PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC 1102 INTRODUCCIÓN A LA PROGRAMACIÓN
Profesor: Rodrigo Sandoval U.

Capítulo II – Introducción al Lenguaje C#

1 EL LENGUAJE C# Y EL FRAMEWORK DE .NET 1

1.1 MICROSOFT .NET FRAMEWORK 1.1 1
1.1.1 Lenguajes de Programación 1
1.1.2 Common Language Runtime (CLR) 2
1.1.3 Biblioteca de clases de .Net 2
1.2 ESTRUCTURA DE UN PROGRAMA EN C# 2
1.2.1 Ejemplo de un programa en C# 2
1.3 IDENTIFICADORES 3
1.3.1 Algunas consideraciones al definir identificadores 4
1.3.2 Ejemplos de identificadores 4
1.4 DECLARACIÓN DE CLASES Y SUS ELEMENTOS 6
1.4.1 Visibilidad/accesibilidad de una clases y sus miembros 6
1.4.2 Elementos de una Clase 7
1.4.3 Campos 7
1.4.4 Tipos de Datos 8
1.4.5 Métodos o Funciones 11
1.4.6 Sintaxis de la llamada a las Funciones definidas 12
1.4.7 La función principal Main() y los programas en C# 13
1.4.8 Ejemplo en C# de la simulación de compra de un boleto en el metro. 13
1.5 EXPRESIONES 15
1.5.1 Expresiones Aritméticas 15
1.5.2 Operadores Compuestos 17
1.5.3 Expresiones Relacionales, Lógicas o Booleanas 18
1.5.4 Precedencia de todos los operadores 19
1.6 INSTRUCCIONES 20
1.7 OPERADOR DE ASIGNACIÓN 20
1.8 CONVERSIÓN DE TIPOS (*TYPE CASTING*) 22
1.9 ENTRADA Y SALIDA 23
1.9.1 Salida con Formato: Console.WriteLine 23
1.9.2 Entrada con Formato: Console.ReadLine y Console.Read 26
1.10 COMENTARIOS, INDENTACIÓN Y OTROS ASPECTOS VISUALES 27
1.10.1 Comentarios 27
1.10.2 Indentación (indexación) y otros aspectos visuales 27
1.10.3 Algunos consejos 28

Material preparado por Rodrigo Sandoval U en Marzo-Agosto 2004
Basado en la documentación del curso IIC1102, año 2003, de Miguel Nussbaum y Marcos Sepúlveda et. al.

1 El Lenguaje C# y el Framework de .NET

C# (pronunciado “C Sharp”) es el nuevo lenguaje de propósito general orientado a objetos creado por Microsoft para su nueva plataforma .NET.

Microsoft.NET es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando estos últimos años con el objetivo de mejorar tanto su sistema operativo como su arquitectura de desarrollo anterior, para obtener una plataforma con la que sea sencilla la construcción de software.

La plataforma .NET ofrece numerosos servicios a las aplicaciones que para ella se escriban, como son un recolección de basura, independencia de la plataforma, total integración entre lenguajes (por ejemplo, es posible escribir una clase en C# que derive de otra escrita en Visual Basic.NET que a su vez derive de otra escrita en Cobol)

Como se deduce del párrafo anterior, es posible programar la plataforma .NET en prácticamente cualquier lenguaje, pero Microsoft ha decidido sacar uno nuevo porque ha visto conveniente poder disponer de un lenguaje diseñado desde 0 con vistas a ser utilizado en .NET, un lenguaje que no cuente con elementos heredados de versiones anteriores e innecesarios en esta plataforma y que por tanto sea lo más sencillo posible para programarla aprovechando toda su potencia y versatilidad.

C# combina los mejores elementos de múltiples lenguajes de amplia difusión como C++, Java, Visual Basic o Delphi. De hecho, su creador Anders Heljsberg fue también el creador de muchos otros lenguajes y entornos como Turbo Pascal, Delphi o Visual J++. La idea principal detrás del lenguaje es combinar la potencia de lenguajes como C++ con la sencillez de lenguajes como Visual Basic, y que además la migración a este lenguaje por los porgramadores de C/C++/Java sea lo más inmediata posible.

Además de C#, Microsoft propociona Visual Studio.NET, la nueva versión de su entorno de desarrollo adaptada a la plataforma .NET y que ofrece una interfaz común para trabajar de manera cómoda y visual con cualquiera de los lenguajes de la plataforma .NET (por defecto, C++, C#, Visual Basic.NET y JScript.NET, aunque pueden añadirse nuevos lenguajes mediante los plugins que proporcionen sus fabricantes).

1.1 Microsoft .NET Framework 1.1

El Framework de .Net es una infraestructura sobre la que se reúne todo un conjunto de lenguajes y servicios que simplifican enormemente el desarrollo de aplicaciones. Mediante esta herramienta se ofrece un entorno de ejecución altamente distribuido, que permite crear aplicaciones robustas y escalables. Los principales componentes de este entorno son:

- Lenguajes de compilación
- Biblioteca de clases de .Net
- CLR (Common Language Runtime)

Actualmente, el Framework de .Net es una plataforma no incluida en los diferentes sistemas operativos distribuidos por Microsoft, por lo que es necesaria su instalación previa a la ejecución de programas creados mediante .Net. El Framework se puede descargar gratuitamente desde la web oficial de Microsoft (ver link de descarga en los recursos del final).

1.1.1 Lenguajes de Programación

.Net Framework soporta múltiples lenguajes de programación y aunque cada lenguaje tiene sus características propias, es posible desarrollar cualquier tipo de aplicación con cualquiera de estos lenguajes. Existen más de 30 lenguajes adaptados a .Net, desde los más conocidos como C# (C Sharp), Visual Basic o C++ hasta otros lenguajes menos conocidos en el mundo microsoft como son Perl o Cobol.

1.1.2 Common Language Runtime (CLR)

El CLR es el verdadero núcleo del Framework de .Net, ya que es el entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios que ofrece el sistema operativo estándar Win32. La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .Net en un mismo código, denominado código intermedio (MSIL, Microsoft Intermediate Language). Para generar dicho código el compilador se basa en el Common Language Specification (CLS) que determina las reglas necesarias para crear código MSIL compatible con el CLR.

1.1.3 Biblioteca de clases de .Net

Cuando se está programando una aplicación muchas veces se necesitan realizar acciones como manipulación de archivos, acceso a datos, conocer el estado del sistema, implementar seguridad, etc. El Framework organiza toda la funcionalidad del sistema operativo en un espacio de nombres jerárquico de forma que a la hora de programar resulta bastante sencillo encontrar lo que se necesita.

Para ello, el Framework posee un sistema de tipos universal, denominado Common Type System (CTS). Este sistema permite que el programador pueda interactuar los tipos que se incluyen en el propio Framework (biblioteca de clases de .Net) con los creados por él mismo (clases). De esta forma se aprovechan las ventajas propias de la programación orientada a objetos, como la herencia de clases predefinidas para crear nuevas clases, o el polimorfismo de clases para modificar o ampliar funcionalidades de clases ya existentes.

Este resumen fue extraído de: http://www.desarrolloweb.com/articulos/1328.php?manual=48

1.2 Estructura de un programa en C#

La declaración de un programa en C# se centra en la definición de clases que estarán involucradas en la lógica de la solución. Este programa puede contener cuantas clases se estime necesarias, y al menos una de ellas debe contener el algoritmo principal o Main().

En pocas palabras, la estructura general y simplificada de un programa en C# es la siguiente: se declaran las librerías cuyas clases se referencian en el programa, luego se declaran las clases requeridas con todos sus detalles, y finalmente, la clase principal (que en muchos casos es la más simple y de corta declaración), cuyo principal y a veces único contenido es el método Main(), también conocido como algoritmo principal. Este Main() es obligatorio en todo programa, y será llamado en forma implícita cuando la ejecución del programa dé inicio.

```
[declaración de namespaces o librerías de clases (externas) requeridas]


[declaración clase 1]
[declaración clase 2]
...
[declaración clase N o principal]
      [algoritmo principal o Main()]
```

En la declaración de un programa, todos los elementos definidos se etiquetan con nombres propios, que se conocen como identificadores. Estos identificadores son asignados arbitrariamente por el programador, siempre cumpliendo algunas reglas que se describen a continuación.

1.2.1 Ejemplo de un programa en C#

El programa más simple de todos: “Hola Mundo”, el cual sólo contiene una clase principal.

```
using System;  // Referencia al namespace de clases y métodos más usado

// Se declara una clase única y exclusivamente para alojar el Main().
class MainApp {

   // Función principal, Main()
   public static void Main() {

      // Escribe el texto a la consola.
      Console.WriteLine("Hola Mundo, estoy hecho en C#!");
      Console.ReadLine(); // Espera un ENTER para cerrar
   }
}
```

Como se ve en el trozo de código anterior, todo en C# pertenece a una clase, y por ende, obliga totalmente a tener un enfoque orientado a estos objetos.

1.3 Identificadores

Los identificadores son los **nombres con que se identifica a los distintos objetos dentro de un programa**, como ser:

- Clases.
- Instancias.
- Namespaces.
- Funciones o métodos.
- Variables, campos o propiedades.
- Constantes.
- Estructuras de datos.

En C#, los **identificadores** siguen las reglas para identificadores recomendados en el Anexo 15 del Unicode Standard, es decir, cada identificador puede contener:

- Letras (las del alfabeto mayúsculas y minúsculas, **menos la ñ, Ñ y las acentuadas**). No es posible emplear acentos o caracteres especiales del alfabeto español.
- Dígitos numéricos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- Símbolo de subrayado (_)
- Adicionalmente, en casos especiales, se puede utilizar el símbolo @ al comienzo de un identificador, cuando éste es una de las palabras claves. Sin ese símbolo, ningún identificador puede ser igual al conjunto de palabras reservadas del lenguaje C#.
- Nótese, que a diferencia de otros lenguajes, C# así como sus antecesores (C, Java, etc.) son “sensibles a las mayúsculas”, es decir, se considera “var1” y “VAR1”, e incluso “Var1”, son identificadores totalmente distintos.

Las palabras reservadas en C# son las siguientes.

abstract	**event**	**New**	**struct**
as	**explicit**	**Null**	**switch**
base	**extern**	**object**	**this**
bool	**false**	**operator**	**throw**
break	**finally**	**out**	**true**
byte	**fixed**	**override**	**try**
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**	

1.3.1 Algunas consideraciones al definir identificadores

En la práctica, **la letra de subrayado (_) se emplea para dar mayor legibilidad a nombres** compuestos por varias palabras. Además, se acostumbra emplear letras minúsculas para nombrar a las variables y las mayúsculas se usan al comienzo de una palabra en funciones declaradas con identificadores compuestos.

Se recomienda **elegir nombres que sean representativos del elemento (atributo o método)** que se defina o del valor que la variable guarde. Por ejemplo, una variable que guardará la edad de un empleado podría llamarse edad_empleado o *edad*, y no simplemente xyz, aunque éste es un identificador válido y el programa funcionará correctamente este nombre.

1.3.2 Ejemplos de identificadores

Los siguientes son todos identificadores válidos en C#.

```
puntero
nombre_empleado
area
TAMANO_STRING
EdadEmpleado
potencia10
i
For
iF
While
Nombre
nombre
```

Los siguientes son identificadores inválidos. Entre paréntesis se indica por qué.

```
nombre-empleado (no se puede usar el guión, sólo el subrayado)
teléfono (no se pueden usar caracteres especiales del Español)
while (ésta es una palabra reservada del lenguaje C#)
```

1.4 Declaración de Clases y sus Elementos

Todos los elementos en cualquier programa en C# pertenecen a una clase. Por ello, lo primero que siempre se debe hacer es declarar cada una de las clases que se requerirán. Eventualmente es factible declarar un programa con una única clase, que contendrá su función principal.

La sintaxis para la declaración de una clase tiene la siguiente forma:

```
[modificadores] class <classname> [: nombreclasebase]
{
      [cuerpo de la clase]
}[;]
```

Ejemplo básico:

```
class persona
{
      private string nombre;
      private int edad;

      public void ImprimirNombre()
      {
            Console.WriteLine("Nombre: {0}", nombre);
      }
}
```

La línea base de la declaración de una clase describe el nombre, los modificadores y atributos, y la clase desde la cual se está heredando. Aquí se identifican como opcionales todos aquellos elementos entre corchetes. Los modificadores pueden ser: public, private, o nada. El nombre de la super clase o clase base, es cualquier nombre que se haya asignado a otra clase.

1.4.1 Visibilidad/accesibilidad de una clases y sus miembros

Cada clase puede pertenecer a un grupo de clases llamado Namespace. A su vez, cada clase se compone de elementos, como atributos y métodos, que forman parte del cuerpo de su declaración y que se revisan en la siguiente sección.

Cada uno de éstos, ya sea la clase o bien los atributos o métodos, pueden ser acotados en su visibilidad o accesibilidad, lo cual limita su uso desde ámbitos fuera de la clase o del namespace. Dicho de otra manera, si una clase contiene elementos que son privados, éstos son sólo visibles para los otros elementos dentro de la misma clase, pero no para otros objetos de otras clases. Por el contrario, si la clase contiene elementos públicos, éstos pueden ser visibles desde instancias de otras clases.

Para determinar la visibilidad o accesibilidad que se tendrá a un elemento en particular, se antepone el término: **private** o **public**, para explicitar su condición. En el caso de las clases, al no explicitar nada, se asume que son public. Por el contrario, en el caso de los elementos de una clase, como atributos o métodos, al no explicitar nada, se asumen como private. Por lo tanto, en la declaración del ejemplo básico de clase anterior, la clase "persona", es pública en su ámbito; sus dos atributos, "nombre" y "edad" son explícitamente declarados como privados, y finalmente el método "ImprimirNombre" es implícitamente privado, al no haberse declarado expresamente su accesibilidad.

1.4.2 Elementos de una Clase

El cuerpo de la declaración de la clase se compone de los miembros de ésta. Entre los miembros se encuentran campos, propiedades, métodos, constantes, operadores, entre otros varios elementos.

Campos: son aquellos atributos de la clase, definidos por un tipo de dato y un identificador. En el ejemplo básico anterior, "nombre" es un campo de tipo string (cadena de texto), que es privado a la clase.

Métodos: secuencias de instrucciones que actúan sobre los datos del objeto, o valores de los campos.

Propiedades: se les conoce también como "campos inteligentes", ya que son métodos que se ven como campos de la clase. Esto permite que al utilizar una instancia de esta clase se logre un mayor nivel de abstracción, ya que no se requiere saber si se accesa un campo directamente o se está ejecutando un método de acceso.

Constantes: tal como lo sugiere su nombre, se trata de campos que no pueden ser modificados durante la ejecución del programa. El valor de la constante se declara junto con la definición de la clase, por lo que es un valor que debe ser conocido previamente a la ejecución del programa. Una alternativa para definir campos de valor fijo es usar el modificador *readonly*, que permite inicializar el valor de un campo en el constructor de la clase, para luego quedar fijo para cualquier efecto posterior.

Estos últimos dos tipos de elementos se verán en detalle en el capítulo IV, Programación OO con C#.

1.4.3 Campos

Para poder resolver un problema empleando un computador 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.

En C#, las variables reciben nombres que respetan las reglas de los identificadores. Además, **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.

Como se verá más adelante, **toda variable debe ser declarada previamente a su uso dentro del programa**, y esto debe hacerse respetando una sintaxis establecida.

En resumen: **Una variable es un objeto que almacena un dato, es asociado a un identificador y a un tipo.**

La forma genérica es la siguiente:

```
<modificador> <tipo> <identificador> [= <valorInicial>] ;
```

Donde **<tipo>** puede ser: **int, float, char,** entre otras posibilidades. **[]** significa opcional. **<valor_inicial>** puede corresponder a una constante del tipo entero, real, caracter, arreglo, etc. **<modificador>** puede ser una de las siguientes posibilidades: public, protected, private, static, const, readonly; que corresponden a descripciones del comportamiento de esta variable en relación al resto de los elementos del programa.

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";
```

Ejemplos de declaraciones de variables con modificadores:

```
public int numero  = 10;  // Visible para otros objetos
private char sigla = 'G';  // Invisible para otros objetos
const string nombre = "Juan";  // Valor fijo, constante
```

Los modificadores private, public, protected definen la visibilidad que tendrán otros objetos de tener acceso a dichos campos. Por otro lado, const, readonly, se refieren a la factibilidad de poder modificar el valor del campo una vez inicializado o definido. Finalmente, static define el comportamiento de todas las instancias de la misma clase, y el tratamiento que puedan tener sobre el campo en cuestión.

1.4.4 Tipos de Datos

Cada uno de los distintos objetos que son manipulados dentro de un programa tiene asociado un **tipo de datos**. Desde el punto de vista de almacenamiento o para llevar a cabo operaciones sobre estos objetos, es necesario tratarlos de distinta manera. Así, la mayoría de los lenguajes de programación separan aquellos objetos que son caracteres de los que son números enteros y números con decimales.

Para la plataforma .NET, los tipos de datos básicos son los siguientes, varios de los cuales sirven para representar cantidades numéricas, y otros para representar texto y otros.

Categoría	Nombre de la Clase	Descripción	Tipo de Dato C#
Número Entero	Byte	Entero sin signo (de 8 bits). Valores de 0 a 256.	byte
	SByte	Entero (de 8 bits). Valores de -127 a 128.	sbyte
	Int16	Entero (de 16 bits). Valores de -32767 a 32768.	short
	Int32	Entero de 32 bits	int
	Int64	Entero de 64 bits	long
	UInt16	Entero sin signo de 16 bits..	ushort
	UInt32	Entero sin signo de 32 bits.	uint
	UInt64	Entero sin signo de 64 bits.	ulong
Números reales (con decimales)	Single	Número con decimales de precisión simple (32 bits).	float

	Double	Número con decimales con precisión doble (64 bits)	double
Lógicos	Boolean	Un valor booleano, verdadero o falso (true/false)	bool
Otros	Char	Un carácter o símbolo Unicote (16 bits)	char
	Decimal	Un valor decimal de 96 bits	decimal
Class objects	Object	La raíz de la jerarquía de objetos.	object
	String	Una cadena de caracteres inmutable, de largo fijo.	string

En el caso del lenguaje de programación C#, **los tipos de datos más empleados** son:

- Los **tipos enteros**: **`int`**, **`short`** y **`long`** para manipular números enteros.
- Los **tipos reales**: **`float`** y **`double`**.
- El tipo: **char** (´a´,´z´,´?´,´$´,´@´,´G´,´7´, etc.; en total los 256 caracteres del código ASCII) para caracteres, letras y símbolos.
- Así como algunos más complejos como los **`strings`** y **`arreglos`**.

Es importante recordar que todo objeto dentro de un programa tiene un tipo de datos asociado, y es necesario tener esta asociación en mente a la hora de manipular, almacenar, operar y hasta desplegar en pantalla dicho objeto.

1.4.4.1 Tipos enteros

Las variables de tipos de datos enteros pueden almacenar únicamente **valores numéricos sin decimales**. Si se intenta almacenar un valor con decimales en una variable entera, se producirá un **truncamiento** del valor.

El tipo entero básico es **`int`**. Existen además las variantes **`short`** que permite almacenar valores más pequeños (con el consiguiente ahorro de espacio en memoria), y el **`long`**, que soporta números más grandes (pero con mayor gasto de memoria).

El valor máximo que se puede almacenar en las variables de cada uno de estos tipos depende de la arquitectura del computador y del compilador que se estén empleando.

Constantes de Tipo Entero

Para declarar valores constantes de tipo entero, hay que reconocer si se trata de enteros de tipo int, o long. Por ejemplo, la inicialización de las siguientes dos variables considera un valor escrito como constante entera, diferenciando que para el int, se usa un número simple y para el long, se agrega una 'L' al final:

```
int n1 = 10;
long n2 = 120390L;
```

1.4.4.2 Tipos reales

Las variables de tipos de datos reales pueden almacenar valores numéricos con decimales.

El tipo real básico es **`float`**. Existe también la variante **`double`**, que permite almacenar valores en **doble precisión**, más grandes y con mayor cantidad de decimales. Sin embargo, el uso de este tipo provoca un mayor gasto de memoria que si se empleara el tipo básico **`float.`** La cantidad de decimales, así como el valor máximo que se puede almacenar en variables de estos tipos depende de la arquitectura del computador y del compilador que se estén empleando.

Constantes de Tipo Real

Al igual que en el caso de los enteros, los valores constantes de tipo real se diferencian en su declaración al identificarse como float o double, aún cuando ambos usan valores decimales en su declaración. El caso de los floats requiere agregar una 'F' al final, tal como se indica en el siguiente ejemplo:

```
float n1 = 10.0F;
double n2 = 120.390;
```

1.4.4.3 Caracteres: Tipo de datos *char*

Las variables de tipos de datos **`char`**, pueden almacenar un caracter del **`código ASCII`** **extendido** (256 caracteres). En realidad, lo que se guarda no es el caracter en sí, sino el código correspondiente. Por lo tanto, puede verse al tipo **`char`** como un subtipo del tipo **`int`**, que puede almacenar enteros entre 0 y 255. De hecho, las expresiones de tipo **`char`** pueden manipularse como enteros.

Las constantes de tipo **`char`** se representan delimitadas por comillas simples. Así, para escribir el caracter que representa la letra A mayúscula escribimos:

```
'A'
```

El hecho de que las expresiones de tipo **`char`** se consideren como enteros permite llevar a cabo **conversión de tipos entre expresiones `char` y expresiones `int`**. Por ejemplo, es posible llevar a cabo las siguientes asignaciones, las cuales son todas equivalentes y almacenan en la variable correspondiente la letra **`A`** mayúscula (código ASCII 65).

```
char ch;
int i;

ch = 65;
i = 'A';

ch = 'A';
i = 65;
```

Además, puede emplearse el símbolo \ para denotar caracteres mediante su **código ASCII**. Con esta notación, el caracter **`'A'`** puede representarse también como **`'\65'`**

Esta notación es muy útil para representar caracteres especiales, que no aparecen en los teclados y no se pueden ver en la pantalla. Este es el caso del **caracter nulo** (ASCII 0): **`'\0'`**, el cual se empleará más adelante para indicar el final de los **`strings`**. Así, las siguientes son representaciones equivalentes en C para el caracter constante que representa la letra A mayúscula:

```
'A'
65
'\65'
```

La preferencia en usar la tercera notación y no la segunda es por simple claridad, para dejar bien claro que se está manipulando un caracter y no un número (aunque para el C es lo mismo).

Otros caracteres especiales que se pueden representar mediante esta notación son:

- **'\n'**: Caracter de nueva línea (a veces usado en **Console.WriteLine**, más adelante)
- **'\t'**: Tabulador (también útil en **Console.WriteLine**)
- **'\\'**: Caracter \ (esta es la única forma de especificarlo)

Más interesante resulta la manera de organizar los caracteres como cadenas de **char**, de manera que puedan manipularse cadenas de texto (**string**). Las cadenas de texto se especifican entre comillas dobles: **"iic1102 sección".**

1.4.5 Métodos o Funciones

Una "función", genéricamente se describe como un **conjunto de instrucciones** (posiblemente acompañadas de algunos datos temporales), que llevan a cabo una subtarea dentro de un programa más complejo. Un método es una función que pertenece a una clase en particular.

Por ejemplo, dentro de un programa que convierte una lista de temperaturas expresadas en grados Fahrenheit, a grados Celsius, podríamos definir una función que haga una de las conversiones, y la emplearíamos varias veces para lograr completar nuestra tarea.

Tenemos entonces que **las funciones dentro de un programa cumplen con subtareas**, para las cuales **es necesario suministrarles algunos datos de entrada, y luego de completadas, recibiremos como resultado algunos datos de salida**. Para el caso de una función que calcula el área de un círculo, la entrada que requiere la función sería el radio del círculo, y la salida que arrojaría luego del cómputo sería el valor para el área. Este hecho se ilustra en el diagrama que se muestra abajo, en el que podemos ver la tarea de transformación de datos que cumple una función: **La alimentamos con una entrada y recibimos una salida en respuesta**.

Por otra parte, para especificar el proceso de transformación que debe llevar a cabo la función, es necesario emplear las instrucciones del lenguaje de programación. Es necesario respetar algunas pautas, tanto para la especificación de los parámetros de entrada, como del valor de retorno, pero la estructura general es la misma.

1.4.5.1 Sintaxis de la definición de Funciones o Métodos

La sintaxis general de definición de métodos es la siguiente:

```
<accesibilidad> <tipo_retorno> <nombre_función> ( <lista_parámetros> ) {
     <declaración_de_variables_locales> ;
     <cuerpo_de_la_función> ;
     return (<valor_retorno>) ;
 }
```

<accesibilidad> se refiere a la visibilidad que tienen otras partes de un programa para ejecutar el método. Las posibilidades son public, private, protected, lo cual define la accesibilidad que se tiene a este método desde otras clases.

<tipo_retorno> es el tipo del valor que devolverá la función. **Es posible definir funciones que no devuelven ningún valor** (para por ejemplo definir funciones que solo imprimen valores o ciertos mensajes en pantalla, para los que no es necesario que devuelva valor alguno). Para definirlos o declararlos, se utiliza como tipo de retorno la palabra reservada **void.** A estas funciones se las conoce también como **procedimientos**, porque no devuelven ningún valor.

<nombre_función> debe ser sustituido por un identificador que corresponde al nombre con que se invocará la función posteriormente (o llamada de la función). Debe ser un identificador válido. El **valor de retorno de la función es asociado al nombre de la función**.

<lista_parámetros> corresponde con una lista de variables denotadas por identificadores asociados a un tipo, cada uno representa un parámetro de entrada. La lista de parámetros puede ser vacía pero **los paréntesis deben aparecer**.

La <declaración_de_variables_locales> se refiere a **variables locales** que podrán ser utilizadas únicamente dentro de la función. Fuera de ella, no serán reconocidas.

El <cuerpo_de_la_función> es una lista de instrucciones, separadas por punto y coma.

<valor_retorno>, debe ser una expresión (constante, aritmética, lógica del tipo que se especificó como <tipo_retorno>, y establece la expresión que retornará la función una vez que haya terminado.

1.4.5.2 Ejemplos: Definición de Funciones

- Función que convierte temperaturas
- Función que calcula el área de un círculo

```
// Función que recibe una temperatura expresada en grados Fahrenheit y
// retorna su equivalente en grados Celsius. El argumento de entrada es
// farenheit de tipo int. El valor de retorno también es de tipo int.
// La fórmula de conversión es C=(F-32)*(5/9). Puede ocurrir truncamiento.
      int celsius(int fahrenheit) {
         return((fahrenheit-32.0)*(5.0/9.0));
      }

// Función que recibe una temperatura expresada en grados Celsius y retorna
// su equivalente en grados Faherenheit. El argumento de entrada es celsius
// de tipo int. El valor de retorno también es de tipo int.
// La fórmula de conversión es F=32+(C*9/5). Puede ocurrir truncamiento.
      int fahrenheit(int celsius) {
         return((celsius*9.0/5.0)+32.0);
      }


// Función que calcula el área de un círculo dado su radio. Tanto el
// argumento de entrada (radio) como el valor de retorno es de tipo
// float. La fórmula implementada es A=Pi*R*R.
      float area(float radio) {
         return(3.14 * radio * radio);
      }
```

1.4.6 Sintaxis de la llamada a las Funciones definidas

La sintaxis general de las llamadas a las funciones definidas es la siguiente:

```
<variable> = <nombre_función> ( <lista_parámetros> ) ;
```

<variable> corresponde a una variable denotada por un identificador del mismo tipo del valor que devolverá la función.

<nombre_función> es el identificador que corresponde al nombre con que se invoca a la función.

<lista_paramétros> debe corresponder a una lista de variables denotadas por identificadores asociadas a un tipo, o expresiones de un tipo. Ya sean variables o expresiones, éstas deben corresponder en cuanto a cantidad, posición y tipos a las variables que se definieron como parámetros de entrada en la definición de la función. Representan los valores que se entregan como entrada a la función. Si en la definición de la función no se usó parámetro alguno, entonces en la llamada tampoco debe escribirse ninguna expresión o variable, pero **los paréntesis deben aparecer**.

```
// Llamada a la Función que recibe una temp. expresada en grados Fahrenheit y
// retorna su equivalente en grados Celsius. El argumento de entrada es
// fahrenheit de tipo int. El valor de retorno también es de tipo int.
// La fórmula de conversion es C=(F-32)*(5/9). Puede ocurrir truncamiento.
      int grados_celsius ;
      grados_celsius = celsius(int fahrenheit) ;

// Llamada a la Función que recibe una temp. expresada en Celsius y retorna
// su equivalente en grados Faherenheit. El argumento de entrada es celsius
// de tipo int. El valor de retorno también es de tipo int.
// La formula de conversión es F=32+(C*9/5). Puede ocurrir truncamiento.
      int grados_farenheit ;
      grados_farenheit = fahrenheit(31) ;

// Llamadas a la función que calcula el área de un círculo dado su radio.
// Tanto los argumento de entrada (radio) como el valor de retorno son
// del tipo float. La fórmula implementada es calculo_area = Pi*R*R.
      float calculo_area ;
      float radio = 5.56 ;

      calculo_area = area(5.56) ;
      calculo_area = area(radio) ;
```

1.4.7 La función principal *Main()* y los programas en C#

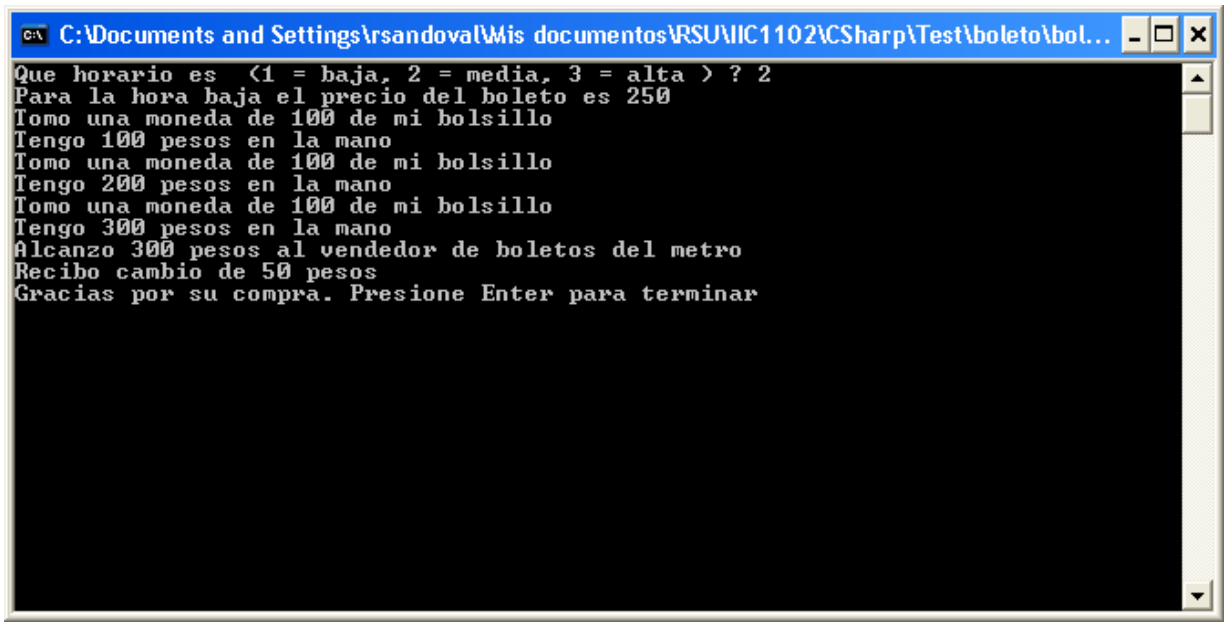
Un programa en C#, está formado por una o más clases, que a su vez cuentan con una o más funciones compuestas de varias instrucciones. Estas funciones que realizan una tarea en particular con los parámetros de entrada, devuelven valores que corresponden a sus resultados, que a su vez son utilizados en otras funciones o métodos.

El computador debe empezar a ejecutar en algún orden estas instrucciones. En el caso de un programa, se comienza por referenciar los namespaces usados, que van en las primeras líneas con la sintaxis `using` *`namespace`*; Luego se registran todas las clases definidas, con sus atributos y métodos, y luego se llega a la función **`Main()`**. Esta función consta también de declaraciones, instrucciones y llamadas a las definiciones de otras funciones e instanciaciones de clase. El orden de ejecución de las instrucciones dentro de la función **`Main()`** es secuencial de arriba a abajo y es así como es posible recién seguir la ejecución del resto de las funciones.

1.4.8 Ejemplo en C# de la simulación de compra de un boleto en el metro.

```
using System;

class Boleteria {

       const int precio1 = 190;
       const int precio2 = 250;
       const int precio3 = 290;

       private int cambio = 0;


       public void ComprarBoleto(int valor, int dinero_en_mano) {
              Console.WriteLine("Alcanzo {0} pesos al vendedor.", dinero_en_mano);
              cambio = dinero_en_mano - valor;
       }
```

```
        public int PreguntarHorario() {
                int hora;
                Console.WriteLine("Que horario es  (1 = baja, 2 = media, 3 = alta ) ? ");
                hora = int.Parse(Console.ReadLine());
                return(hora);
        }

        // PrecioBoleto: indica el valor del boleto, según la hora indicada
        public int PrecioBoleto(int hora) {
                int precio = 0;
                if (hora == 1) precio = 190;
                if (hora == 2) precio = 250;
                if (hora == 3) precio = 290;
                Console.WriteLine("Para la hora baja el precio del boleto es {0} ", precio);
                return(precio);
        }

        public void TomarCambio() {
                Console.WriteLine("Recibo cambio de {0} pesos", cambio);
        }

}

class Bolsillo {

        private int en_la_mano = 0;


        public void SacarMonedas(int moneda) {
                Console.WriteLine("Tomo una moneda de {0} de mi bolsillo ", moneda);
                en_la_mano = en_la_mano + moneda;
        }

        public int EnLaMano() {
                return (en_la_mano);
        }
}



class MainApp {

        static void Main() {

                Boleteria boleteria = new Boleteria();
                Bolsillo bolsillo = new Bolsillo();
                int hora;
                int valor;

                hora  = boleteria.PreguntarHorario();
                valor = boleteria.PrecioBoleto(hora);

                // Se sacan las monedas ($300) del bolsillo
                bolsillo.SacarMonedas(100);
                Console.WriteLine("Tengo {0} pesos en la mano ", bolsillo.EnLaMano());
                bolsillo.SacarMonedas(100);
                Console.WriteLine("Tengo {0} pesos en la mano ", bolsillo.EnLaMano());
                bolsillo.SacarMonedas(100);
                Console.WriteLine("Tengo {0} pesos en la mano ", bolsillo.EnLaMano());

                boleteria.ComprarBoleto(valor, bolsillo.EnLaMano());

                boleteria.TomarCambio();

                Console.WriteLine("Gracias por su compra. Presione Enter para terminar");
                Console.ReadLine();
        }
}
```



1.5 Expresiones

Se conocen como expresiones los valores que son manipulados dentro de un programa. Estos valores **pueden surgir como resultado de efectuar operaciones, como valores de retorno de las funciones, al extraerlos de las variables, o simplemente pueden ser constantes expresadas directamente dentro del código del programa**.

Las expresiones, al igual que el resto de las entidades dentro de un programa, tienen asociado un tipo de datos. Este tipo indicará la forma en que podremos manipular la expresión, por ejemplo, en cuáles variables podremos almacenar su valor.

Existe un conjunto de operadores para cada tipo de datos, de manera que siempre es posible construir una nueva expresión a partir de una o varias ya existentes. Tenemos, por ejemplo, operadores aritméticos para construir expresiones numéricas, y operadores lógicos para construir expresiones booleanas.

1.5.1 Expresiones Aritméticas

Las expresiones aritméticas son valores numéricos, de tipos enteros o reales. Pueden ser constantes, resultados de operaciones aritméticas como sumas o multiplicaciones, contenido de variables numéricas, o combinaciones de las anteriores unidas mediante operadores aritméticos.

Operadores binarios:

`+`	**Suma**
`-`	**Resta**
`*`	**Multiplicación**
`/`	**División**
`%`	**Módulo (sólo sobre tipos de números enteros)**

Operadores unitarios:

- Signo negativo

++ Incremento

-- Decremento

Precedencia:

++ -- -(unitario)

*** / %**

+ -

Asociatividad:

Por la izquierda

Expresiones

Los operandos pueden ser variables o constantes numéricas. Sin embargo, los operadores unarios ++ y -- fueron pensados para aplicarse a variables y el operador unario - a constantes.

operando operador-binario operando

operador-unario operando

Caso particular:

- **++ -- (notación pre o postfija)**

Ejemplos de expresiones aritméticas:

```
2 + 3 * 2        * tiene precedencia sobre + (Resultado: 8)
(2 + 3) * 2      los paréntesis alteran la precedencia (Resultado: 10)
-2 / x           si x vale cero se producirá un error de ejecución
x++              notación postfija (retorna el valor antes del incremento)
++x              notación prefija (retorna el valor después del incremento)
2 - 1 + 3        asociatividad por la izquierda: 4
15 % 6           operador módulo o residuo (Resultado: 3)
-x               le cambia el signo al contenido de x
3 * 4 % 5        igual precedencia, pero asociativa a izq. (Retorna: 2)
3 * (4 % 5)      los paréntesis alteran la precedencia (Retorna: 12)
1 + 15 % 6       % tiene precedencia mayor (Retorna: 4)
```

El resultado de una operación siempre debe ser almacenado en otra variable, o utilizado en alguna función como parámetro.

Ej:

```
class MainApp {
      static void Main() {
        int a, b, c;
        c = a + b;  // Correcto
        Console.WriteLine("{0}", a + b);  // Correcto
        a + b;   // Error
      }
}
```

1.5.2 Operadores Compuestos

Un operador compuesto es una combinación entre un operador binario, junto con un operador de asignación (=). La sintaxis general es:

```
x op= y;
```

lo cual es totalmente equivalente a

```
x = x op y;
```

Por ejemplo:

```
int a, b;
b = 10; a = 3;
a += b;
Console.WriteLine("{0}", a); // Imprime 13
```

1.5.3 Expresiones Relacionales, Lógicas o Booleanas

Este tipo de valores se emplea para dirigir al programa por un determinado flujo de ejecución dependiendo de la evaluación a *verdadero* o *falso* de una expresión lógica. En C# se conoce el tipo booleano como **bool**.

Operadores binarios:

&& ***AND*** **(hace la conjunción lógica de dos valores bool).**
|| ***OR*** **(hace la disyunción lógica de dos valores bool).**

Operadores unitarios:

! ***NOT*** **lógico**

Operadores relacionales o de comparación:

Este tipo de operadores permite comparar el valor de dos expresiones de cualquier tipo, devolviendo como resultado un valor booleano (entero).

== **igual**

!= **distinto**

< **menor**

<= **menor o igual**

> **mayor**

>= **mayor o igual**

Precedencia:

La precedencia está establecida por el siguiente orden de evaluación. La asociatividad se aplica de izquierda a derecha, para operadores con igual precedencia.

< <= >= >

== !=

&&

||

Expresiones

Los operandos de las expresiones booleanas son valores booleanos, incluyendo aquellos que resulten de la comparación mediante los operadores relacionales mencionados anteriormente. Pueden emplearse paréntesis para dejar clara la precedencia de evaluación.

```
operando operador-lógico-binario operando

operador-lógico-unario operando
```

Ejemplos de expresiones booleanas:

```
x                   Verdadero cuando el contenido de la variable x
                    es true.

!x                  Verdadero cuando el contenido de la variable x
                    es false.

true                Verdadero siempre.

true || x           Verdadero siempre, no importa lo que sea x.

false               Falso siempre.

false && x          Falso siempre, no importa lo que sea x.

(x > 5) && !y       Verdadero cuando el contenido de la variable x
                    es mayor que 5, y el contenido de y es false.

(x==1)              Verdadero si el valor de la variable x es 1.

5 == 6              retorna Falso.
4 > = 2             Retorna Verdadero.
((4 % 2 == 0) || (4 % 3 == 0)) && false
```

1.5.4 Precedencia de todos los operadores

Máxima

```
( )   [ ]   -    -
!   ~   ++   --
*   /    %
+   -
<<
<   <=   >= >
== =
&
|
&&
||
=   +=   -=   /=
```

Mínima

1.6 Instrucciones

Las instrucciones son las órdenes que reconoce un computador mientras se encuentra ejecutando un programa. De hecho, un programa es una **secuencia de instrucciones** que al ejecutarse resuelve algún problema para el usuario.

Los programas están escritos en un lenguaje de programación, el cual define el conjunto de instrucciones válidas que podemos utilizar al escribirlos.

En nuestro caso emplearemos el lenguaje de programación C#, por lo que será necesario que estudiemos las instrucciones que componen dicho lenguaje. A lo largo del curso vamos a ir introduciendo estas instrucciones de manera paulatina, hasta llegar a cubrir un subconjunto bastante importante del lenguaje.

Las instrucciones disponibles pueden clasificarse dentro de alguna de las siguientes categorías:

- **Secuenciales**
  - Ejemplos: declaraciones de variables, asignaciones, llamado a funciones, salida (**`Console.WriteLine`**) y entrada (**`Console.ReadLine`**)
- **Control de flujo (o decisión)**
  - Ejemplos: **`if, switch`**
- **Iteración**
  - Ejemplos: **`for, while, do-while, foreach`**

Sintaxis

La sintaxis de las instrucciones en C# varía bastante dependiendo de cada instrucción. En general, toda instrucción debe ir terminada por un **punto y coma**, de la siguiente manera:

<instruccion> ;

Existen instrucciones complejas que pueden abarcar varias líneas, y que pueden contener otras instrucciones dentro de ellas en forma subordinada. Además, las instrucciones se pueden agrupar en bloques, delimitados por corchetes: { }, de la siguiente manera:

```
{
    <instruccion1> ;
    <instruccion2> ;
    ...
    <instruccionN> ;
}
```

1.7 Operador de Asignación

El operador de asignación (=) permite almacenar datos en las variables. Es un operador binario, que requiere que su primer operando (el del lado izquierdo) sea una variable. El segundo operando (el del lado derecho) puede ser otra variable, un valor constante, o una expresión en general. La sintaxis general de este operador es la siguiente:

<variable> **=** <expresion> ;

En caso de que el valor asignado sea una constante, el resultado de esta operación es que el valor expresado se almacenará en la variable especificada al lado izquierdo.

Si el valor asignado es otra variable, se hará una copia del valor almacenado en la segunda variable y se almacenará en la variable del lado izquierdo. Si se trata de una expresión, ésta se evaluará previamente y su resultado se almacenará en la variable indicada.

En todo caso, es necesario tener en cuenta algunas consideraciones sobre los tipos de las variables y de los valores que se están asignando. Más adelante se tratarán los aspectos relativos a la conversión de tipos.

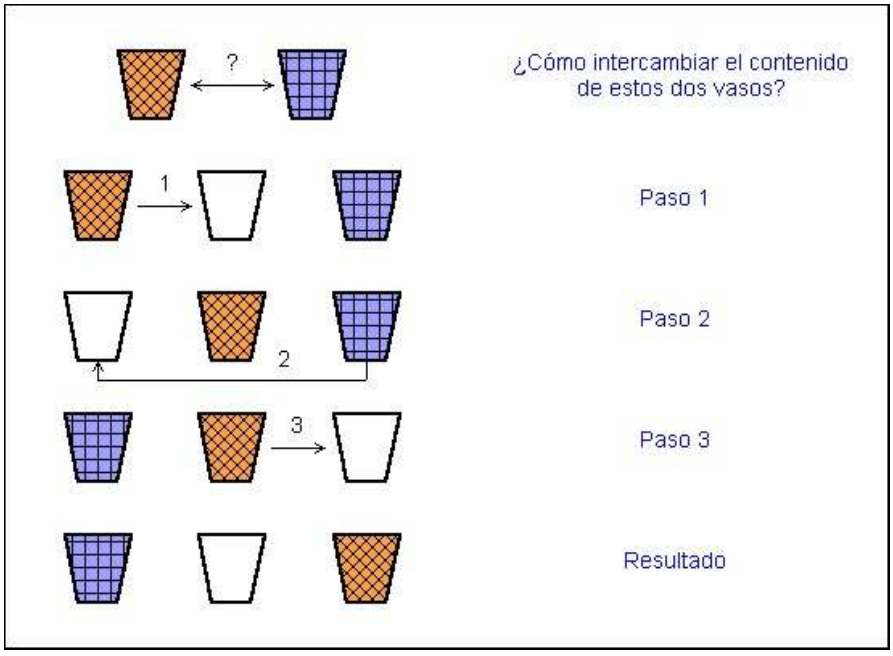
Ejemplos

- **indice = 0;**
  - Se almacenará el valor 0 en la variable i*ndice*
- **suma = a+b;**
  - Se obtendrá la suma de las variables *a* y *b*, y el resultado se almacenará en la variable *suma*
- **NuevaX = ViejaX;**
  - El valor de la variable *ViejaX* se copiará a la variable *NuevaX*
- **j = (i = 9)**
  - Asigna el valor de 9 a las variables *i* y *j*

1.7.1.1 Ejemplo: Intercambio del contenido de dos variables (vasos)

En este ejemplo ilustraremos la problemática de intercambiar el contenido de dos variables. Al igual que cuando nos enfrentamos al problema de intercambiar el contenido de dos vasos, en este caso será necesario emplear un tercer elemento de almacenamiento (variable o vaso), para guardar temporalmente durante el intercambio uno de los contenidos.

El problema del intercambio del contenido de dos variables puede plantearse de manera similar al problema de intercambiar el contenido de dos vasos llenos. La solución en ambos casos es simple: **Se necesita de un tercer elemento de almacenamiento** (vaso o variable) para el almacenamiento temporal del contenido.



```
using System;

class MainApp {

      static void Main() {
        int var1, var2;  // Variables enteras que serán intercambiadas
        int aux;         // Variable auxiliar necesaria para hacer el intercambio
        var1 = 10;       // Asigna valores a las variables
        var2 = 30;

        // Imprime el valor inicial de las variables, antes del intercambio
        Console.WriteLine("var1 contiene {0} y var2 contiene {1}", var1, var2);

        aux = var1;      // Paso 1: El valor inicial de var1 se guarda en aux
        var1 = var2;     // Paso 2: El contenido de var2 se pasa a var1
        var2 = aux;      // Paso 3: El valor inicial de var1 se pasa a var2

        // Imprime el valor final de las variables, luego del intercambio
        Console.WriteLine("var1 contiene {0} y var2 contiene {1}", var1, var2);

        Console.Write("Presione ENTER para terminar."); Console.ReadLine();
      }
}
```

1.8 Conversión de Tipos (*Type Casting*)

El lenguaje C# permite cambiar el tipo de datos de las expresiones en el momento de llevar a cabo las asignaciones. Por ejemplo, si se asigna una expresión entera a una variable de tipo real, el valor de la expresión se convertirá primero al tipo real antes de llevar a cabo la asignación. Este resulta ser el caso más simple de una conversión de tipos, pero existen algunas situaciones en las que se requiere de cierta atención especial:

- De un tipo real a uno entero
    - Se trunca el valor a su parte entera. La parte decimal se pierde.
- Asignaciones entre **`char`** e **`int`**
    - Puesto que en C# se manejan los caracteres mediante su código ASCII (que es un número entero), este tipo de asignaciones funciona en general sin problemas.

La forma de realizar la conversión es requerida en forma explícita por C#. Esto se hace siguiendo la siguiente sintaxis:

```
(<tipo>) <expresion>
```

Ejemplo:

```
int a = 2;
float b;
b = a; // Error: no se puede convertir implícitamente int en float
b = (int) a; //Correcto
```

Ejemplos

- **parte_entera = 7/4;**
    - Como todos los valores involucrados son int, a la variable *parte_entera* se le asignará el

resultado de la división pero truncado: 1

- **(int)(7.0/4.0)**
  - El resultado de esta expresión también es 1
- **7.0/4.0**
  - El resultado de la división será exacto, con todos los decimales
- **7.0/4**
  - Se convierte el 4 a 4.0 y la división arroja decimales
- **x = 7.0/4.0**
  - El resultado de la división tendrá decimales, pero si la variable *x* es de tipo entero al asignar el valor habrá truncamiento.
- **(float)1**
  - Convierte 1 a tipo *float*, resultando en 1.0

1.9 Entrada y Salida

La mayor parte de los programas interactúan con el usuario durante su ejecución. De esta forma, el usuario puede suministrar al programa sus datos de entrada, y puede recibir de éste los resultados obtenidos.

Para llevar a cabo esta interacción es necesario contar con instrucciones destinadas para tal efecto. C# incluye un conjunto de instrucciones destinadas a llevar a cabo entrada y salida de datos. Quizá las de mayor uso son las orientadas a la entrada y salida de datos con formato (namespace **`System`**).

Para llevar a cabo salida con formato contamos con la instrucción **`Console.WriteLine`** y **`Console.Write`**, las cuales permiten desplegar en la pantalla del computador información de variados tipos, permitiéndonos a la vez especificar la forma en que dichos datos deberán aparecer (formato).

Ambas instrucciones utilizan una serie de caracteres de conversión para especificar el formato que deberá tener la información de salida. De este modo será posible interpretarla correctamente.

Para obtener entradas con formato disponemos de la instrucción **`Console.ReadLine`** y **`Console.Read`**, las cuales permiten capturar desde el teclado del computador información de variados tipos.

1.9.1 Salida con Formato: Console.WriteLine

La función **`Console.WriteLine`**, permite desplegar datos en la pantalla, con la posibilidad de especificar el formato con que se desea que aparezcan. Esta instrucción se diferencia de **`Console.Write`**, en que la primera incluye un cambio de línea al final, permitiendo comenzar en una nueva línea al siguiente llamado a la misma instrucción, mientras que la segunda no hace un cambio de línea al terminar.

Sintaxis General:

```
Console.WriteLine("cadena-de-formato",<lista_de_argumentos>);
```

La cadena de formato debe ir delimitada por comillas dobles, y debe incluir una referencia y eventual especificación de conversión por cada argumento en la lista de argumentos. El texto que se incluya en la cadena de formato aparecerá durante el despliegue, y cada especificación de conversión será reemplazada por el correspondiente argumento.

La lista de argumentos consiste de una secuencia de expresiones que se desean desplegar. Las expresiones pueden ser variables, constantes u operaciones que deben resolverse antes del despliegue. También es posible especificar llamados a funciones que devuelvan algún valor.

1.9.1.1 Referencia a la Lista de Argumentos

En la cadena de formato, se indican los índices referenciados a la lista de argumentos, especificando el orden en que se mostrarán estos elementos. El primer elemento de la lista corresponde al índice 0, el segundo al 1, y así sucesivamente. Si por ejemplo se requiere mostrar tres datos en pantalla, se puede hacer de la siguiente manera:

```
int a = 1, b = 2, c = 3;
Console.WriteLine("Los valores son: {0}, {1}, {2}", a, b, c);
```

La salida en pantalla de esto sería:

```
1, 2, 3
```

Siempre es posible saltarse la cadena de control, cuando el valor a imprimir es uno solo y no se requiere mayor texto. Para imprimir sólo la variable a, se usa:

```
Console.WriteLine(a);
```

1.9.1.2 Especificaciones de Formato

Es posible definir cómo se requiere mostrar el valor de una variable en la lista de argumentos, de modo de controlar la estética de los resultados en pantalla. Este formato dependerá del tipo de dato que se quiera mostrar, y del espacio que quiera dedicarse en pantalla, lo que ayuda a tabular y organizar la información en pantalla en forma más precisa.

La forma de indicar un especificador de formato es agregando información en el par de llaves para la referencia de la lista de argumentos, de la forma: **`{indice:formato}`**.

indice siempre referencia, desde el 0, al primero, segundo, ... elemento de la lista.

Formato puede ser uno de los siguientes:

- C o c, para formato monetario (tomado de la configuración de Windows)
- D o d, para números enteros (decimales).
- E o e, para números en formato de exponente (ej: 1.234E+005)
- F o f, para números reales, con parte decimal
- G o g, para formato general.
- N o n, similar a F, pero con la separación de miles.
- P o p, porcentaje.
- R o r, round-trip o viaje redondo, que se usa en números reales. Garantiza que un valor numérico convertido a string será re-transformado de vuelta sin perder información.
- X o x, número en formato hexadecimal (ej: 1A24C)

Ejemplo:

```
int a = 123456;
Console.WriteLine("{0:C}", a);  // $123,456.00
Console.WriteLine("{0:D}", a);  // 123456
Console.WriteLine("{0:E}", a);  // 1.23456E+005
Console.WriteLine("{0:F}", a);  // 123456.00
Console.WriteLine("{0:G}", a);  // 123456
```

```
Console.WriteLine("{0:N}", a);  // 123,456.00
Console.WriteLine("{0:P}", a);  // 12,345,600.00 %
Console.WriteLine("{0:X}", a);  // 1E240
```

Para especificar una cantidad definida de ceros en la parte decimal, se acompaña el **formato** con un número que representa la cantidad de ceros.

Ejemplo:

```
int a = 123456;
Console.WriteLine("{0:F5}", a);  // 123456.00000
```

Para controlar el espacio en que esta información se despliega en pantalla, se puede especificar una cantidad de espacios que el dato utilizará en pantalla, e incluso la alineación. Esto se especifica de la forma: **`{indice:espacio}`**

Espacio siempre toma el valor de un número entero, positivo o negativo, de acuerdo a las siguientes condiciones:

- Número negativo, indica que el campo debe ajustarse a la izquierda.
- Número positivo, indica que el campo debe ajustarse a la derecha.
- El número indicado en espacio, indica la cantidad de espacios en total que debe ocupar el campo.

Ejemplo:

```
int a = 123456;
Console.WriteLine("{0,-15:F5}", a);  // 123456.00000
Console.WriteLine("{0,15:F5}", a);   //    123456.00000
Console.WriteLine("{0,10}", a);      //     123456
```

Dentro de la expresión de conversión pueden especificarse las siguientes secuencias que tienen un significado especial:

- **n** - Para provocar que el cursor cambie de línea
- \t - Para producir una tabulación en el despliegue
- \ASCII - Despliega el caracter con el código ASCII suministrado

Ejemplos:

`Console.WriteLine("El valor de la variable x es: {0}", x);`

- Desplegará: **El valor de la variable x es: 100** (si x contenía un valor igual a 100), y avanzará a la siguiente línea luego del despliegue

`Console.WriteLine("El valor de dos + cinco es: {0}", 2+5);`

- Desplegará: **El valor de dos + cinco es: 7** y avanzará a la siguiente línea luego del despliegue

`Console.WriteLine("El área de un circulo con radio {0:F2} es {1:F2}", radio, area);`

- Desplegará: **El area de un circulo con radio 1.00 es 3.14** (suponiendo los valores indicados para las variables) y avanzará a la siguiente línea luego del despliegue

1.9.2 Entrada con Formato: Console.ReadLine y Console.Read

La función **Console.ReadLine**, al igual que **Console.Read**, permite capturar datos desde el teclado, distinguiendo entre ambas por la necesidad de agregar un ENTER al final del ingreso en el caso de la primera.

Sintaxis General:

```
string texto;
texto = Console.ReadLine();
```

Usualmente se requiere transformar esta entrada de datos en algún tipo más razonable de acuerdo al contexto del ejemplo y para ello se utilizan los métodos nativos de conversión desde texto de los tipos de dato numéricos. En dichos casos, la sintaxis es:

```
<tipo> numero;
numero = <tipo>.Parse(Console.ReadLine());
```

La primitiva Parse(), recibe un string y lo transforma en un valor acorde con el tipo de dato referenciado.

Ejemplos:

```
int edad;
Console.Write("Ingrese su edad: ");
edad = int.Parse(Console.ReadLine());
```

- Espera un entero que será almacenado en la variable edad.

```
float nota;
Console.Write("Ingrese su nota: ");
nota = float.Parse(Console.ReadLine());
```

- Espera un número real (con decimales) que será almacenado en la variable nota.

```
string nombre;
Console.Write("Ingrese su Nombre: ");
edad = Console.ReadLine();
```

- Espera un texto (string) que será almacenado en la variable nombre.

1.10 Comentarios, Indentación y Otros Aspectos Visuales

1.10.1 Comentarios

Con el objetivo de hacer los programas más legibles, es importante la incorporación de comentarios que ayuden a comprender lo que hizo el programador. **Es importante documentar dentro del programa cualquier aspecto que no quede claro de la simple lectura del código. Los comentarios son IMPRESCINDIBLES.**

Con este fin, el lenguaje ofrece la posibilidad de incluir dentro del programa cualquier texto que pretenda comentar el código escrito. Este texto puede abarcar una o varias líneas, y debe incluirse dentro de los delimitadores mostrados a continuación.

Sintaxis General:

```
// Comentario simple – siempre se separan por líneas,
//  debiendo comenzar cada nueva línea con otro par de //


/* Comentario – si se usa el asterisco con 'slash', se puede
  seguir en varias líneas sin volver a declarar en cada línea
  el símbolo de comentarios ... hasta que se termina con */
```

Ejemplos:

```
/* Programa para calcular ...
   Escrito por ...

   Este ejemplo se refiere al algoritmo que permite calcular el valor
   de la fórmula con los parámetros ...
*/

int radio;  // Guarda el radio del circulo que se usa como referencia

/* La siguiente función calcula el área de un círculo */
float area(float radio) { ...
```

1.10.2 Indentación (indexación) y otros aspectos visuales

A pesar de que el lenguaje de programación C# ofrece una amplia libertad en cuanto a la forma en que pueden escribirse los programas, **es necesario establecer algunas normas que ayuden a hacer los programas más legibles**, y por lo tanto más fáciles de comprender y modificar. Aunque no es posible ofrecer una lista exhaustiva de todas las decisiones de formato que un programador debe enfrentar al escribir sus programas, intentaremos establecer algunas pautas, y el resto lo dejaremos para ser percibido a lo largo del curso al estudiar los ejemplos.

Ahora bien, las normas aquí mostradas **no son absolutas**, y reflejan únicamente las preferencias del autor. Existen muchas otras, quizá hasta mejores. Lo importante aquí es adoptar un estándar y respetarlo al máximo posible. De esta forma escribiremos programas correctos, no sólo en cuanto a su funcionalidad, sino también en cuanto a su apariencia.

1.10.3 Algunos consejos

- ¿Dónde poner los corchetes?

```
funcion (argumentos) {
    ...
  }
```

- ¿Cómo indentar el código?
  Los programas normalmente tienen una estructura jerárquica (por niveles), y es importante destacarla al escribirlos. Esto se logra mediante indentación, es decir, dejando algunos espacios entre el código que esta en un nivel y el del siguiente nivel. Pero ¿cuándo indentar?. Lo más sencillo es mirar los ejemplos. Una regla sencilla es que **cada vez que se abran corchetes** se estará creando un nuevo nivel y por lo tanto es necesario indentar. Por ejemplo:

```
static void Main () {
   ...
   while (...) {
      ...
      for (...) {
         ...
      }
      ...
   }
   ...
}
```

- Para ponerle nombre a las cosas
  Trate siempre de que el nombre ayude a comprender la función del objeto que está identificando. Por ejemplo, una variable que almacena la edad de un empleado podría llamarse *Edad*, o mejor aún *EdadEmpleado*, y no simplemente *x*. Las funciones deben tener nombres activos, como *CalculaPension*, y no simplemente *Pension*, o peor aún un nombre sin sentido. Cuanto más significativos sean los nombres, menor será la cantidad de comentarios que deberá incluir.

- ¿Cuándo poner comentarios?

  Es importante **iniciar todo programa con un comentario en el que se explica lo que hace el programa de manera general**, cuáles son sus entradas y cuáles sus salidas, así como las restricciones que tiene, si es que existen. Toda función deberá tener un comentario similar. Además, aquellas variables cuyo nombre no sea lo suficientemente explícito, deberán tener un comentario sobre su objetivo en la declaración. Por ejemplo:

```
/* Programa para ...
   El programa lee del usuario ...
   ... y le devuelve como respuesta ...
   Restricciones: No es posible ...
*/

int x;              /* Almacena el valor para ... */

int EdadEmpleado;   /* Obvio: No hace falta comentario */


/* Funcion para .... */
...

/* Funcion para .... */
```

```
...

/* Funcion principal: Hace lo siguiente ... */
main() {

   /* Dentro del codigo tambien puede haber comentarios,
      para explicar aspectos que pueden ser dificiles de
      comprender */
}
```