayúsculas, para los nombres de los métodos también se ha propuesto que inicien con un verbo.

Ejemplo: sumarNumeros, calcularPerimetro, pintar, terminar.

Posible lista de parámetros: la lista de parámetros es opcional porque la función podría no tenerlos, en caso de que los tenga se trata de variables locales (sólo accesibles y visibles por el propio método), se separa por comas en las que se debe de especificar el tipo y nombre de cada uno, se inicializan en la llamada recibiendo los valores especificados por los argumentos de la llamada. Si la función no lleva parámetros hay que colocar los paréntesis (Peláez, 2005).

Lo contenido entre las llaves { } se conoce como el **cuerpo del método**, y está conformado por la siguiente información:

Definición de variables locales: dentro de los métodos se pueden definir variables, estas sólo son accesibles dentro del método en donde se han definido. Este tipo de variables no se inicializan por defecto, si no se inicializan en el momento de la definición, se deben de inicializar con un valor antes de utilizarlas, de lo contrario el compilador detectará un error (Peláez, 2005).

Sentencias: son las instrucciones necesarias para realizar una tarea.

La instrucción **return** devuelve el control de la ejecución al paso que hizo la llamada. Si se tiene un método no devuelve nada y se elimina la sentencia **return,** la función termina con la llave final, también llamada **llave de cierre de la función.** Es en estos casos cuando el método devuelve un tipo **void**. En los casos en los que el método retorna un valor, la función no puede ser de este tipo y la sentencia **return,** además de producir la salida de la función, especificará el valor de retorno al punto donde se hizo la llamada. En un método de una función puede haber más de una sentencia **return**, pero sólo se ejecuta una a un tiempo (Peláez, 2005).

A continuación, se presentan estos conceptos en un ejemplo:

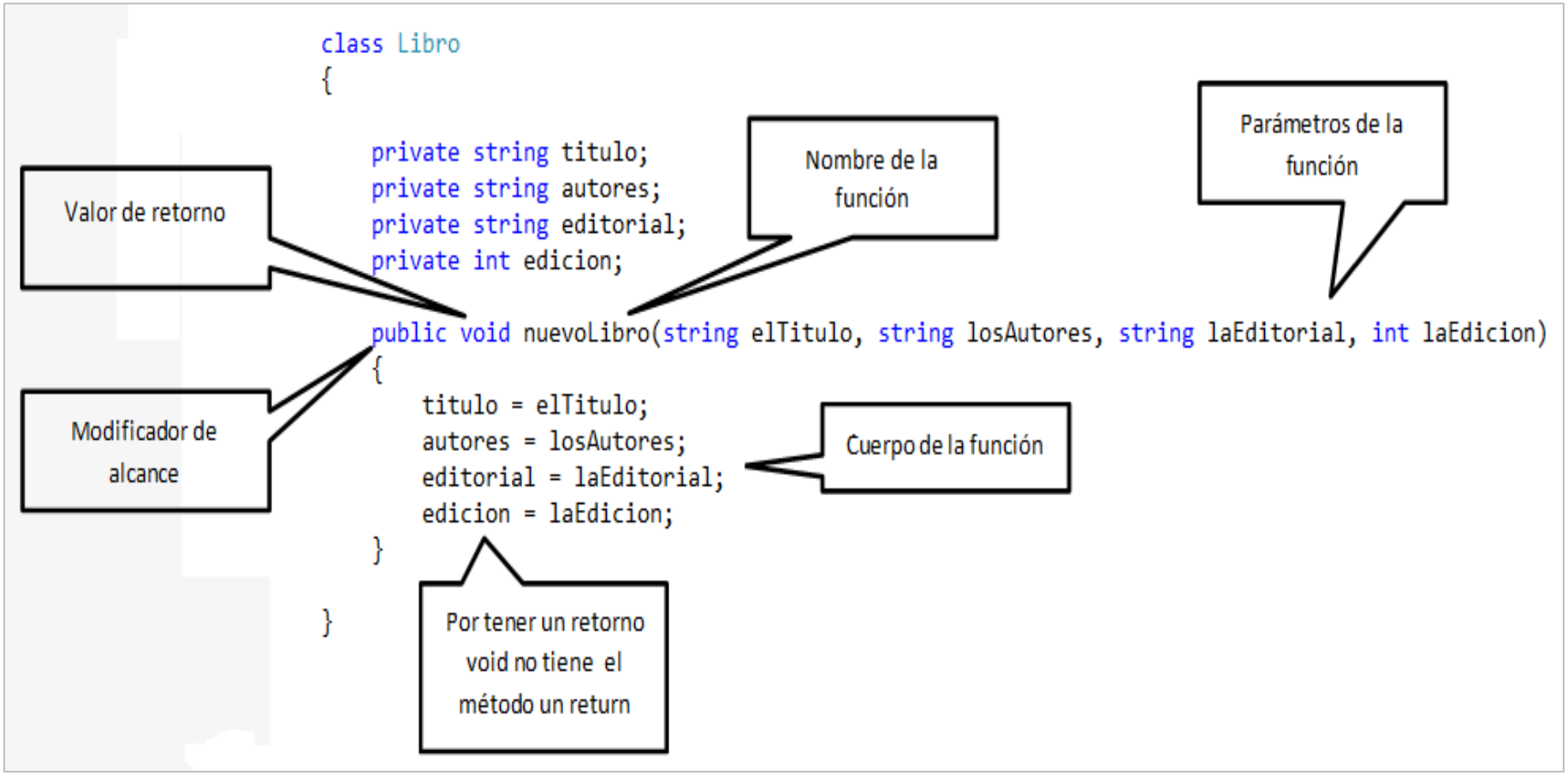
En el mundo real existen los libros. De entre sus muchas características un libro tiene título, autores, editorial y año de edición. Si se quisiera llevar este concepto de libro a un objeto en CSharp, se tendría una clase llamada **Libro**, con los datos miembro: **título**, **autores**, **editorial** y **edición**. Los tres primeros datos de **tipo string** y el último de **tipo entero**. El listado de esta idea se muestra a continuación:

```
class Libro
{

    private string titulo;
    private string autores;
    private string editorial;
    private int edicion;

}
```

Si se requiere agregar un método llamado **nuevoLibro** que reciba como parámetros **elTitulo, losAutores**, **laEditorial** y **laEdicion** de tipo **string** en los tres primeros casos y la última de tipo entero, estos parámetros se pasan al objeto como se observa a continuación:


class Libro
{
private string titulo;
private string autores;
private string editorial;
private int edicion;
public void nuevoLibro(string elTitulo, string losAutores, string laEditorial, int laEdicion)
{
titulo = elTitulo;
autores = losAutores;
editorial = laEditorial;
edicion = laEdicion;
}
}
Valor de retorno
Nombre de la función
Parámetros de la función
Modificador de alcance
Cuerpo de la función
Por tener un retorno void no tiene el método un return

Supón que se necesita un método llamado **antiguedadDelLibro** que recibe el año actual como un número entero y devuelve el número de años que tiene de impreso el libro. La función relacionada se muestra a continuación:

```
public int antiguedadDelLibro(int añoActual)
{
    int antiguedad = añoActual - edicion;
    return antiguedad;
}
```

A fin de no generar nuevas variables, CSharp permite repetir el nombre de los parámetros y el nombre de los datos miembro, la forma de diferenciarlos es utilizando la palabra reservada **this** para los datos miembro. Observa esto con un ejemplo:

Supón que se necesita una función llamada **cambiarTitulo** que permita precisamente cambiar el título del libro. Una forma de hacer esto se muestra a continuación:

```
public void cambiarTitulo(string titulo)
{
    this.titulo = titulo;
}
```

this.titulo = Dato miembro de la clase

titulo = Parámetro del método

En el ejemplo se está **guardando el parámetro dentro del objeto**.

Finalmente recuerda que si el método es público podrá ser utilizado por cualquier programa. Observa el uso de este objeto:

```
static void Main(string[] args)
{

    Libro libro = new Libro();
    libro.nuevoLibro("C# para estudiantes","Douglas Bell","Ed. Person",2010);
    int antiguedad = libro.antiguedadDelLibro(2013);
    System.Console.WriteLine("El libro tiene {0} años de antigüedad.",antiguedad);
    System.Console.ReadLine();

}
```

Cuando se ejecuta este programa se obtiene el siguiente resultado:

"El libro tiene tres años de antigüedad"

Recuerda que CSharp te permite utilizar variables y nombres de métodos con caracteres latinos como **int añoActual**, pero no es recomendable porque te pueden ocasionar problemas si cambias de computadora y ésta tiene definido en Windows el idioma inglés u otro. Por ello, es preferible **int antiguedad** a **int antigüedad. System.Console.ReadLine()**, para que se detenga y haya tiempo de ver el resultado de la ejecución.

Ten en cuenta que los **métodos miembro** trabajan en conjunto, a ello se le nombra **propiedades**, y es lo que se abordará en el siguiente subtema.

1.1.2. Propiedades

Las **propiedades** son métodos miembros que ofrecen un mecanismo flexible para leer, escribir o calcular los valores de campos privados. Se pueden utilizar las propiedades como si fuesen miembros de datos públicos, aunque en realidad son métodos especiales denominados descriptores de acceso. De este modo, se puede tener acceso a los datos con facilidad, a la vez que proporcionan la seguridad y flexibilidad de los métodos (MSDN, 2013 f).

Para ejemplificar lo que son las propiedades, considera el caso en que se requiera construir un **objeto** llamado **Reloj**, con tres datos miembro: **horas**, **minutos** y **segundos**. Todos ellos de tipo **int** y **privados** para que el concepto de encapsulamiento se aplique, es decir, que sólo sean accesibles por los mismos miembros del objeto. Esto en CSharp se desarrollaría como se muestra en el siguiente listado:

```
class Reloj
{
    private int horas;
    private int minutos;
    private int segundos;
}
```

En este ejemplo se han encapsulado los datos para que no puedan ser modificados desde el exterior, por lo que para poder lograr que sean manejados desde el exterior se utiliza el concepto de **propiedad** que proporciona CSharp.

El formato para declarar una propiedad en CSharp es el siguiente:

```
public <tipo> <nombre>
{
get { return <código lectura> }
set { <código escritura> }
}
```

Dentro de esa propuesta de desarrollo, los estándares de Microsoft (MSDN, 2013 b) plantean que el nombre de las propiedades inicie con una **letra Mayúscula**, teniendo el mismo nombre que la propiedad a la cual van a servir de interfaz. Siguiendo ese delineamiento la propiedad para horas tendría el siguiente código:

```
class Reloj
{
    private int horas;

    public int Horas
    {
        get { return horas; }
        set { horas = value; }
    }

    private int minutos;
    private int segundos;
}
```

Como se observa, se ha implementado una interfaz para horas. En el **método 'get'**, que devuelve el contenido de la **variable horas**, sólo se utiliza la palabra reservada "**return**". En el **método 'set'**, que permite guardar valores dentro de la variable horas, utiliza el parámetro **'value'** que representa el valor que se está recibiendo y se guarda dentro de una variable. Un ejemplo del uso del objeto es el siguiente:

```
class Program
{
    static void Main(string[] args)
    {
        Reloj reloj = new Reloj();
        reloj.Horas = 10;
        System.Console.WriteLine("Horas tecleadas: " + reloj.Horas);
    }
}
```

Se crea una instancia de reloj

Se asigna un valor de diez a la propiedad Horas, esta se la pasa al dato miembro horas

Se imprime el contenido de la variable horas a través de la propiedad Horas

De esta manera se logra el acceso a datos privados mediante el concepto de propiedades. Como puedes observar, el dato **horas** al ser privado sólo se puede acceder a través de la propiedad **Horas**.

Ahora, imagina que necesitas una propiedad de sólo lectura llamada **Hora_Actual**.

Una propiedad de sólo lectura significa que no es posible asignar un valor, sólo será posible ver su contenido.

Considera necesario que cuando se imprima el contenido de **Hora_Actual** se muestre **horas:minutos:segundos.** Una respuesta válida sería **“10:30:22”**.

No sería correcto hacer algo como esto: **reloj.Hora_Actual=“10:30:22”;** porque se está utilizando a la propiedad para escribir un valor y en la propuesta no se permite. Sólo se requiere ver el contenido, es decir, una propiedad de “sólo lectura”.

El código para únicamente ver su contenido, es decir, de “sólo lectura” se muestra a continuación:

```
public string Hora_Actual
{
    get
    {
        string tmp = System.Convert.ToString(horas) + ":" +
            System.Convert.ToString(minutos) + ":" +
            System.Convert.ToString(segundos);
        return tmp;
    }
}
```

En el código puedes observar tres cosas:

1) No es necesario que una propiedad encapsule un valor dentro de la clase. En este caso sólo existe la propiedad **Hora_Actual** pero no está ligada a un dato específico dentro de la clase.

2) No existen instrucciones para el **método set**, por lo tanto, la propiedad no puede recibir valores; será de sólo lectura. Si faltara el código de las **instrucciones para get** y existiera uno para la **instrucción set**, la propiedad sería de sólo escritura, pero no se podría recuperar un valor de la variable.
3) No es forzoso el uso de sólo retornar o tomar valores como parte de las propiedades, es posible hacer cualquier operación dentro de ellas.

Finalmente, **System.Convert.ToString** convierte el dato **horas**, que es de tipo entero, a un dato de tipo **string**.

De acuerdo con lo anterior, se puede resumir que la encapsulación es un mecanismo que permite a los programadores determinar qué datos y métodos miembro serán utilizados por la misma clase y por las otras que compondrán el desarrollo, al mismo tiempo que proporciona una serie de herramientas para lograrlo.

Las principales ventajas de la encapsulación consisten en que

- Facilita a los desarrolladores el ocultar detalles de la implementación interna y sólo deja visibles aquellos que son seguros de usar.
- Cuando se modifica, permite que los cambios posteriores de los objetos no afecten la funcionalidad de las clases derivadas, porque los programadores al no poder acceder no pueden alterarlos.

En los siguientes temas continuarás viendo las ventajas de los otros pilares de la programación orientada a objetos.

1.2. Construcción y destrucción en CSharp

Una de las mejores definiciones de **constructores** y **destructores** se puede encontrar en el libro *La biblia de CSharp* de Ferguson *et ál*. (2005) donde se menciona que:

Los **constructores y los destructores** son métodos especiales, el primero de ellos sirve para inicializar un objeto a un estado conocido que sea necesario para poder utilizarlo. Por el contrario, un **destructor** es un método especial que se ejecuta cuando el CLR (el entorno común de ejecución) destruye los objetos de la clase. Permite liberar recursos (memoria, valores, dispositivos) para que otros objetos dispongan de ellos (p. 423).

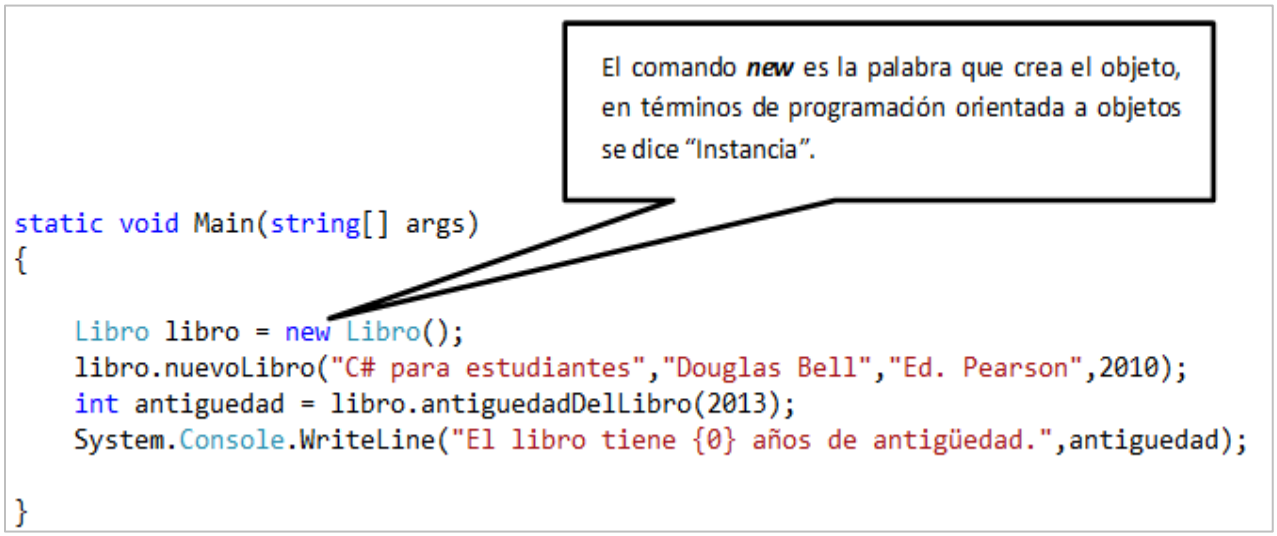
Este es un concepto de la **programación orientada a objetos**, es decir, cualquier lenguaje que soporte este modelo de desarrollo debe proveer herramientas para implementar los métodos especiales.

Así mismo, en la mayoría de los lenguajes, si en el momento de desarrollar un objeto no se declara constructor o destructor, el compilador agregará de forma automática uno de ellos con características especiales.

Ten en cuenta que todos los objetos tienen cuatro fases importantes en su ciclo de vida:

1. **Creación:** reserva de espacio en memoria para un nuevo objeto.
2. **Construcción:** establecimiento de las condiciones iniciales necesarias para manipular un objeto. Se revisará en el subtema 1.2.1. Constructores.
3. **Operación:** es la parte donde el objeto realiza las actividades que se le encargan, mediante la programación de sus métodos. Todos los procesos de desarrollo de las aplicaciones se manejan en este punto.
4. **Destrucción:** establecimiento de las condiciones necesarias para que un objeto pueda desaparecer (entre ellas, la liberación del espacio de memoria). Esto se revisará con detalle en el subtema 1.2.2. Constructores.

Lo correspondiente a la **creación** fue revisado de manera indirecta con anterioridad, y lo recordarás al crear una **instancia de libro**. Para ello, considera el ejemplo revisado durante el primer tema de esta unidad:



En el siguiente subtema se expondrá lo correspondiente a la segunda y cuarta etapa del ciclo de vida de un objeto; es decir, la etapa de construcción y destrucción, dentro de la programación en lenguaje CSharp. También se explicarán los métodos utilizados en cada una de las etapas mencionadas.

1.2.1. Constructores

En relación con el ciclo de vida de un objeto, para este subtema comenzarás con la revisión de la segunda etapa que, como sabes, se refiere al uso de los constructores en CSharp.

Los **constructores** son **métodos de clase** que se ejecutan cuando se crea un objeto de un tipo determinado, en CSharp tienen el mismo nombre que la clase y, normalmente, inicializan los miembros de datos del nuevo objeto (MSDN, 2013 i).

Retomando el ejemplo del inicio de la unidad, si se tiene una clase llamada **Libro** su **constructor** será un **método** también llamado **Libro**.

Un constructor que no toma ningún parámetro se denomina **constructor predeterminado** y si no se escribe uno, el compilador define uno con esas características (sin ningún parámetro).

Cuando se utiliza la instrucción **new** se está invocando al constructor de la clase.

Un constructor tendrá el siguiente formato:

```
[modificadores] nombreDeLaClase ([ Tipo1 Parámetro1, Tipo2 Parámetro2, ...])
{

  Definición de variables locales;   //Sólo accesibles desde el constructor

   Sentencias;
Cuerpo del método.

}
```

Basado en MSDN (2013 i).

En donde:

Modificadores: los constructores sólo pueden ser públicos o privados. Para fines prácticos de esta unidad, todos los **constructores serán públicos**. Los constructores privados tienen un uso muy especial que revisarás más adelante.

NombreDeLaClase: todos los constructores deben llamarse como se llama la clase. Exactamente igual en letra y tipo de ellas (mayúsculas y minúsculas).

Posible lista de parámetros. La lista de parámetros formales es opcional, el constructor podría no tener, cuando los tiene se tratan como variables locales, sólo accesibles desde el interior del constructor. Estos parámetros se separan por comas, llevan el tipo y el nombre. Se inicializan en la llamada recibiendo los valores. Si el constructor no tiene parámetros hay que poner los paréntesis.

El **cuerpo del constructor** tendrá el siguiente formato:

Definición de variables locales: dentro de los constructores se pueden definir variables que sólo son accesibles dentro de ellos. Este tipo de variables no se inicializan por defecto, por lo tanto, se deben inicializar en el momento de su definición o deben de inicializar antes de utilizarlas, pues de lo contrario el compilador detecta un error.

Sentencias: son las instrucciones necesarias para realizar determinada tarea.

Es necesario observar dos puntos importantes:

1. Un constructor no devuelve ningún tipo de dato. Incluso no debe devolver un **tipo void**. Por lo tanto, no se pone ninguna palabra en **tipo**.
2. Como consecuencia de lo anterior, no es válido utilizar la palabra **return** en ningún caso.

Retomando el **objeto** del **reloj**, se agregará un constructor que reciba los parámetros de **horas**, **minutos, segundos**, y los guarde dentro de los **datos miembro** del objeto. El código quedaría como se muestra a continuación:

```
public Reloj(int horas, int minutos, int segundos)
{
    this.horas = horas;
    this.minutos = minutos;
    this.segundos = segundos;
}
```

Los elementos que inician con el operador **this**, se refieren a los datos del propio objeto y los elementos que no inician con esa palabra son los parámetros del constructor.

Importante: es altamente recomendado por Microsoft que se utilice el operador **this** para evitar la creación de una gran cantidad de variables en el momento de la definición de un método o constructor. Esta técnica de programación es muy utilizada en el desarrollo de aplicaciones reales.

Un objeto puede tener **varios constructores**, si supones que se necesita un constructor que no reciba parámetros y por lo tanto inicialice todos los datos miembro en cero. En ese caso el constructor tendría la siguiente apariencia:

```
public Reloj()
{
    this.horas = 0;
    this.minutos = 0;
    this.segundos = 0;
}
```

El compilador distingue que son dos constructores diferentes por el número de parámetros de cada uno. Cuando se invocan en el programa principal se puede ver cómo se construyen dos instancias.

```
static void Main(string[] args)
{

    Reloj reloj1 = new Reloj();
    Reloj reloj2 = new Reloj(10, 32, 43);
    System.Console.WriteLine(reloj1.Hora_Actual);
    System.Console.WriteLine(reloj2.Hora_Actual);

}
```

En el primer caso **reloj1**, sus valores internos de horas, minutos y segundos en 0.

En el segundo caso **reloj2**, sus valores internos de horas son de 10, el de minutos de 32 y el de segundos de 43.

Esto lo podemos ver en los datos que se imprimen:

0:0:0

10:32:43

Ten en cuenta que los constructores pueden realizar cualquier actividad, la única condición es que no pueden devolver un valor. En el siguiente subtema se explicará lo correspondiente a los destructores.

1.2.2. Destructores

En este subtema se abordarán los métodos opuestos a los constructores, es decir, los **destructores.** Si un constructor inicializa datos cuando el objeto es construido, un destructor (como su nombre lo indica) destruye los datos que el constructor inicializó.

Los **destructores** se utilizan para destruir instancias de clases. Como dice el libro de Microsoft, *C# Language Specification* (2010): "es posible comparar un destructor como lo opuesto a un constructor" (p. 484). Un destructor en programación orientada a objetos es una **función miembro especial** que es invocada automáticamente cuando el objeto se deja de referenciar (termina de ejecutarse). Según Eckel Bruce (2010), sus principales actividades son:

1. Liberar los recursos computacionales que el objeto haya adquirido en el tiempo de ejecución al terminar de residir en memoria.
2. Eliminar todas las referencias que tenga de otros recursos u objetos.

De acuerdo con MSDN (2013 c), al contrario de un constructor:

a. Una clase sólo puede tener un destructor.
b. No se puede llamar a los destructores. Se invocan automáticamente.
c. Un destructor no permite modificadores de acceso ni tiene parámetros.

La **sintaxis de un destructor** es la siguiente:

```
~NombreDeLaClase()
{
Código que implementa la funcionalidad del destructor

}
```

Basado en MSDN (2013 c).

Importante: al símbolo ~ se le conoce como **tilde** en los países latinos.

Directamente en el portal de Microsof de CSharp (2013), se dice que “El programador no tiene control sobre cuándo se llama al destructor porque esto está determinado por el recolector de basura”, pero cuando este se llame ejecutará el código que se implemente.

> **Importante**: el *Garbage Collector* o **Recolector de basura** es el responsable de liberar memoria, eliminando todos aquellos objetos que ya no se están utilizando. Es un proceso automático y el usuario no tiene control.

Un ejemplo simple de un recolector de basura es el que se muestra a continuación, retomando el ejemplo de la clase reloj.

```
~Reloj()
{
    horas = 0;
    minutos = 0;
    segundos = 0;
}
```

Las actividades en este destructor son simples. Se limpian las variables del objeto. Esto de manera general no es necesario, pero ejemplifica muy bien una labor final.

Los destructores tienen pocos **usos**, pero el más común es el **cerrado de bases de datos** guardando información a fin de evitar inestabilidades en ellas.

1.3. Sobrecarga

Para continuar con el estudio de la programación orientada a objetos en CSharp es momento de que identifiques el uso de la **sobrecarga**. Para ello necesitas tener en cuenta la **firma de un método**, concepto que se refiere a la combinación del dato que regresa, el nombre del mismo y los argumentos que recibe (Deitel, 2007).

A partir del concepto anterior es posible afirmar que la sobrecarga de métodos es la creación de varios métodos con el mismo nombre, pero con firmas diferentes. En programación orientada a objetos la sobrecarga se refiere a la **posibilidad de tener dos o más funciones con el mismo nombre pero con funcionalidad diferente**, basada en el tipo de parámetros que recibe (Deitel, 2007).

Por ejemplo, si se tiene dos métodos llamados **leerDatos**, pero uno recibe dos enteros y el otro recibe dos números flotantes, se dice que el método **leerDatos** está sobrecargado dos veces. Es posible sobrecargar constructores y métodos (como se explicó en el subtema 1.2.1. Constructores, son similares pero pensados para situaciones distintas).

Los operadores por default están sobrecargados. Por ejemplo, el operador '**+**' puede sumar enteros, flotantes o combinaciones de ellos; lo que representa es la suma de dos valores.

Toma en cuenta que la sobrecarga es una de las características más importantes de la programación orientada a objetos. Esta característica permite implementar algoritmos que cumplen la misma función, pero que tienen parámetros diferentes.

1.3.1. Sobrecarga de constructores

Al ser el constructor un método, también es posible sobrecargarlo, lo cual resulta de gran utilidad cuando se necesita instanciar varios objetos de una forma diferente al momento de crearlos. Un ejemplo podría ser el siguiente:

> Se cuenta con un objeto llamado **Persona**, que puede recibir datos en el momento de instanciarse. Una instancia se forma conociendo el nombre de la persona; otra, conociendo su nombre y su tarjeta de trabajo, pero también es posible instanciar un objeto **Persona** conociendo su tarjeta, nombre, edad y sexo.

En el ejemplo anterior se cuenta con un constructor sobrecargado tres veces.

Para profundizar al respecto recuerda el ejemplo del reloj, considera que necesitas un constructor que reciba la hora en forma de una cadena que reciba el siguiente formato: HH:MM:SS, pero también necesitas un constructor que reciba la hora en forma de número de segundos y lo convierta a horas, minutos y segundos (esto te puede parecer raro, pero es la forma en la que la mayoría de las aplicaciones como Excel o sistemas operativos como Windows o Linux manejan el tiempo). El ejemplo se puede implementar como se muestra a continuación:

```
public Reloj(string tiempo)
{
    string horas = tiempo.Substring(0, 2);
    string minutos = tiempo.Substring(3, 2);
    string segundos = tiempo.Substring(6, 2);
    this.horas = System.Convert.ToInt16(horas);
    this.minutos = System.Convert.ToInt16(minutos);
    this.segundos = System.Convert.ToInt16(segundos);
}
```

```
public Reloj(int tiempo)
{
    this.horas = (tiempo - tiempo % 3600) / 3600;
    tiempo = tiempo % 3600;
    this.minutos = (tiempo - tiempo % 60) / 60;
    this.segundos = tiempo % 60;
}
```

Al analizar el código se puede decir que el constructor está sobrecargado porque los dos reciben el mismo número de parámetros (sólo uno), pero son de diferente tipo y cada uno de ellos tiene un procedimiento diferente para calcular el número de horas, minutos y segundos que tendrá el objeto.

El programa principal instanciará (recuerda que todo programa en CSharp debe instanciarse para poder ejecutarse) a cada uno de los objetos dependiendo del argumento que se le indique, esto se puede observar en el siguiente listado:

```
static void Main(string[] args)
{

    Reloj reloj1 = new Reloj("10:20:30");
    Reloj reloj2 = new Reloj(12245);
    System.Console.WriteLine(reloj1.Hora_Actual);
    System.Console.WriteLine(reloj2.Hora_Actual);

}
```

La primera instancia del **reloj1** recibe como parámetro una cadena, la instancia del **reloj2** recibe un número que representa una hora en segundos. El compilador sabe qué constructor usar en función del parámetro que se está enviando. Esto se puede comprobar en la salida del programa, que será la siguiente:

10:20:30

3:24:5

Un objeto puede tener tantos constructores como sean necesarios, sólo es necesario recordar que el tipo de parámetros que recibe debe cambiar.

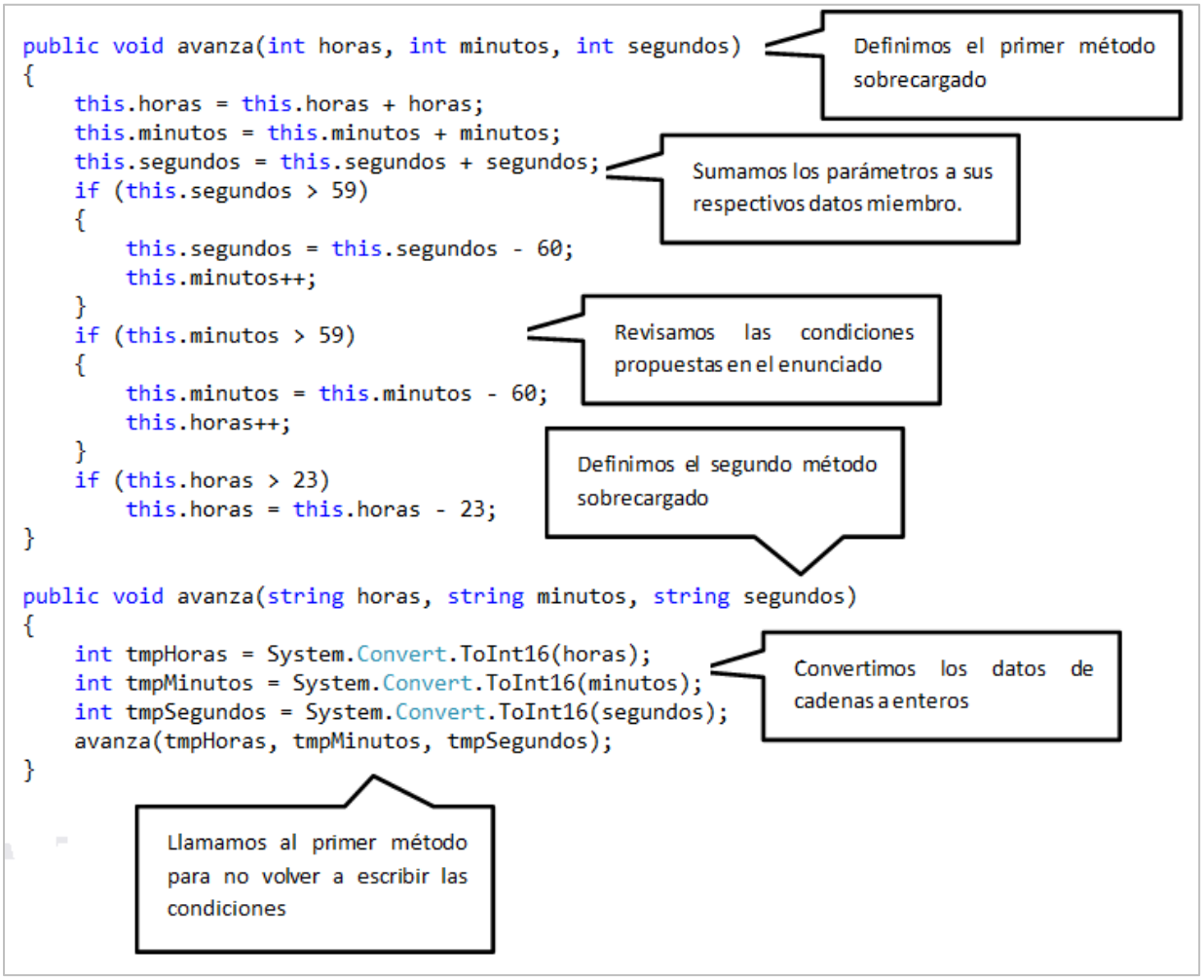
1.3.2. Sobrecarga de métodos

Según MSDN (2013 g), la **sobrecarga** consiste en crear más de un método en una clase con el mismo nombre y distintos tipos de argumentos. Esto es muy práctico, pues no tienes que renombrar cada función según el tipo de valor que acepta.

En el ejemplo del objeto **reloj**, en caso de que se presenten dos requerimientos, el primero sería un método llamado avanza, el cual recibiría un valor en horas, minutos y segundos; el segundo método debería estar sobrecargado, uno de ellos recibiría tres enteros y las otras tres cadenas que contendrían horas, minutos y segundos.

Primer requerimiento: un método que se llame **avanza** y reciba un **valor** en **horas**, **minutos** y **segundos**. Se debe aumentar estos valores al **reloj**, pero recuerda que al sumar los segundos o minutos del parámetro a de la clase, no pueden ser mayor que 59 porque al ser 60 aumenta un minuto o una hora más, dependiendo del parámetro que se trate. También debes cuidar que las horas, al sumarse las del parámetro con las del objeto, no sean mayor que 23. El reloj deberá de iniciarse a 0.

Segundo requerimiento: el método debe estar sobrecargado, uno de ellos recibirá tres enteros y otro recibirá tres cadenas que contendrán las horas, los minutos y los segundos. Es posible ver esto en el siguiente listado:




```
public void avanza(int horas, int minutos, int segundos)
{
    this.horas = this.horas + horas;
    this.minutos = this.minutos + minutos;
    this.segundos = this.segundos + segundos;
    if (this.segundos > 59)
    {
        this.segundos = this.segundos - 60;
        this.minutos++;
    }
    if (this.minutos > 59)
    {
        this.minutos = this.minutos - 60;
        this.horas++;
    }
    if (this.horas > 23)
        this.horas = this.horas - 23;
}

public void avanza(string horas, string minutos, string segundos)
{
    int tmpHoras = System.Convert.ToInt16(horas);
    int tmpMinutos = System.Convert.ToInt16(minutos);
    int tmpSegundos = System.Convert.ToInt16(segundos);
    avanza(tmpHoras, tmpMinutos, tmpSegundos);
}
```

Puede concluirse que cada método sobrecargado es capaz tener su propio código. En el ejemplo anterior también se observa cómo un método puede llamar a otro, siempre y cuando utilice los parámetros adecuados.

```
static void Main(string[] args)
{

    Reloj reloj1 = new Reloj("10:20:30");
    Reloj reloj2 = new Reloj(12245);
    reloj1.avanza(10, 10, 55);
    reloj2.avanza("8", "46", "24");
    System.Console.WriteLine(reloj1.Hora_Actual);
    System.Console.WriteLine(reloj2.Hora_Actual);

}
```

Como puede observarse en el programa principal **Main**, el uso de los constructores y métodos sobrecargados puede ser de manera libre de acuerdo con ciertas necesidades. El compilador C# .NET sabrá cuál utilizar de acuerdo con el tipo de parámetros (para fines prácticos de esta unidad didáctica, compilador de C Sharp, C#, Visual CSharp, C# .NET o CSharp .NET son sinónimos). La siguiente lista muestra ejemplos de métodos sobrecargados válidos:

```
1 public string suma(double a, double b)
2 public  string suma(double a, int b)
3 public int suma(int da, int db)
```

Las tres firmas son válidas porque tienen distinto tipo de parámetros que reciben. En el caso de la firma tres, no es trascendente que devuelva un entero ni que los parámetros tengan otro nombre, lo importante es el número de parámetros y su tipo.

Ejemplo de firmas no válidas:

```
4 public string suma(double da, double db)
5 public int suma(double a, double b)
```

En la firma 4 se observa un error. No importa que cambie el nombre de los parámetros, su tipo es igual al ejemplo de la firma 1.

En la firma 5 se observa un error. No importa que el valor de retorno cambie, el tipo y número de parámetros es igual al ejemplo de la firma 1.

Se concluye que la sobrecarga es una herramienta útil en el desarrollo de las aplicaciones, al dar un nombre único a procesos que esencialmente son iguales y en los que sólo se cambian los

valores con los que se trabaja. El caso más notorio antes de que el polimorfismo existiera es el del **lenguaje C**, donde existían los métodos llamados **itoa**, **ftoa** y **ltoa**, todos ellos devuelven una cadena, pero la primera recibe un entero, la segunda un **float** y la tercera un **long (**métodos que se eliminaron al implementarse la programación orientada a objetos). Con el polimorfismo esto se puede evitar al dar un sólo nombre al método y los parámetros le dirán al compilador qué operación debe hacer.

1.3.3. Sobrecarga de operadores

Como se mencionó en el tema 1.3. Sobrecarga, un operador es naturalmente sobrecargado porque el mismo operador permite hacer diversas operaciones con parámetros diferentes. Por ejemplo: el operador '+'

Float + Float realiza la suma sobre dos flotantes.

Integer + Integer realiza una suma sobre dos enteros.

String + String realiza la concatenación sobre dos cadenas.

Y puede probarse cualquier combinación de parámetros.

CSharp es uno de los pocos lenguajes que permite reescribir (sobrecargar) un operador, agregándole nuevas funcionalidades. Para ello, según el MSDN (2013 h), se deben seguir las siguientes reglas:

No se debe realizar sobrecarga a operadores de forma que realicen operaciones no naturales. Por ejemplo, sobrecargar el operador '+' para que realice una multiplicación.

1. Algunos operadores deben sobrecargarse en pares, por ejemplo, al sobrecargarse el operador de igualdad '==', también se debe modificar el de desigualdad '!=', junto con '>' también debe modificarse '>='.

La siguiente tabla muestra la posibilidad de sobrecarga de los operadores (MSDN, 2013 c):

Operadores	Posibilidad de sobrecarga
+, -, !, ~, ++, --, true, false	Estos operadores unarios sí pueden sobrecargarse.
+, -, *, /, %, &, |, ^, <<, >>	Estos operadores binarios sí pueden sobrecargarse.
==, !=, <, >, <=, >=	Los operadores de comparación pueden sobrecargarse, pero deben ser en pares
&&, ||	Los operadores lógicos condicionales no pueden sobrecargarse, pero se evalúan mediante '&' y '|', los cuales sí pueden sobrecargarse.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Los operadores de asignación no pueden sobrecargarse, pero '+=', por ejemplo, se evalúa con '+', el cual sí puede sobrecargarse.
=, ., ?:, ->, new, is, sizeof, typeof	Estos operadores no pueden sobrecargarse.

Tabla 1. Posibilidad de sobrecarga de operadores (MSDN, 2013 c).

La sintaxis para sobrecargar un operador es la siguiente:

```
public static ValorDeRetorno operator
OperadorASobreCarger(Tipo c1, Tipo c2)
{
Procedimiento
}
```

Basado en MSDN, (2013 c)

Las palabras en cursivas son palabras reservadas obligatorias, donde:

ValorDeRetorno: es el tipo de dato que retornará el operador.

OperadorASobreCargar: alguno de la tabla anterior.

Tipo: valores que recibirá el operador.

Ahora un ejemplo.

Recordarás que toda fracción se representa de la siguiente forma:

$$\frac{Numerador}{Denominador}$$

Y que la suma de dos fracciones se realiza de la siguiente manera:

$$\frac{a}{b}+\frac{c}{d}=\frac{a \times d+b \times c}{b \times d}$$

Con los anteriores conceptos se desarrollará un objeto que se llamará fracción. Tendrá dos datos miembros de tipo entero llamados Numerador y Denominador. Estos se recibirán en el constructor. Se tendrá un método llamado **imprime** que devolverá la fracción en el formato Numerador/Denominador.

Se sobrecargará el operador '+'que permita sumar dos fracciones. El código del enunciado se muestra a continuación:

```
class Fraccion
{

    private int numerador;
    private int denominador;

    public Fraccion(int numerador, int denominador)
    {
        this.numerador = numerador;
        this.denominador = denominador;
    }
```

```
    public string imprime()
    {
        return numerador + "/" + denominador;
    }

    public static Fraccion operator +(Fraccion F1, Fraccion F2)
    {
        Fraccion fraccionSuma = new Fraccion(0, 0);
        fraccionSuma.numerador = F1.numerador * F2.denominador
            + F2.numerador * F1.denominador;
        fraccionSuma.denominador = F1.denominador * F2.denominador;
        return fraccionSuma;
    }
}
```

El uso de esta clase se muestra a continuación:

```
static void Main(string[] args)
{

    Fraccion Fra1 = new Fraccion(1, 2);
    Fraccion Fra2 = new Fraccion(1, 3);
    Fraccion Fra3 = Fra1 + Fra2;

    System.Console.WriteLine("Fracción 1   : "+ Fra1.imprime());
    System.Console.WriteLine("Fracción 2   : "+ Fra2.imprime());
    System.Console.WriteLine("Fracción Suma: "+ Fra3.imprime());

}
```

Fracción 1: 1/2

Fracción 2: 1/3

Fracción Suma: 5/6

Como se aprecia en el ejemplo, el operador '+' se ha sobrecargado y en este programa también se permite sumar fracciones.

Hasta aquí se concluye el desarrollo de los temas 1.2. Construcción y destrucción en CSharp, y 1.3. Sobrecarga. Es momento de que realices algunas actividades de aprendizaje.

Cierre de la unidad

Durante el desarrollo de esta unidad abordaste los temas de **encapsulación**, **construcción**, **destrucción** y **sobrecarga**, los cuales te muestran cómo se pueden desarrollar en **CSharp** los conceptos de programación orientada a objetos.

Ten en cuenta que el tema de **encapsulación** es pilar fundamental de la programación orientada a objetos, pues significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema (MSDN, 2013). Como viste en los ejemplos, sus propiedades te dan control completo del interior del objeto y cierra el contenido a manejos indebidos.

La **construcción y la destrucción** son métodos utilizados en estas dos etapas (cuando el objeto es instanciado y cuando deja de ser referenciado), muy importantes dentro de la vida de un objeto, cuando se construye y cuando se destruye.

Importante: recuerda que todos los objetos tienen un ciclo de vida.

En lenguajes orientados a objetos, todo objeto tiene un ciclo de vida (nace cuando se instancia, vive cuando ejecuta sus métodos de trabajo, muere cuando se deja de hacer referencia a él); en la primera etapa se utiliza un constructor y en la segunda un destructor. Estos métodos son de suma importancia porque hacen énfasis en estos dos momentos en la vida de un objeto: la construcción y la destrucción. En las unidades subsecuentes verás otros métodos que se encargan de otros momentos en dicho ciclo. Ten siempre en cuenta que el ciclo de vida de un

objeto empieza por su declaración, instanciación y uso en un programa CSharp, hasta que desaparece.

Finalmente, la **sobrecarga** se refiere al uso del mismo identificador y sus parámetros, pero en distintos contextos (distinto tipo). En la siguiente unidad verás que este concepto es el inicio de algo aún más complejo y útil llamado **polimorfismo,** donde la sobrecarga pasa al siguiente nivel, no sólo cambia el tipo de parámetros sino también el número de ellos.

Te espera una unidad muy interesante, donde aprenderás nuevos conceptos de programación orientada a objetos mediante CSharp, en la plataforma .NET.

Para saber más

En esta sección se presentan algunos recursos que es recomendable consultes para profundizar en el estudio de los principales temas revisados en la unidad.

- La primera recomendación es la base de conocimientos que se encuentra en el portal oficial de CSharp en español, propuesto por Microsoft; ahí encontrarás la explicación y varios ejemplos de encapsulación, construcción, destrucción y sobrecarga. Esta página es válida para cualquier versión, desde 2008 hasta 2015. Se encuentra disponible en: https://docs.microsoft.com/es-es/visualstudio/csharp-ide/encapsulate-field-refactoring-csharp?view=vs-2015&redirectedfrom=MSDN
- El libro de José Antonio González Seco, *El lenguaje de programación C#.* Es un desarrollo libre y, aunque es una versión muy anterior, los conceptos fundamentales no han cambiado.
- En el siguiente portal hallarás varios videos de algunos conceptos importantes de CSharp. Se encuentra en su versión en español, para que puedas acceder a la información y observar los videos. Este sitio también es parte de la base de conocimientos de CSharp y está disponible en https://csharp.com.es/
- Página oficial de Visual Estudio: http://msdn.microsoft.com/en-us/vstudio//bb798022.aspx

Fuentes de consulta

Básicas

Archer, T. (2010). *A fondo C#*. Madrid: McGraw-Hill.

Deitel, M. (2004). *Cómo programar en CShar y Java*, 4a. ed. España: Pearson.

(2007). *Cómo programar en CSharp,* 2da. ed. España: Pearson.

Eckel, B. y O'Brien, L. (2010). *Thinking in C#*. Estados Unidos: Prentice Hall.

Ferguson J. *et al*. (2005). *La biblia de CSharp.* Madrid: Anaya.

González Seco, J.A. (s/f). *Lenguaje de programación en C#*.

Michaellis, M. (2010). *Essential C# 4.0*. Estados Unidos: Addison Wesley.

Microsoft Corporation. (1999-2007). *C# Language Specification Version 3.0*. Estados Unidos: Microsoft Corporation.

Microsoft. (2022). Exploración de la programación orientada a objetos con clases y objetos. https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/tutorials/classes

Microsoft. (2022). Programación orientada a objetos (C#). https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/tutorials/oop

MSDN. Microsoft Developer Network. (2013 a). *Clases y estructuras (Guía de programación de C#).* https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/

(2013 b). *Normas referentes al uso de minúsculas y mayúsculas.* *http://msdn.microsoft.com/es-es/library/ms173109%28v=vs.90%29.aspx*

(2022 c). *Destructores (Guía de programación de C#)*. *https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/destructors*

(2022 d). *Operadores sobrecargables (Guía de programación de C#)*. https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/operator-overloading?view=toolsforcordova-2015#overloadable-operators

(2022 e). *Programación orientada a objetos C# y Visual Basic*. https://docs.microsoft.com/es-es/dotnet/visual-basic/programming-guide/concepts/object-oriented-programming

(2022 f). *Propiedades (Guía de programación de C#)*. https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/properties

(2022 g). *Propiedades y métodos sobrecargados.* http://msdn.microsoft.com/es-es/library/1z71zbeh%28v=VS.90%29.aspx

(2022 h). *Sobrecarga de operadores (C# y Java)*. https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/operator-overloading?view=toolsforcordova-2015

(2022 i). *Utilizar constructores (Guía de programación de C#).* https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/using-constructors

Peláez, M. (2005). Java: Conceptos básicos de P.O.O, curso desarrollo de aplicaciones informáticas. *I.E.S. Galileo de Valladolid*.

Román, C. *Temas especiales de computación. Programación con Java*. http://profesores.fi-b.unam.mx/carlos/java/indice.html

Sharp, J. (2010). *Microsoft Visual C# 2010*. Estados Unidos: Microsoft Press.

Troelsen, A. (2010). *Pro C# 2010 and the .NET 4 Platform*. Estados Unidos: Apress.