|  |
| --- |
| FUNDAMENTOS DE PROGRAMACIÓN |
|  |
| Diseño: Ing. Héctor Javier Padilla Lara |

**Nombre de la asignatura:** Fundamentos de Programación

**Carrera:** Ing. En Sistemas Computacionales

**Clave de la asignatura:** ACU-0402

**Horas teoría-Horas práctica-Créditos:** 1-2- 4

# Objetivo General del Curso

Contenido

[Objetivo General del Curso 2](#_Toc269889051)

[UNIDAD I. Introducción a la programación orientada a objetos 4](#_Toc269889052)

[1. Conceptos básicos de programación orientada a objetos. 4](#_Toc269889053)

[¿Qué es la POO? 4](#_Toc269889054)

[Objetos 5](#_Toc269889055)

[Clases 5](#_Toc269889056)

[Estructura interna de un objeto 6](#_Toc269889057)

[Métodos y Mensajes 6](#_Toc269889058)

[Envió de mensajes 7](#_Toc269889059)

[Comprender propiedades, métodos y eventos 7](#_Toc269889060)

[Abstracción 9](#_Toc269889061)

[Encapsulación 9](#_Toc269889062)

[Herencia 9](#_Toc269889063)

[Polimorfismo 9](#_Toc269889064)

[UNIDAD 2. Clases y Objetos 10](#_Toc269889065)

[1. Clases y objetos 10](#_Toc269889066)

[1.1 Declaración de una clase 10](#_Toc269889067)

[1.1.1 Miembros de una clase 10](#_Toc269889068)

[1.2 Tipos de datos 12](#_Toc269889069)

[1.3 Operadores 12](#_Toc269889070)

[Operaciones lógicas: 13](#_Toc269889071)

[Operador condicional: 14](#_Toc269889072)

[1.5 Conversiones 15](#_Toc269889073)

[Conversiones implícitas 15](#_Toc269889074)

[Conversiones explícitas 16](#_Toc269889075)

[1.6 Modificadores de acceso 16](#_Toc269889076)

[1.7 Propiedades 17](#_Toc269889077)

[UNIDAD 4. Estructuras de condición 19](#_Toc269889078)

[1. Estructura IF 19](#_Toc269889079)

[Instrucción If … Else 20](#_Toc269889080)

[Instrucción If … Else If 20](#_Toc269889081)

[2. Estructura SWITCH 22](#_Toc269889082)

[Unidad V. Estructuras repetitivas 25](#_Toc269889083)

[Tema 2.1 25](#_Toc269889084)

[Tema 2.2 25](#_Toc269889085)

[Tema 2.3 25](#_Toc269889086)

[Tema 2.3.1 25](#_Toc269889087)

[Unidad II. Nombre de la unidad 25](#_Toc269889088)

[Tema 2.1 25](#_Toc269889089)

[Tema 2.2 25](#_Toc269889090)

[Tema 2.3 25](#_Toc269889091)

[Tema 2.3.1 25](#_Toc269889092)

[Anexos 27](#_Toc269889093)

[Bibliografía 28](#_Toc269889094)

# UNIDAD I. Introducción a la programación orientada a objetos

### Conceptos básicos de programación orientada a objetos.

### ¿Qué es la POO?

Las siglas POO se corresponden con Programación Orientada a Objetos, aunque muchas veces las podemos encontrar escritas en inglés OOP (Object Oriented Programming).

La Programación Orientada a Objetos trata de utilizar una visión real del mundo dentro de nuestros programas. La visión que se tiene del mundo dentro de la POO es que se encuentra formado por objetos.

La POO es una forma de abordar el problema más natural. Al decir natural significa más en contacto con el mundo real que nos rodea, de esta forma si queremos resolver un problema determinado, debemos identificar cada una de las partes del problema con objetos presentes en el mundo real.

### Objetos

Un objeto es una representación o modelización de un objeto real perteneciente a nuestro mundo, por ejemplo, podemos tener un objeto perro que represente a un perro dentro de nuestra realidad, o bien un objeto factura, cliente o pedido.

Los objetos en la vida real tienen todos, dos características: estado y comportamiento.

El estado de un objeto viene definido por una serie de valores que lo definen y que lo diferencian de objetos del mismo tipo. En el caso de tener un objeto perro, su estado estaría definido por su raza, color de pelo, tamaño, etc.

Y el comportamiento viene definido por las acciones que pueden realizar los objetos, por ejemplo, en el caso del perro su comportamiento sería: saltar, correr, ladrar, etc. El comportamiento permite distinguir a objetos de distinto tipo, así por ejemplo el objeto perro tendrá un comportamiento distinto a un objeto gato.

En el mundo real, frecuentemente existen muchos objetos de un mismo tipo. Por ejemplo, tu reloj es uno de los muchos relojes que existen en el mundo.

Usando terminología orientada a objetos podemos decir que tu objeto reloj es una instancia de una clase de objetos conocidos como relojes.

Los relojes poseen el mismo estado (color, marca) y comportamiento (dar la hora,…) parecido. Sin embargo las características de cada reloj son independientes y pueden ser diferentes de los demás relojes.

Los parámetros o variables que definen el estado de un objeto se denominan atributos o variables miembro y las acciones que pueden realizar los objetos se denominan métodos o funciones miembro.

### Clases

Una clase es un molde, patrón o prototipo que define un tipo de objeto determinado. Un objeto es, por lo tanto, el "resultado" de una clase.

Mediante las clases podremos crear o instanciar objetos de un mismo tipo, estos objetos se distinguirán unos de otros a través de su estado, es decir, el valor de sus atributos.

La clase la vamos a utilizar para definir la estructura de un objeto, es decir, estado y comportamiento.

Cuando construyen bicicletas, los fabricantes sacan ventaja del hecho que las bicicletas comparten características para construir muchas bicicletas del mismo patrón. Sería muy ineficiente producir un nuevo patrón para cada bicicleta que ellos construyan.

En el software, también es posible tener muchos objetos del mismo tipo y que comparten características: rectángulos, registros de empleados, videoclips, entre otros. Como los fabricantes de bicicletas, puedes tomar ventaja del hecho que los objetos son similares para crear un patrón para esos objetos, es decir una clase.

La clase es un concepto que no se va a utilizar directamente en nuestros programas o aplicaciones. Lo que vamos a utilizar van a ser objetos concretos creados (instancias) de una clase determinada. La clase es algo genérico y abstracto, es similar a una idea. Cuando decimos piensa en un coche todos tenemos en mente la idea general de un coche, con puertas, ruedas, un volante, etc., sin embargo cuando decimos "ese coche que está estacionado ahí fuera", ya se trata de un coche determinado, con una matrícula, de un color, con un determinado número de puertas, y que podemos tocar y utilizar si es necesario.

Un ejemplo que se suele utilizar para diferenciar clases y objetos es el ejemplo del molde de galletas. El molde para hacer galletas sería una clase, y las galletas que hacemos a partir de ese molde ya son objetos concretos creados a partir de las características definidas por el molde.

Las clases ofrecen el beneficio de la reutilización, utilizaremos la misma clase para crear varios objetos. El programador tiene la responsabilidad absoluta de crear clases propias, pero también puede tener acceso a las clases desarrolladas por otros.

### Estructura interna de un objeto

La estructura interna de un objeto está compuesta por tres elementos fundamentales como son:

* Propiedades: Son las características observables de un objeto. Las propiedades se reconocen porque describen un aspecto del objeto que podemos medir con una escala establecida previamente. A cada propiedad se le debe asignar un valor el cual permite identificar de manera única al objeto.
* Métodos: Se define como un conjunto de acciones que un objeto puede realizar para conseguir un propósito. Los métodos representan la parte viva e interesante de un objeto y se emplean habitualmente para modificar las propiedades del objeto. Al modificar cualquiera de las propiedades del objeto se altera su apariencia y se genera un cambio que el usuario de la aplicación puede percibir.
* Eventos: Todos los objetos se relacionan con el mundo que los rodea, esto significa que ningún objeto está aislado y siempre recibe el influjo de otros objetos. Los eventos son los estímulos que un objeto ejerce sobre otro

### Métodos y Mensajes

Los mensajes son la forma que tienen de comunicarse distintos objetos entre sí. Un objeto por sí sólo no es demasiado útil, sino que se suele utilizar dentro de una aplicación o programa que utiliza otros objetos.

El comportamiento de un objeto está reflejado en los mensajes a los que dicho objeto puede responder. Representan las acciones que un determinado objeto puede realizar.

Un mensaje enviado a un objeto representa la invocación de un determinado método sobre dicho objeto, es decir, la ejecución de una operación sobre el objeto. Es la manera en la que un objeto utiliza a otro, el modo en el que dos objetos se comunican, ya que la ejecución de ese método retornará el estado del objeto invocado o lo modificará.

Los mensajes se utilizan para que distintos objetos puedan interactuar entre sí y den lugar a una funcionalidad más compleja que la que ofrecen por separado. Un objeto lanzará o enviará un mensaje a otro objeto si necesita utilizar un método del segundo objeto. De esta forma si el objeto A quiere utilizar un método del objeto B, le enviará un mensaje al objeto A.

Podemos decir que el objeto persona envía el mensaje "girar a la izquierda" al objeto bicicleta.

Los mensajes pueden contener parámetros. Los parámetros se utilizan para enviar valores que el método necesita para realizar su función. Por ejemplo teniendo un método en la clase bicicleta llamado "Girar" que recibe como parámetro la dirección (derecha o izquierda).

Un mensaje está compuesto por los siguientes tres elementos:

1 - El objeto destino, hacia el cual el mensaje es enviado

2 - El nombre del método a llamar

3 - Los parámetros solicitados por el método

### Envió de mensajes

En un sistema, los objetos trabajan en conjunto. Esto se logra mediante el envío de mensajes entre ellos. Un objeto envía un mensaje para realizar una operación, y el objeto ejecutará la operación.

Una televisión y su control remoto puede ser un ejemplo del mundo que nos rodea. Cuando desea ver un programa de televisión, busca el control remoto, se sienta en su silla favorita y presiona el botón de encendido. ¿Qué ocurre? El control remoto le envía, literalmente, un mensaje al televisor para que se encienda. El televisor recibe el mensaje, lo identifica como una petición para encenderse y así lo hace. Cuando desea ver otro canal, presiona el botón correspondiente del control remoto, mismo que envía otro mensaje a la televisión (cambiar de canal). El control remoto también puede comunicar otros mensajes como ajustar el volumen, silenciar y activar los subtítulos.

Muchas de las cosas que hace mediante el control remoto, también las podrá hacer si se levanta de la silla, va a la televisión y presiona los botones correspondientes. La interfaz que la televisión le presenta (el conjunto de botones y perillas) no es, obviamente, la misma que le muestra al control remoto (un receptor de rayos infrarrojos).

### Comprender propiedades, métodos y eventos

Un objeto corriente como un globo de helio también tiene propiedades, métodos y eventos. Entre las propiedades de un globo se incluyen atributos visibles como su alto, diámetro y color. Otras propiedades describen su estado (inflado o desinflado), y atributos que no se ven, como su tiempo de fabricación. Todos los globos tienen estas propiedades, aunque sus valores pueden diferir de un globo a otro.

Un globo también tiene métodos o acciones conocidas que puede realizar. Tiene un método para inflarse (llenarlo con helio), un método para desinflarse (expulsar su contenido) y un método para elevarse (soltarlo). De la misma manera, todos los globos pueden tener estos métodos.

Los globos también tienen respuestas a ciertos eventos externos. Por ejemplo, un globo responde al evento de ser pinchado desinflándose o al evento de ser soltado elevándose.

Propiedades

Globo.Color = Rojo

Globo.Diametro = 10

Globo.Inflado = Si

El orden del código es el siguiente: el objeto destino (Globo) seguido por la propiedad (Color) seguida por la asignación del valor (= Rojo). Puede cambiar el color del globo sustituyendo un valor diferente.

Métodos

Los métodos de un globo se denominan de este modo.

Globo.Inflar

Globo.Desinflar

Globo.Elevar(5) “5 metros”

El orden es parecido al de una propiedad: el objeto (un nombre), seguido por el método (regularmente un verbo). En el tercer método, hay un elemento adicional, llamado parámetro, que especifica la distancia a que se elevará el globo. Algunos métodos tendrán uno o más parámetros para describir aún más la acción que se va a realizar.

Eventos

El globo podría responder a un evento de la siguiente manera.

Globo.HacerRuido("Bang")

Globo\_Pinchado() Globo.Desinflar

Globo.Inflado = No

En este caso, el código describe el comportamiento del globo cuando se produce un evento Pinchado: llama al método HacerRuido con un parámetro "Bang", (el tipo de ruido a realizar), luego llama al método Desinflar. Puesto que el globo ya no está inflado, la propiedad Inflado se establece en No.

### Abstracción

La abstracción es el proceso de simplificar un problema complejo. Al enfrentarse a un problema un problema, uno no debería abrumarse con cada uno de los detalles, mas bien, simplificarlo enfocándose tan sólo en los aspectos relevantes para encontrar la solución.

Una abstracción se centra en la vista externa de un objeto, de un modo que sirva para separar el comportamiento esencial de un objeto.

### Encapsulación

La encapsulación o encapsulamiento es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B, y viceversa.

### Herencia

La herencia es aquella propiedad de la programación orientada a objetos que le permite a una clase, llamada clase derivada, compartir la estructura y el comportamiento de otra clase, llamada clase base.

### Polimorfismo

Característica fundamental de la POO, que no es otra cosa que la posibilidad de construir varios métodos con el mismo nombre, pero con relación a la clase a la que pertenece cada uno, con comportamientos diferentes.

Polimorfismo es la propiedad que indica, literalmente, la posibilidad de que una entidad tome muchas formas. En términos prácticos, el polimorfismo permite a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas, según sea el objeto que se referencia en ese momento.

# UNIDAD 2. Clases y Objetos

## Clases y objetos

C# es un lenguaje orientado a objetos. Y el concepto fundamental en torno a la orientación a objetos es la clase.

Una clase es como una plantilla que describe cómo deben ser las instancias de dicha clase, de forma que cuando creamos una instancia, ésta tendrá exactamente los mismos métodos y variables que los que tiene la clase.

Los datos y métodos contenidos en una clase se llaman miembros de la clase y se accede a ellos siempre mediante el operador “.”.

## 1.1 Declaración de una clase

La sintaxis básica para definir una clase es la que a continuación se muestra:

class <nombreClase>  
{  
  <miembros>  
}

Por ejemplo:

class Persona  
 {  
   **string** Nombre;

**int** Edad;

**string** CURP;

}

## 1.1.1 Miembros de una clase

Los miembros de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma.

Una declaración de una clase puede contener declaraciones de campos, métodos, propiedades, eventos, indexadores, operadores, constructores, destructores y tipos.

**Campos**. Para poder resolver un problema empleando una computadora es necesario contar con algún mecanismo para almacenar datos en forma temporal. Esto se logra con las variables, las que en el caso de ser parte de una clase, se les conoce como campos o atributos.

Toda variable debe tener un tipo de dato asociado, que define básicamente la clase de dato que se almacenará en ella y la forma en que deberá ser tratado.

Toda variable debe ser declarada previamente a su uso dentro del programa, y esto debe hacerse respetando una sintaxis establecida.

Para definir cuáles son los campos de los que una clase dispone se usa la siguiente sintaxis:

<tipoCampo> <nombreCampo>;

class Persona  
 {  
   string Nombre;

   int Edad;

   string CURP;

}

Ejemplos de declaraciones de variables:

int numero;

char sigla;

string nombre;

Ejemplos de declaraciones de variables asociadas a un dato:

int numero = 10;

char sigla = ’G’;

string nombre = “Juan”;

  Para acceder a un campo de un determinado objeto se usa la sintaxis:

<objeto>.<campo>

            Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y   cambiar su valor por 20 se haría:

p.Edad = 20;

## 1.2 Tipos de datos

Una de las cosas más importantes al empezar a estudiar un lenguaje es saber que tipos de datos primitivos tenemos para nuestro uso común, el contenido de los mismos, sus relaciones

La mayoría de los tipos básicos de C# provienen de los tipos del C++, y son los siguientes

|  |  |  |
| --- | --- | --- |
| Tipo de Datos | Descripción | Ejemplo |
| object | Es el tipo base de todos los tipos | object obj=null; |
| string | Secuencia de caracteres Unicode | string st="Sierra"; |
| sbyte | entero con signo de 8 bit (1 byte) | sbyte val=12; |
| short | entero con signo de 16 bit | short val=12; |
| int | entero con signo de 32 bit | int val=12; |
| long | entero con signo de 64 bit | long val=12; long val2=34L; |
| bool | tipo booleano (cierto - falso) | bool opc=false; bool opc=true; |
| char | tipo caracter, se corresponde con un Unicode | char val='h'; |
| byte | entero sin signo de 8 bit | byte val=12; byte val2=12U; |
| ushort | entero sin signo de 16 bit | ushort val=12; ushort val2=12U; |
| uint | entero sin signo de 32 bit | uint val=12; uint val2=12U; |
| ulong | entero sin signo de 64 bit | ulong val=12; ulongval2=12U; ulong val3=24L ulong val4=34UL |
| float | numero en punto flotante con precision simple | float val=12.23F float val2=12.23 |
| double | numero en punto flatante con precision doble | double val=12.23 double val2=12.23D |
| decimal | numero decima con 28 digitos significativos | decimal val=1.23M |

## 1.3 Operadores

Un operador en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.   
  
A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:   
  
Operaciones aritméticas:   
Los operadores aritméticos incluidos en C# son los típicos de suma (+), resta (-), producto (\*), división (/) y módulo (%)

Operaciones lógicas:  
 Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (&& y &), "or" (|| y |), "not" (!) y "xor" (^)   
Los operadores && y || se diferencia de & y | en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación && es falso se devuelve false directamente, sin evaluar el segundo; y si el primer operando de una || es cierto se devuelve true directamente, sin evaluar el otro.   
  
Operaciones relacionales:   
Se han incluido los tradicionales operadores de igualdad (==), desigualdad (!=), "mayor que" (>), "menor que" (<), "mayor o igual que" (>=) y "menor o igual que" (<=)   
  
Operaciones de asignación:  
 Para realizar asignaciones se usa en C# el operador =, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión a = b asigna a la variable a el valor de la variable b y devuelve dicho valor, mientras que la expresión c = a = b asigna a c y a el valor de b (el operador = es asociativo por la derecha)   
También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:   
  
temperatura = temperatura + 15;   // Sin usar asignación compuesta  
temperatura += 15;                // Usando asignación compuesta

Las dos líneas anteriores son equivalentes, pues el operador compuesto += lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando. Como se ve, permite compactar bastante el código.

Aparte del operador de asignación compuesto +=, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: +=, -=, \*=, /=, %=, &=, |=, ^=, <<= y >>=. Nótese que no hay versiones compuestas para los operadores binarios && y ||.   
  
Otros dos operadores de asignación incluidos son los de incremento (++) y decremento (--) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

temperatura = temperatura + 1; // Sin usar asignación compuesta ni

incremento

temperatura += 1;              // Usando asignación compuesta  
temperatura++;                 // Usando incremento

Si el operador ++ se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador --. Por ejemplo:   
  
c = b++; // Se asigna a c el valor de b y luego se incrementa b  
c = ++b; // Se incrementa el valor de b y luego se asigna a c

La ventaja de usar los operadores ++ y -- es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador traducirlos en una única instrucción en código máquina.   
  
Operadores de cadenas:  
 Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo,

"Hola"+" mundo" devuelve "Hola mundo", y

"Hola mund" + "o" también.

Operador condicional:  
 Es el único operador incluido en C# que toma 3 operandos, y se usa así:

<condición> ? <expresión1> : <expresión2>

El significado del operando es el siguiente: se evalúa <condición> Si es cierta se devuelve el resultado de evaluar <expresión1>, y si es falsa se devuelve el resultado de evaluar <condición2>. Un ejemplo de su uso es:   
  
b = (a>0)? a : 0;      // Suponemos a y b de tipos enteros

En este ejemplo, si el valor de la variable a es superior a 0 se   asignará a b el valor de a, mientras que en caso contrario el valor que se le asignará será 0.   
  
1.4 Variables

Una variable puede verse simp**l**emente como un hueco en el que se puede almacenar un objeto de un determinado tipo al que se le da un cierto nombre. Para poderla utilizar sólo hay que definirla indicando cual será su nombre y cuál será el tipo de datos que podrá almacenar, lo que se hace siguiendo la siguiente sintaxis:

<tipoVariable> <nombreVariable>;

    Una variable puede ser definida dentro de una definición de clase, en cuyo caso se correspondería con el tipo de miembro que hasta ahora hemos denominado campo. También puede definirse como un variable local a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código. Otra posibilidad es definirla como parámetro de un método, que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo puede ser accedida desde código ubicado dentro del método.

### 1.5 Conversiones

Dado que a C# se le asignan tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni tampoco utilizar para almacenar valores de otro tipo, a menos que dicho tipo pueda convertirse en el tipo de la variable. Por ejemplo, no existe conversión de un entero a una cadena arbitraria cualquiera. Por lo tanto, después de declarar i como entero, no puede asignarle la cadena "Hello", como se muestra en el código siguiente.

int i;

i = "Hello"; // Error: "Cannot implicitly convert type 'string' to 'int'"

Sin embargo, en ocasiones puede que sea necesario copiar un valor en un parámetro de método o variable de otro tipo. Por ejemplo, puede que tenga una variable de tipo entero que deba pasar a un método cuyo parámetro es de tipo double. Estos tipos de operaciones se denominan conversiones de tipos. En C#, puede realizar los siguientes tipos de conversiones:

### Conversiones implícitas

No se requiere una sintaxis especial porque la conversión se realiza con seguridad de tipos y no se perderán datos. Entre los ejemplos se incluyen las conversiones de tipos enteros de menor a mayor y las conversiones de clases derivadas en clases base.

Conversiones explícitas (conversiones de tipos): las conversiones explícitas requieren un operador de conversión. Las variables de origen y destino son compatibles, pero existe el riesgo de perder datos debido a que el tipo de la variable de destino es más pequeño que (o es una clase base de) la variable de origen.

### Conversiones explícitas

Una conversión de tipo es una manera de informar al compilador de forma explícita de que pretende realizar la conversión y que está al tanto de que puede producirse una pérdida de datos. Para realizar una conversión de tipo, especifique entre paréntesis el tipo al que se va a aplicar dicha conversión delante del valor o la variable que se va a convertir. El programa no se compilará sin el operador de conversión de tipo.

class Test

{

void Prueba()

{

double x = 1234.7;

int a;

// Cast double to int.

a = (int)x;

}

}

## Modificadores de acceso

Los modificadores de acceso definen el nivel de acceso que cierto código tiene en los miembros de la clase como métodos y propiedades. Es necesario aplicar el modificador de acceso deseado a cada miembro, de otra forma el tipo de acceso por default es implícito.

**private**:

* Este es el nivel de acceso más restrictivo.
* Es accesible solamente dentro del cuerpo de la clase o estructura en que está declarado.
* Tipos declarados dentro (nested types) del mismo cuerpo de la clase también tienen acceso a los tipos privados.

**public**:

* Este es el nivel de acceso con menos restricciones.
* No hay restricciones para accesar miembros públicos, son tan visibles como la clase.

public class A  
{  
   private int x;

byte y;

public bool z=false;

   void F()  
   {

}

public int G()

{

return 0;

}

}  
 

## 1.7 Propiedades

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Éste código puede usarse para comprobar que no se asignen valores inválidos, para calcular su valor sólo al solicitar su lectura, etc.

    Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase. En la práctica lo que se suele hacer escribir como código a ejecutar cuando se le asigne un valor, código que controle que ese valor sea correcto y que lo almacene en un campo privado si lo es; y como código a ejecutar cuando se lea su valor, código que devuelva el valor almacenado en ese campo público. Así se simula que se tiene un campo público sin los inconvenientes que estos presentan por no poderse controlar el acceso a ellos.

   Para definir una propiedad se usa la siguiente sintaxis:

|  |
| --- |
| <tipoPropiedad> <nombrePropiedad> {  set  {  <códigoEscritura>  }   get  {  <códigoLectura>  } } |

    Una propiedad así definida sería accedida como si de un campo de tipo <tipoPropiedad> se tratase, pero en cada lectura de su valor se ejecutaría el <códigoLectura> y en cada escritura de un valor en ella se ejecutaría <códigoEscritura>

    Al escribir los bloques de código get y set hay que tener en cuenta que dentro del código set se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial del mismo tipo de dato que la propiedad llamado value (luego nosotros no podemos definir uno con ese nombre en <códigoEscritura>); y que dentro del código get se ha de devolver siempre un objeto del tipo de dato de la propiedad.

    En realidad el orden en que aparezcan los bloques de código set y get es irrelevante. Además, es posible definir propiedades que sólo tengan el bloque get (propiedades de sólo lectura) o que sólo tengan el bloque set (propiedades de sólo escritura) Lo que no es válido es definir propiedades que no incluyan ninguno de los dos bloques.

Sintaxis para declarar variables.

# <tipoDato> <nombre> [=<valor inicial>] [,<nombre> [=<valor inicial>]… ];

# UNIDAD 4. Estructuras de condición

En una estructura de condición/decisión, el programa al ser ejecutado toma una decisión, ejecutar o no ciertas instrucciones si se cumplen o no ciertas condiciones. Las condiciones devuelven un valor, verdadero o falso, determinando así la secuencia a seguir.

Por ejemplo, si el programa pide una clave de acceso, deberá continuar con la ejecución normal en caso de que la clave introducida por el usuario sea correcta, y deberá salir del mismo en caso contrario.

### Estructura IF

Si (hoy es sabado)

{

Me levanto tarde;

}

En programación se evalúa a verdadero o falso la condición, que es lo que está dentro de los paréntesis. Si el resultado de evaluar esta condición es true (verdadero) se ejecutan las líneas del bloque, y si se evalúa a false (falso) no se ejecutan.

if (num==10)

{

string s= "El número es igual a 10";

}

En el ejemplo anterior se evalúa como verdadero o falso lo que está dentro de los paréntesis, es decir, num==10. Por lo tanto, el operador == retornará true siempre el valor que contenga la variable num sea 10, y false si es otra cosa.

No se debe confundir el operador de comparación **==** con el de asignación **=.**

Como puedes apreciar, la instrucción *if* ejecuta el código de su bloque siempre que la expresión que se evalúa retorne true. Sin embargo, no es necesario abrir el bloque en el caso de que solamente haya que ejecutar una sentencia. Podríamos haber escrito el ejemplo de esta otra forma:

if (num==10)

string s= "El número es igual a 10";

No es necesario utilizar los caracteres **{** y **}** para definir el bloque condicional del código. La regla es que si un bloque if contiene solo una línea de código, las llaves no son requeridas.

### Instrucción If … Else

También puede ocurrir que tengamos que realizar una serie de acciones si se da una condición y otras acciones en caso de que esa condición no se dé. Pues bien, para eso tenemos la instrucción *else*.

Si (hoy es sábado o es domingo)

{

Me levanto tarde;

}

De lo contrario

{

Me levanto temprano;

Voy a la escuela;

}

Si se cumple la condición, se realiza el código del bloque if (si), y si no se cumple se realiza el código del bloque else (de lo contrario). Si el bloque consta de una única instrucción, podemos ahorrarnos las llaves.

if ( x > 9 )

s = "x es mayor que 9";

else

s = "x es menor o igual que 9";

### Instrucción If … Else If

Hasta ahora hemos visto instrucciones if que realizan instrucciones basadas en las decisiones de una sola condición. Algunas veces es necesario realizar decisiones basadas en varios criterios diferentes. Para este propósito se puede usar la instrucción if… else if…

Veamos un ejemplo, si compras el libro te regalo el separador, de lo contrario, si compras la pluma te regalo el cargador, de lo contrario, si compras el cuaderno te regalo un llavero, y, de lo contario, no te regalo nada.

Si (compras el libro)

{

    te regalo el separador;

}

de lo contrario si (compras la pluma)

{

    te regalo el cargador;

}

de lo contrario si (compras el cuaderno)

{

    te regalo un llavero;

}

de lo contrario

{

    no te regalo nada;

}

Si se cumple alguna de las condiciones se producen una serie de consecuencias, y si no se cumple ninguna de las tres condiciones la consecuencia es que no hay regalo alguno.

if (num==10)

{

Mensaje = "El número es igual a 10";

}

else if (num>5)

{

Mensaje = "El número es mayor que 5";

}

else if (num>15)

{

Mensaje = "El número es mayor que 15";

}

else

{

Mensaje = "El número no es 10 ni mayor que 5";

}

Si la variable *num* tiene asignado un valor de 10, se ejecutará el bloque del primer if, por lo que la variable *Mensaje* contendrá "El número es igual a 10". Sin embargo, a pesar de que 10 es también mayor que cinco, no se ejecutará el bloque del primer else if, pues ni siquiera se llega a comprobar dado que ya se ha cumplido una condición en la estructura. Si *num* vale un número mayor que 5, menor que 15 y distinto de 10, o sea, 9, por ejemplo, se ejecuta el bloque del primer else if siendo "el número es mayor que 5" el valor de *Mensaje*. Ahora bien: ¿qué sucede si el número es mayor que 15? Pues sucede exactamente lo mismo, ya que, si es mayor que 15 también es mayor que 5, de modo que se ejecuta el bloque del primer else if y después se dejan de comprobar el resto de las condiciones. Por lo tanto, en este ejemplo, el bloque del segundo else if no se ejecutaría en ningún caso. Por último, el bloque del else se ejecutará siempre que *num* valga 5 o menos de 5, pues es el único caso en el que no se cumple ninguna de las condiciones anteriores.

Las condiciones que se evalúen en un if o en un else if no tienen por qué ser tan sencillas. Recuerda que estas condiciones han de retornar true o false necesariamente, por lo que podemos usar y combinar todo aquello que pueda retornar true o false, como variables de tipo bool, métodos o propiedades que retornen un tipo bool, condiciones simples o compuestas mediante los operadores lógicos && (AND lógico) || (OR lógico) y ! (NOT lógico), o incluso mezclar unas con otras.

### Estructura SWITCH

Una instrucción switch funciona de un modo muy similar a una construcción con if...else if... else. Sin embargo, hay un diferencia que es fundamental: *mientras en las construcciones if...else if... else las condiciones pueden ser distintas para cado uno de los if o else if, en una instrucción switch se evalúa siempre la misma expresión, comprobando los posibles resultados que esta pueda regresar*.

Un switch equivale a comprobar las diferentes situaciones que se pueden dar con respecto a una misma cosa.

Por ejemplo, si te compras un coche y tienes tres tipos de financiamiento: En caso de usar la primera opción te descuento un 10 por ciento, en caso de usar la segunda opción te descuento un cinco por ciento, en caso de usar la tercera opción te descuento un dos por ciento, y en caso de no querer financiamiento no te descuento nada. En este ejemplo se comprueba siempre el valor de un solo elemento, que en este caso sería el tipo de financiamiento.

comprobemos (tipo de financiamiento)

{

    en caso de 1:

        te descuento un 10%;

        Nada más;

    en caso de 2:

        te descuento un 5%;

        Nada más;

    en caso de 3:

        te descuento un 2%;

        Nada más;

    en otro caso:

        no te descuento nada;

        Nada más;

}

En el ejemplo anterior se comprueba el valor del “tipo de financiamento”; si vale 1 sucede el primer caso, si vale 2 el segundo caso, si vale 3 el tercer caso, y si vale cualquier otra cosa sucede el último caso.

switch (tipo\_financiamiento)

{

    case 1:

        descuento=10;

        break;

    case 2:

        descuento=5;

        break;

    case 3:

        descuento=2;

        break;

    default:

        descuento=0;

        break;

}

Solamente se usa un bloque para la instrucción *switch* completa (ninguno de los *case* ni los *default* abre ningún bloque). Una vez que se terminan las instrucciones para cada caso hay que poner la instrucción *break* para salir del *switch.*

Si queremos que el programa haga las mismas cosas en distintos casos, habrá que poner todos estos casos y no cerrarlos con *break*. Por ejemplo, si el descuento del 5% aplica tanto para la segunda como para la tercera opción de financiamiento:

switch (tipo\_financiamiento)

{

    case 1:

        descuento=10;

        break;

    case 2:

    case 3:

        descuento=5;

        break;

    default:

        descuento=0;

        break;

}

Así, en caso de que *tipo*\_*financiamiento* valiera 2 ó tres, el descuento sería del 5%, pues, si *tipo*\_*financiamiento* vale 2, el programa entraría por “case 2” y continuaría por “case 3” ejecutando el código de este al no haber cerrado el case 2 con break, y si *tipo*\_*financiamiento* vale 3 entraría por “case 3” ejecutando, por lo tanto, el mismo código.

Sin embargo, no es válido establecer acciones para el “case 2” sin cerrarlo con break. En este caso ocurriría un error diciendo que hay que cerrar “case 2”.

La instrucción default no es obligatoria y no tiene porqué aparecer al final si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

La expresión que se ha de comprobar en un *switch* ha de ser de tipos sbyte, byte, short, ushort, int, uint, long, ulong, char o string.

# Unidad V. Estructuras repetitivas

## Tema 2.1

## Tema 2.2

## Tema 2.3

### Tema 2.3.1

# Unidad II. Nombre de la unidad

## Tema 2.1

## Tema 2.2

## Tema 2.3

### Tema 2.3.1

# Anexos

# Bibliografía