DOSSIER
PROGRAMACIÓN I

DOSSIER ASIGNATURA

PROGRAMACIÓN I

DATOS GENERALES DE LA ASIGNATURA

Carrera:	Ingeniería en Sistemas
Plan de Estudio:	Ciencias Económicas y Administrativas
Asignatura:	Programación I
Requisito:	Fundamentos de Programación
Régimen:	Cuatrimestral
Semestre/cuatrimestre:	III Cuatrimestre
Año Lectivo:	2020
Modalidad:	Presencial
Frecuencia semanal:	4 horas clase
Turno:	Diurno
Horas:	64
Créditos:	4
Año académico:	I
Docente:	Hanier Steve Morales Pérez
Contacto:	8511-1445
Correo Electrónico:	hsmp16@gmail.com

INDICE

I. INTRODUCCIÓN 1
II. OBJETIVOS 2
2.1 Objetivos Conceptuales 2
2.2 Objetivos Procedimentales 2
2.3 Objetivos Actitudinales 3
2.4 Logros de Aprendizaje 3
III. UNIDADES 4
3.1 Unidad I – Introducción a la Programación Estructurada 4
3.1.1 ¿Qué es C# y qué entorno usaremos? 4
3.1.2 Escribir un texto en C# 4
3.1.3 Operaciones Aritméticas 5
3.1.4 Identificadores 6
3.1.5 Comentarios 6
3.1.6 Datos por el usuario 7
3.1.7 using System 8
3.2 Unidad II - Estructuras de Control, Condicionales y Repetitivas 9
3.2.1 Estructura alternativa if 9
3.2.2 if y Sentencias Compuestas 10
3.2.3 Operadores relacionales 11
3.2.4 if – else 11
3.2.5 Sentencia switch 12
3.2.6 Estructura repetitiva while 13
3.2.7 Contadores 14
3.2.8 Estructura foreach 14
3.3 Unidad III – Estructuras y Arreglos 15
3.3.1 Definición de un array 15
3.3.2 Valor inicial de un arreglo 16
3.3.3 Recorriendo los elementos de una tabla 17
3.4 Unidad IV – Funciones y Apuntadores 19
3.4.1 Conceptos básicos sobre funciones 19
3.4.2 Parámetros de una función 20
3.4.3 Estructuras dinámicas 23
3.4.4 Pila en C# 24
3.4.5 Una cola en C# 25
3.4.6 Las listas 26
3.4.7 Array List 26
IV. Conclusiones 28
V. Bibliografía 29
VI. Recursos Electrónicos 29

I. INTRODUCCIÓN

La asignatura **Programación I** es la complementariedad de *Fundamentos de Programación*. Por tanto, es requisito fundamental haber cursado y aprobado esta última. En esta asignatura se pretende dotar al estudiante de las técnicas de programación estructuradas, potencializando el pensamiento lógico para la resolución de tareas mediante un lenguaje de programación en donde él y la estudiante sean capaces de combinar las técnicas algorítmicas previamente aprendidas con las reglas sintácticas y semánticas de los lenguajes de programación.

La asignatura persigue la adquisición, desarrollo y perfeccionamiento de las técnicas de programación, orientada hacia la programación estructurada, y se tratará de abordar un poco hacia la programación orientada a objetos. Alguno de los contenidos que abarcará son: Conceptos Básicos, Operadores y Expresiones, haciendo mucho énfasis en las Estructuras de Control, arreglos, funciones (del sistema y definidas por el usuario), y apuntadores. Se recomienda utilizar el lenguaje de programación C#. Además, se ha agregado en la última unidad los temas sobre Robótica Educativa.

Los contenidos de la asignatura se distribuyen en 64 horas presenciales y 128 horas de estudio independiente. El programa contiene 4 créditos académicos.

II. OBJETIVOS

2.1 Objetivos Conceptuales

2.1.1 Conoce los conceptos básicos de programación, técnicas de programación estructurada, modular y orientada a objetos.

2.1.2 Conoce los operadores lógicos, aritméticos y relacionales, sus prioridades y expresiones matemáticas.

2.1.3 Identifica y aplica estructuras de control: estructuras condicionales y estructuras repetitivas.

2.1.4 Conoce y aplica el funcionamiento de los arreglos unidimensionales y multidimensionales.

2.1.5 Conoce y usa las funciones propias del Lenguaje C#.

2.1.6 Conoce la metodología de creación de funciones definidas por el usuario.

2.1.7 Conoce el funcionamiento de los apuntadores, y la metodología y ventajas de uso en la creación de programas.

2.1.8 Explorar e investigar información sobre robótica educativa.

2.2 Objetivos Procedimentales

2.2.1 Domina conceptos de programación y técnicas de programación estructurada y modular.

2.2.2 Maneja los tipos de datos estáticos básicos y compuestos.

2.2.3 Maneja las estructuras de control.

2.2.4 Capacidad de resolución de problemas mediante diseño descendente y dominio de la utilización de funciones.

2.2.5 Maneja de los tipos de datos dinámicos y de la gestión dinámica de memoria.

2.2.6 Documentación y estructuración de código.

2.2.7 Es capaz de resolver expresiones matemáticas que contengan cualquiera de los operadores.

2.2.8 Resuelve problemas aplicados a la programación, usando sentencias de control.

2.2.9 Construye y simula programas en lenguaje C#, que requieren de arreglos unidimensionales y multidimensionales.

2.2.10 Crea programas en C# utilizando funciones del sistema.

2.2.11 Crea funciones definidas según la necesidad y el problema a resolver.

2.2.12 Crea programas haciendo uso de apuntadores en diferentes ámbitos.

2.2.13 Construir código en bloques de programación en robótica educativa.

2.3 Objetivos Actitudinales

2.3.1 Desarrollar en el futuro profesional, el buen uso de la lógica computacional y la aplicación de los ejes transversales de URACCAN: Perspectiva Intercultural de Género, Interculturalidad, Autonomía de los Pueblos, Buen Vivir y Desarrollo con Identidad, Diálogo de Saberes, Articulación Institucional.

2.3.2 Practica el compañerismo, la disciplina, la puntualidad, la honestidad y el respeto mutuo durante el proceso de enseñanza aprendizaje.

2.3.3 Capacidad para trabajar con un entorno de desarrollo integrado.

2.4 Logros de Aprendizaje

2.4.1 Dominar la programación estructurada.

2.4.2 Domina las bases del lenguaje C#.

2.4.3 Entender el entorno .NET.

2.4.4 Emplear el lenguaje C# para la solución de problemas.

2.4.5 Comprender la sintaxis de C# mediante la programación estructurada.

2.4.6 Interpretar el paradigma de orientación a objetos como una herramienta para crear distintos tipos de programas.

2.4.7 Construir algoritmos y bloques de código con robótica educativa.

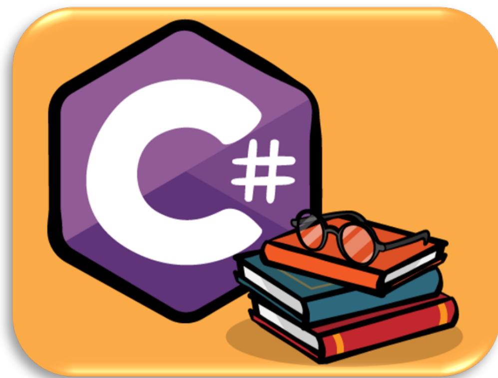
III. UNIDADES

3.1 Unidad I – Introducción a la Programación Estructurada

3.1.1 ¿Qué es C# y qué entorno usaremos?

C# es un lenguaje de programación de ordenadores. Se trata de un lenguaje moderno, evolucionado a partir de C y C++, y con una sintaxis muy similar a la de Java. Los programas creados con C# no suelen ser tan rápidos como los creados con C, pero a cambio la productividad del programador es mucho mayor y es más difícil cometer errores.

Se trata de un lenguaje creado por Microsoft para realizar programas para su plataforma .NET, pero fue estandarizado posteriormente por ECMA y por ISO, y existe una implementación alternativa de "código abierto", el "proyecto Mono", que está disponible para Windows, Linux, Mac OS X y otros sistemas operativos. Empezaremos por implementar Visual Studio el cual puedes encontrarlo en la Web Oficial de Microsoft.



3.1.2 Escribir un texto en C#

Veamos con un primer ejemplo de programa en C#, posiblemente el más sencillo. Se trata de escribir un texto en pantalla. Vamos a analizarlo ahora con más detalle:

```
1  public class Ejemplo01
2  {
3      public static void Main()
4      {
5          System.Console.WriteLine("Hola");
6      }
7  }
```

Esto escribe "Hola" en la pantalla. Pero hay muchas "cosas raras" alrededor de ese "Hola", de modo vamos a comentarlas antes de proseguir, aunque muchos de los detalles los aplazaremos para más adelante. En este primer análisis, iremos desde dentro hacia fuera:

- **WriteLine("Hola");** : "Hola" es el texto que queremos escribir, y WriteLine es la orden encargada de escribir (Write) una línea (Line) de texto en pantalla.

- **Console.WriteLine("Hola");** : WriteLine siemprp irá precedido de "Console." porque es una orden de manejo de la "consola" (la pantalla "negra" en modo texto del sistema operativo).

- **System.Console.WriteLine("Hola");** : Las órdenes relacionadas con el manejo de consola (Console) pertenecen a la categoría de sistema (System).
- Las llaves { y } se usan para delimitar un bloque de programa. En nuestro caso, se trata del bloque principal del programa (Main).

- **public static void Main()** : Main indica cual es "el cuerpo del programa", la parte principal (un programa puede estar dividido en varios fragmentos, como veremos más adelante). Todos los programas tienen que tener un bloque "Main". Los detalles de por qué hay que poner delante "public static void" y de por qué se pone después un paréntesis vacío los iremos aclarando más tarde. De momento, deberemos memorizar que ésa será la forma correcta de escribir "Main".

- **public class Ejemplo01** : De momento pensaremos que "Ejemplo01" es el nombre de nuestro programa. Una línea como esa deberá existir también siempre en nuestros programas (aunque el nombre no tiene por qué ser tan "rebuscado"), y eso de "public class" será obligatorio. Nuevamente, aplazamos para más tarde los detalles sobre qué quiere decir "class" y por qué debe ser "public".

3.1.3 Operaciones Aritméticas

Parece evidente que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero alguna de las operaciones matemáticas habituales tienen símbolos menos intuitivos. Veamos cuales son los más importantes:

Operador	Operación
+	Suma
-	
*	Multiplicación
/	División
%	Resto de la división ("módulo")

Así, podemos calcular el resto de la división entre dos números de la siguiente forma:

```
public class Ejemplo02
{
    public static void Main()
    {
        System.Console.WriteLine("El resto de dividir 19 entre 5 es");
        System.Console.WriteLine(19 % 5);
    }
}
```

3.1.4 Identificadores

Los nombres de variables (lo que se conoce como identificadores) pueden estar formados por letras, números o el símbolo de subrayado (_) y deben comenzar por letra o subrayado. No deben tener espacios intermedios. También hay que recordar que las vocales acentuadas y la eñe son problemáticas, porque no son letras "estándar" en todos los idiomas, así que no se pueden utilizar como parte de un identificador en la mayoría de lenguajes de programación.

Por eso, no son nombres de variable válidos:

```
1    1numero       (empieza por número)
2    un numero     (contiene un espacio)
3    Año1          (tiene una eñe)
4    MásDatos      (tiene una vocal acentuada)
```

3.1.5 Comentarios

Podemos escribir comentarios, que el compilador ignorará, pero que pueden ser útiles para nosotros mismos, haciendo que sea más fácil recordar el cometido un fragmento del programa más adelante, cuando tengamos que ampliarlo o corregirlo.

Existen dos formas de indicar comentarios. En su forma más general, los escribiremos entre /* y */:

```
1    int suma;  /* Guardaré el valor para usarlo más tarde */
```

Un comentario puede empezar en una línea y terminar en otra distinta, así:

```
/*  Esto
    es un comentario que
    ocupa más de una línea
*/
```

3.1.6 Datos por el usuario

Hasta ahora hemos tenido datos prefijados, pero eso es poco frecuente en el mundo real. Es mucho más habitual que los datos los introduzca el usuario, o que se lean desde un fichero, o desde una base de datos, o se reciban de Internet o cualquier otra red. El primer caso que veremos será el de interaccionar directamente con el usuario.

Si queremos que sea el usuario de nuestro programa quien teclee los valores, necesitamos una nueva orden, que nos permita leer desde teclado. Pues bien, al igual que tenemos *Console.WriteLine*, también existe *Console.ReadLine*. Para leer textos, haríamos:

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

Un ejemplo de programa que sume dos números tecleados por el usuario sería:

```
// Ejemplo en C#: sumar dos números introducidos por el usuario
public class Ejemplo03
{
    public static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        System.Console.WriteLine("Introduce el primer número");
        primerNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        System.Console.WriteLine("Introduce el segundo número");
        segundoNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        suma = primerNumero + segundoNumero;

        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

3.1.7 using System

Va siendo hora de hacer una pequeña mejora: no es necesario repetir "System." al principio de la mayoría de las órdenes que tienen que ver con el sistema (por ahora, las de consola y las de conversión), si al principio del programa utilizamos "using System":

```
1   // Ejemplo en C#: "using System" en vez de "System.Console"
2   using System;
3   
4   public class Ejemplo04
5   {
6       public static void Main()
7       {
8           int primerNumero;
9           int segundoNumero;
10          int suma;
11  
12          Console.WriteLine("Introduce el primer número");
13          primerNumero = Convert.ToInt32(Console.ReadLine());
14          Console.WriteLine("Introduce el segundo número");
15          segundoNumero = Convert.ToInt32(Console.ReadLine());
16          suma = primerNumero + segundoNumero;
17  
18          Console.WriteLine("La suma de {0} y {1} es {2}",
19              primerNumero, segundoNumero, suma);
20      }
21  }
```

3.2 Unidad II - Estructuras de Control, Condicionales y Repetitivas

Casi cualquier problema del mundo real que debamos resolver o tarea que deseemos automatizar supondrá tomar decisiones: dar una serie de pasos en función de si se cumplen ciertas condiciones o no. En muchas ocasiones, además esos pasos deberán ser repetitivos. Vamos a ver cómo podemos comprobar si se cumplen condiciones y también cómo hacer que un bloque de un programa se repita.

3.2.1 Estructura alternativa if

La primera construcción que emplearemos para comprobar si se cumple una condición será **si ... entonces**. Su formato es:

```
1    if (condición) sentencia;
```

Es decir, debe empezar con la palabra *if*, la condición se debe indicar entre paréntesis y a continuación se detallará la orden que hay que realizar en caso de cumplirse esa condición, terminando con un punto y coma.

Vamos a verlo con un ejemplo:

```
1   // Condiciones con if
2
3   using System;
4
5   public class Ejemplo05
6   {
7       public static void Main()
8       {
9           int numero;
10
11          Console.WriteLine("Introduce un número");
12          numero = Convert.ToInt32(Console.ReadLine());
13          if (numero>0) Console.WriteLine("El número es positivo.");
14      }
15  }
```

Este programa anterior pide un número al usuario. Si es positivo (mayor que 0), escribe en pantalla "El número es positivo."; si es negativo o cero, no hace nada.

Como se ve en el ejemplo, para comprobar si un valor numérico es mayor que otro, usamos el símbolo ">". Para ver si dos valores son iguales, usaremos dos símbolos de "igual": if (numero==0). Las demás posibilidades las veremos dentro de muy poco. En todos los casos, la condición que queremos comprobar deberá indicarse entre paréntesis.

Este programa comienza por un comentario algo más detallado que los de los ejemplos anteriores, que nos recuerda de qué se trata.

Si la orden *if* es larga, se puede partir en dos líneas para que resulte más legible**:**

```
1        if (numero>0)
2            Console.WriteLine("El número es positivo.");
```

3.2.2 if y Sentencias Compuestas

Habíamos dicho que el formato básico de *if* es *if (condición) sentencia*; Esa sentencia que se ejecuta si se cumple la condición puede ser una sentencia simple o una compuesta. Las sentencias compuestas se forman agrupando varias sentencias simples entre llaves ({ y }), como en este ejemplo:

```
1   using System;
2   
3   public class Ejemplo06
4   {
5       public static void Main()
6       {
7           int numero;
8   
9           Console.WriteLine("Introduce un número");
10          numero = Convert.ToInt32(Console.ReadLine());
11          if (numero > 0)
12          {
13              Console.WriteLine("El número es positivo.");
14              Console.WriteLine("Recuerde que también puede usar negativos.");
15          }  // Aquí acaba el "if"
16      }      // Aquí acaba "Main"
17  }          // Aquí acaba "Ejemplo06"
18  
```

3.2.3 Operadores relacionales

Hemos visto que el símbolo **>** es el que se usa para comprobar si un número es mayor que otro. El símbolo de "menor que" también es sencillo, pero los demás son un poco menos evidentes, así que vamos a verlos:

Operador	Operación	Operador	Operación
<	Menor que	**>=**	Mayor o igual que
>	Mayor que	**==**	Igual a
<=	Menor o igual que	**!=**	No igual a (distinto de)

Así, un ejemplo, que diga si un número no es cero sería:

```
1   using System;
2
3   public class Ejemplo07
4   {
5       public static void Main()
6       {
7           int numero;
8
9           Console.WriteLine("Introduce un número");
10          numero = Convert.ToInt32(Console.ReadLine());
11          if (numero != 0)
12              Console.WriteLine("El número no es cero.");
13      }
14  }
```

3.2.4 if – else

Podemos indicar lo que queremos que ocurra en caso de que no se cumpla la condición, usando la orden "else" (en caso contrario), así:

```
1   using System;
2
3   public class Ejemplo08
4   {
5       public static void Main()
6       {
7           int numero;
8
9           Console.WriteLine("Introduce un número");
10          numero = Convert.ToInt32(Console.ReadLine());
11          if (numero > 0)
12              Console.WriteLine("El número es positivo.");
13          else
14              Console.WriteLine("El número es cero o negativo.");
15      }
16  }
```

3.2.5 Sentencia switch

Si queremos ver **varios posibles valores**, sería muy pesado tener que hacerlo con muchos "if" seguidos o encadenados. La alternativa es emplear la orden "switch", cuya sintaxis es:

```
1   switch (expresion)
2   {
3       case valor1: sentencia1;
4                    break;
5       case valor2: sentencia2;
6                    sentencia2b;
7                    break;
8       case valor3:
9                    goto case valor1;
10          ...
11      case valorN: sentenciaN;
12                   break;
13      default:
14                   otraSentencia;
15                   break;
16  }
```

- Tras la palabra "**switch**" se escribe la expresión a analizar, entre paréntesis.
- Después, tras varias órdenes "**case**" se indica cada uno de los valores posibles.
- Los pasos (porque pueden ser varios) que se deben dar si la expresión tiene un cierto valor se indican a continuación, terminando con "**break**".
- Si hay que hacer algo en caso de que no se cumpla ninguna de las condiciones, se detalla después de la palabra "**default**".

Una cadena de texto, se declara con la palabra *string*, se puede leer de teclado con ReadLine (sin necesidad de convertir) y se le puede dar un valor desde programa si se indica entre comillas dobles. Por ejemplo, un programa que nos salude de forma personalizada si somos "Hanier" o "Pedro" podría ser:

```
1   using System;
2
3   public class Ejemplo09
4   {
5       public static void Main()
6       {
7           string nombre;
8
9           Console.WriteLine("Introduce tu nombre");
10          nombre = Console.ReadLine();
11
12          switch (nombre)
13          {
14              case "Hanier":  Console.WriteLine("Bienvenido, Hanier.");
15                             break;
16              case "Pedro": Console.WriteLine("Que tal estas, Pedro.");
17                             break;
18              default:       Console.WriteLine("No recuerdo quién eres.");
19                             break;
20          }
21      }
22  }
```

3.2.6 Estructura repetitiva while

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden "while". Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final del bloque repetitivo.

En el primer caso, su sintaxis es:

```
while (condición)
    sentencia;
```

Es decir, la sentencia se repetirá **mientras** la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita más de una sentencia, basta agruparlas entre llaves.

Un ejemplo que nos diga si cada número que tecleemos es positivo o negativo, y que termine cuando tecleemos el número 0, podría ser:

```
using System;

public class Ejemplo10
{
    public static void Main()
    {
        int numero;

        Console.Write("Teclea un número (0 para salir): ");
        numero = Convert.ToInt32(Console.ReadLine());

        while (numero != 0)
        {
            if (numero > 0) Console.WriteLine("Es positivo");
            else Console.WriteLine("Es negativo");

            Console.WriteLine("Teclea otro número (0 para salir): ");
            numero = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

3.2.7 Contadores

Ahora que sabemos "repetir" cosas, podemos utilizarlo también para contar. Por ejemplo, si queremos contar del 1 al 5, usaríamos una variable que empezase en 1, que aumentaría una unidad en cada repetición y se repetiría hasta llegar al valor 5, así como esta imagen de la izquierda.

```
1  using System;
2
3  public class Ejemplo11
4  {
5      public static void Main()
6      {
7          int n = 1;
8
9          while (n < 6)
10         {
11             Console.WriteLine(n);
12             n = n + 1;
13         }
14     }
15 }
```

3.2.8 Estructura foreach

Nos queda por ver otra orden que permite hacer cosas repetitivas: "foreach" (se traduciría "para cada"). La veremos más adelante, cuando manejemos estructuras de datos más complejas, que es en las que la nos resultará útil para extraer los datos de uno en uno. De momento, el único dato compuesto que hemos visto (y todavía con muy poco detalle) es la cadena de texto, "string", de la que podríamos obtener las letras una a una con "foreach" así:

```
1  using System;
2
3  public class Ejemplo12
4  {
5      public static void Main()
6      {
7          Console.Write("Dime tu nombre: ");
8          string nombre = Console.ReadLine();
9          foreach(char letra in nombre)
10         {
11             Console.WriteLine(letra);
12         }
13     }
```

3.3 Unidad III – Estructuras y Arreglos

3.3.1 Definición de un array

Una tabla, vector, matriz o arreglo es un conjunto de elementos, todos los cuales son del mismo tipo, y a los que accederemos usando el mismo nombre. Por ejemplo, si queremos definir un grupo de números enteros, el tipo de datos que usaremos para declararlo será *int [ ]*:

```
1    int[] ejemplo;
```

Cuando sepamos cuantos datos vamos a guardar (por ejemplo 4), podremos reservar espacio con la orden "new", así:

```
1    ejemplo = new int[4];
```

Si sabemos el tamaño desde el principio, podemos reservar espacio a la vez que declaramos la variable:

```
1    int[] ejemplo = new int[4];
```

Podemos acceder a cada uno de los valores individuales indicando su nombre (ejemplo) y el número de elemento que nos interesa, pero con una precaución: se empieza a numerar desde 0, así que en el caso anterior tendríamos 4 elementos, que serían ejemplo[0], ejemplo[1], ejemplo[2], ejemplo[3]. Por tanto, podríamos dar el valor 15 al primer elemento de nuestro array así:

```
1    ejemplo[0] = 15;
```

Habitualmente, como un array representa un conjunto de números, utilizaremos nombres en plural, como en:

```
1    int[] datos = new int[100];
```

Como ejemplo, vamos a definir un grupo de 5 números enteros y hallar su suma:

```
1   using System;
2   
3   public class Ejemplo13
4   {
5       public static void Main()
6       {
7   
8           int[] numeros = new int[5];   // Un array de 5 números enteros
9           int suma;                     // Un entero que será la suma
10  
11          numeros[0] = 200;             // Les damos valores
12          numeros[1] = 150;
13          numeros[2] = 100;
14          numeros[3] = -50;
15          numeros[4] = 300;
16          suma = numeros[0] +           // Y calculamos la suma
17              numeros[1] + numeros[2] + numeros[3] + numeros[4];
18          Console.WriteLine("Su suma es {0}", suma);
19          // Nota: esta es la forma más ineficiente e incómoda
20          // Lo mejoraremos...
21      }
22  }
```

3.3.2 Valor inicial de un arreglo

```
1   using System;
2   
3   public class Ejemplo14
4   {
5       public static void Main()
6       {
7           int[] numeros =             // Un array de 5 números enteros
8               {200, 150, 100, -50, 300};
9           int suma;                   // Un entero que será su suma
10  
11          suma = numeros[0] +         // Hallamos la suma
12              numeros[1] + numeros[2] + numeros[3] + numeros[4];
13          Console.WriteLine("Su suma es {0}", suma);
14          // Nota: esta forma es algo menos engorrosa, pero todavía no
15          // está bien hecho.  Lo seguiremos mejorando.
16      }
17  }
```

Como se muestra acá arriba, al igual que ocurría con las variables "normales", podemos dar valor inicial a los elementos de una tabla al principio del programa, si conocemos todos su valores. En este caso, los indicamos todos entre llaves, separados por comas.

3.3.3 Recorriendo los elementos de una tabla

Es de esperar que exista una forma más cómoda de acceder a varios elementos de un array, sin tener siempre que repetirlos todos, como hemos hecho en:

```
suma = numeros[0] + numeros[1] + numeros[2] + numeros[3] + numeros[4];
```

El truco consistirá en emplear cualquiera de las estructuras repetitivas que ya hemos visto (while, do..while, for), por ejemplo así:

```
suma = 0;                   // Valor inicial de la suma: 0
    for (int i=0; i<=4; i++)   // Y calculamos la suma repetitiva
        suma += numeros[i];
```

El programa ya completo se verá así:

```
using System;

public class Ejemplo15
{
    public static void Main()
    {
        int[] numeros =             // Un array de 5 números enteros
            {200, 150, 100, -50, 300};
        int suma;                   // Un entero que será su suma

        suma = 0;                   // Valor inicial de la suma: 0
        for (int i=0; i<=4; i++)    // Y calculamos la suma repetitiva
            suma += numeros[i];

        Console.WriteLine("Su suma es {0}", suma);
    }
}
```

En este caso, que sólo sumábamos 5 números, no hemos escrito mucho menos, pero si trabajásemos con 100, 500 o 1000 números, la ganancia en comodidad sí que sería evidente.

Lógicamente, también podemos pedir los datos al usuario de forma repetitiva, usando estructuras:

```
1   using System;
2
3   public class Ejemplo16
4   {
5       public static void Main()
6       {
7
8           int[] numeros = new int[5];
9           int suma;
10
11          for (int i=0; i<=4; i++)   // Pedimos los datos
12          {
13              Console.Write("Introduce el dato numero {0}: ", i+1);
14              numeros[i] = Convert.ToInt32(Console.ReadLine());
15          }
16
17          suma = 0;            // Y calculamos la suma
18          for (int i=0; i<=4; i++)
19              suma += numeros[i];
20
21          Console.WriteLine("Su suma es {0}", suma);
22      }
23  }
```

3.4 Unidad IV – Funciones y Apuntadores

3.4.1 Conceptos básicos sobre funciones

En C#, al igual que en C y los demás lenguajes derivados de él, todos los "trozos de programa" son funciones, incluyendo el propio cuerpo de programa, **Main**. De hecho, la forma básica de **definir** una función será indicando su nombre seguido de unos paréntesis vacíos, como hacíamos con "Main", y precediéndolo por ciertas palabras reservadas, como "public static **void**", cuyo significado iremos viendo muy pronto. Después, entre llaves indicaremos todos los pasos que queremos que dé ese "trozo de programa".

Por ejemplo, podríamos crear una función llamada "Saludar", que escribiera varios mensajes en la pantalla:

```
1    public static void Saludar()
2    {
3      Console.Write("Bienvenido al programa ");
4      Console.WriteLine("de ejemplo");
5      Console.WriteLine("Espero que estes bien");
6    }
```

Ahora desde dentro del cuerpo de nuestro programa, podríamos *llamar* a esa función:

```
1    public static void Main()
2    {
3      Saludar();
4      ...
5    }
```

Así conseguimos que nuestro programa principal sea más fácil de leer.

Un detalle importante: tanto la función habitual "Main" como la nueva función "Saludar" serían parte de nuestra "class", es decir, el código completo sería el siguiente(ver en la próxima página):

```
using System;

public class Ejemplo17
{

    public static void Saludar()
    {
        Console.Write("Bienvenido al programa ");
        Console.WriteLine("de ejemplo");
        Console.WriteLine("Espero que estés bien");
    }

    public static void Main()
    {
        Saludar();
        Console.WriteLine("Nada más por hoy...");
    }

}
```

Como ejemplo más detallado, la parte principal de una agenda o de una base de datos simple como las que hicimos en el tema anterior, podría ser simplemente:

```
LeerDatosDeFichero();
do {
   MostrarMenu();
   opcion = PedirOpcion();
   switch( opcion ) {
      case 1: BuscarDatos(); break;
      case 2: ModificarDatos(); break;
      case 3: AnadirDatos(); break;
      ...
```

3.4.2 Parámetros de una función

Es muy frecuente que nos interese indicarle a nuestra función ciertos datos con los que queremos que trabaje. Los llamaremos "parámetros" y los indicaremos dentro del paréntesis que sigue al nombre de la función, separados por

comas. Para cada uno de ellos, deberemos indicar su tipo de datos (por ejemplo "int") y luego su nombre.

Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil crear una función auxiliar que nos los muestre con el formato que nos interese (que podría ser con exactamente 3 decimales). Lo podríamos hacer así:

```
1    public static void EscribirNumeroReal( float n )
2    {
3        Console.WriteLine( n.ToString("#.###") );
4    }
```

Y esta función se podría usar desde el cuerpo de nuestro programa así:

```
1   EscribirNumeroReal(2.3f);
```

Recordemos que el sufijo *f* sirve para indicar al compilador que trate ese número como un *float*, porque de lo contrario, al ver que tiene cifras decimales, lo tomaría como *double*, que permite mayor precisión... pero a cambio nosotros tendríamos un mensaje de error en nuestro programa, diciendo que estamos pasando un dato "double" a una función que espera un *float*.

El programa completo podría quedar así:

```
1   using System;
2   
3   public class Ejemplo18
4   {
5   
6       public static void EscribirNumeroReal( float n )
7       {
8           Console.WriteLine( n.ToString("#.###") );
9       }
10  
11      public static void Main()
12      {
13          float x;
14  
15          x= 5.1f;
16          Console.WriteLine("El primer numero real es: ");
17          EscribirNumeroReal(x);
18          Console.WriteLine(" y otro distinto es: ");
19          EscribirNumeroReal(2.3f);
20      }
21  
22  }
```

Como ya hemos anticipado, si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos (incluso si todos son del mismo tipo), y separarlos entre comas:

```
1    public static void EscribirSuma( int a, int b )
2    {
3        ...
4    }
```

De modo que un programa completo de ejemplo para una función con dos parámetros podría ser:

```
using System;

public class Ejemplo19
{

    public static void EscribirSuma( int a, int b )
    {
        Console.Write( a+b );
    }

    public static void Main()
    {
        Console.Write("La suma de 4 y 7 es: ");
        EscribirSuma(4, 7);
    }

}
```

Como se ve en estos ejemplos, se suele seguir un par de convenios:

- Ya que las funciones expresan acciones, en general su nombre será un verbo.

- En C# se recomienda que los elementos públicos se escriban comenzando por una letra mayúscula (y recordemos que, hasta que conozcamos las alternativas y el motivo para usarlas, nuestras funciones comienzan con la palabra "public"). Este criterio depende del lenguaje.

3.4.3 Estructuras dinámicas

Hasta ahora teníamos una serie de variables que declarábamos al principio del programa o de cada función. Estas variables, que reciben el nombre de *estáticas*, tienen un tamaño asignado desde el momento en que se crea el programa. Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Por ejemplo, si queremos crear una agenda, necesitaremos ir añadiendo nuevos datos. Si reservamos espacio para un máximo de 10 fichas, no podremos llegar a añadir la número 11. Una solución típica (pero no muy buena) es sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, etc. Otra (no muy buena) solución sería la de trabajar siempre en el disco y usar acceso aleatorio para leer cada ficha desde disco cuando sea necesario, pero esta alternativa es muchísimo más lenta.

La solución real suele ser crear estructuras *dinámicas*, que puedan ir creciendo o disminuyendo según nos interese.

Algunos ejemplos de estructuras de este tipo son:

- **Las pilas:** Una pila de datos se comportará de forma similar a una pila de libros: podemos apilar cosas en la cima, o extraer de la cima. Se supone que no se puede tomar elementos de otro sitio que no sea la cima, ni dejarlos en otro sitio distinto. De igual modo, se supone que la pila no tiene un tamaño máximo definido, sino que puede crecer arbitrariamente.

- **Las colas:** Una cola de datos se comportará como las del cine (en teoría): la gente llega por un sitio (la cola) y sale por el opuesto (la cabeza). Al igual que antes, supondremos que un elemento no puede entrar a la cola ni salir de ella en posiciones intermedias y que la cola puede crecer hasta un tamaño indefinido.

- **Las listas:** Mas versátiles pero más complejas de programar, en las que se puede añadir elementos en cualquier posición y obtenerlos o borrarlos de cualquier posición.

Y existen otras estructuras dinámicas más complejas y que nosotros no trataremos, como los árboles, en los que cada elemento puede tener varios sucesores (se parte de un elemento raíz, y la estructura se va ramificando), o los grafos, formados por una serie de nodos unidos por aristas.

Todas estas estructuras tienen en común que, si se programan correctamente, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño (máximo) prefijado. Además, ciertas operaciones, como las ordenaciones o los borrados, pueden ser más rápidas que en un array.

Veremos ejemplos de cómo crear estructuras dinámicas de estos tipos en C#.

3.4.4 Pila en C#

Para crear una pila, emplearemos la clase Stack. Una pila nos permitirá introducir un nuevo elemento en la cima ("apilar", en inglés "push") y quitar el elemento que hay en la cima ("desapilar", en inglés "pop"). Este tipo de estructuras se suele denotar también usando las siglas "LIFO" (Last In First Out: lo último en entrar es lo primero en salir).

Para utilizar la clase "Stack" y la mayoría de las que veremos en este tema, necesitamos incluir en nuestro programa una referencia a "System.Collections". Así, un ejemplo básico que creara una pila, introdujera tres palabras y luego las volviera a mostrar sería:

```
1   using System;
2   using System.Collections;
3
4   public class Ejemplo20
5   {
6       public static void Main()
7       {
8           string palabra;
9
10          Stack miPila = new Stack();
11          miPila.Push("Hola,");
12          miPila.Push("soy");
13          miPila.Push("yo");
14
15          for (byte i=0; i<3; i++)
16          {
17              palabra = (string) miPila.Pop();
18              Console.WriteLine( palabra );
19          }
20      }
21  }
```

Como se puede ver en este ejemplo, no hemos indicado que sea una pila de strings, sino simplemente una pila. Por eso, los datos que extraemos son objetos, que deberemos convertir al tipo de datos que nos interese utilizando un "typecast" (conversión forzada de tipos), como en palabra = (string) miPila.Pop();

La implementación de una pila en C# es algo más avanzada que lo que podríamos esperar en una

pila estándar: incorpora también métodos como:

- "Peek", que mira el valor que hay en la cima, pero sin extraerlo.
- "Clear", que borra todo el contenido de la pila.
- "Contains", que indica si un cierto elemento está en la pila.
- "ToArray", que devuelve toda la pila convertida a un array.
- También tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

3.4.5 Una cola en C#

Podemos crear colas si nos apoyamos en la clase Queue. En una cola podremos introducir elementos por la cabeza ("Enqueue", encolar) y extraerlos por el extremo opuesto, el final de la cola ("Dequeue", desencolar). Este tipo de estructuras se nombran a veces también por las siglas FIFO (First In First Out, lo primero en entrar es lo primero en salir). Un ejemplo básico similar al anterior, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```
1   using System;
2   using System.Collections;
3
4   public class Ejemplo21
5   {
6       public static void Main()
7       {
8           string palabra;
9
10          Queue miCola = new Queue();
11          miCola.Enqueue("Hola,");
12          miCola.Enqueue("soy");
13          miCola.Enqueue("yo");
14
15          for (byte i=0; i<3; i++)
16          {
17              palabra = (string) miCola.Dequeue();
18              Console.WriteLine( palabra );
19          }
20      }
21  }
```

La implementación de una cola que incluye C# es más avanzada que eso, con métodos similares a los de antes:

- "Peek", que mira el valor que hay en la cabeza de la cola, pero sin extraerlo.
- "Clear", que borra todo el contenido de la cola.
- "Contains", que indica si un cierto elemento está en la cola.
- "ToArray", que devuelve toda la cola convertida a un array.
- Al igual que en la pila, también tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

3.4.6 Las listas

Una lista es una estructura dinámica en la que se puede añadir elementos sin tantas restricciones. Es habitual que se puedan introducir nuevos datos en ambos extremos, así como entre dos elementos existentes, o bien incluso de forma ordenada, de modo que cada nuevo dato se introduzca automáticamente en la posición adecuada para que todos ellos queden en orden.

En el caso de C#, tenemos dos variantes especialmente útiles: una lista a cuyos elementos se puede acceder como a los de un array ("ArrayList") y una lista ordenada ("SortedList").

3.4.7 Array List

En un ArrayList, podemos añadir datos en la última posición con "Add", insertar en cualquier otra con "Insert", recuperar cualquier elemento usando corchetes, o incluso ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
1    using System;
2    using System.Collections;
3    
4    public class Ejemplo22
5    {
6        public static void Main()
7        {
8            ArrayList miLista
9              = new ArrayList();
10           // Añadimos en orden
11           miLista.Add("Hola,");
12           miLista.Add("soy");
13           miLista.Add("yo");
14   
15           // Mostramos lo que contiene
16           Console.WriteLine( "Contenido actual:");
17           foreach (string frase in miLista)
18               Console.WriteLine( frase );
19   
20           // Accedemos a una posición
21           Console.WriteLine( "La segunda palabra es: {0}",
22               miLista[1] );
23   
```

```
24          // Insertamos en la segunda posicion
25          miLista.Insert(1, "Como estas?");
26
27          // Mostramos de otra forma lo que contiene
28          Console.WriteLine( "Contenido tras insertar:");
29          for (int i=0; i<miLista.Count; i++)
30              Console.WriteLine( miLista[i] );
31
32          // Buscamos un elemento
33          Console.WriteLine( "La palabra \"yo\" está en la posición {0}",
34              miLista.IndexOf("yo") );
35
36          // Ordenamos
37          miLista.Sort();
38
39          // Mostramos lo que contiene
40          Console.WriteLine( "Contenido tras ordenar");
41          foreach (string frase in miLista)
42              Console.WriteLine( frase );
43
44          // Buscamos con búsqueda binaria
45          Console.WriteLine( "Ahora \"yo\" está en la posición {0}",
46              miLista.BinarySearch("yo") );
```

```
49          miLista.Reverse(); // Invertimos la lista
50
51          // Borramos el segundo dato y la palabra "yo"
52          miLista.RemoveAt(1);
53          miLista.Remove("yo");
54
55          // Mostramos nuevamente lo que contiene
56          Console.WriteLine( "Contenido dar la vuelta y tras eliminar dos:");
57          foreach (string frase in miLista)
58              Console.WriteLine( frase );
59
60          // Ordenamos y vemos dónde iría un nuevo dato
61          miLista.Sort();
62          Console.WriteLine( "La frase \"Hasta Luego\"...");
63          int posicion = miLista.BinarySearch("Hasta Luego");
64          if (posicion >= 0)
65              Console.WriteLine( "Está en la posición {0}", posicion );
66          else
67              Console.WriteLine( "No está. El dato inmediatamente mayor "+
68                  "es el {0}: {1}",
69                  ~posicion, miLista[~posicion] );
70      }
71  }
```

IV. Conclusiones

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y tres estructuras básicas: secuencia, selección (if y switch) e iteración (bucles for y while); asimismo, se considera innecesario y contraproducente el uso de la instrucción de transferencia incondicional (goto), que podría conducir a código espagueti, mucho más difícil de seguir y de mantener, y fuente de numerosos errores de programación.

Con posterioridad a la programación estructurada se han creado nuevos paradigmas tales como la programación modular, la programación orientada a objetos, la programación por capas y otras, así como nuevos entornos de programación que facilitan la programación de grandes aplicaciones y sistemas.

La asignatura de **Programación I** ha brindado las bases de la programación estructurada a estudiantes de Ingeniería en Sistemas I Año, en el año lectivo dos mil veinte. Esto servirá de pilar fundamental para las próximas clases de **Programación II** y **Programación III**, en la cual los estudiantes trabajarán con diferentes paradigmas de programación.

V. Bibliografía

Cosio, N. A. (2010). *Guía Total del Programador C#.* Fox Andina.

El Lenguaje de Programación. BRIAN W. KERNIGHAN & DENNIS M. RITCHIE. Editorial Pearson Education. Segunda Edición.

VI. Recursos Electrónicos

- https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/single-dimensional-arrays
- https://docs.microsoft.com/en-us/dotnet/api/system.array?view=netcore-3.1
- http://www.nachocabanes.com/csharp/curso2015/csharp05.php
- **https://platzi.com/clases/programacion-estructurada/**
- **https://platzi.com/clases/fundamentos-csharp/**

DOSSIER PROGRAMACIÓN I