

Departamento de Lenguajes y Sistemas Informáticos

Tema 7. Plataforma .net

Primeros programas con C#

Herramientas Avanzadas para el Desarrollo de Aplicaciones

Escuela Politécnica Superior
Universidad de Alicante

Programación en C#. Parte I: Primeros programas con C#

Objetivos

1. Introducción a la plataforma .NET
2. Conocer los orígenes de C#
3. Crear una aplicación de consola
4. Conocer la estructura básica de un programa C#
5. Conocer la gestión de excepciones
6. Conocer las colecciones de datos de llist

INDICE

1. ¿Qué es .net?
2. Arquitectura .net
3. Arquitectura .net Framework
4. Introducción a C#
5. Mi primer programa con C#
6. Aspectos básicos del lenguaje
7. Gestión de excepciones
8. Colecciones de datos

Introducción a .Net

1

¿Qué es .Net?

¿Qué es .net?

- .net es una plataforma que permite el desarrollo de aplicaciones software y librerías.
- .net contiene el compilador y las herramientas necesarias para construir, depurar y ejecutar estas aplicaciones.

¿Qué es .net? (II)

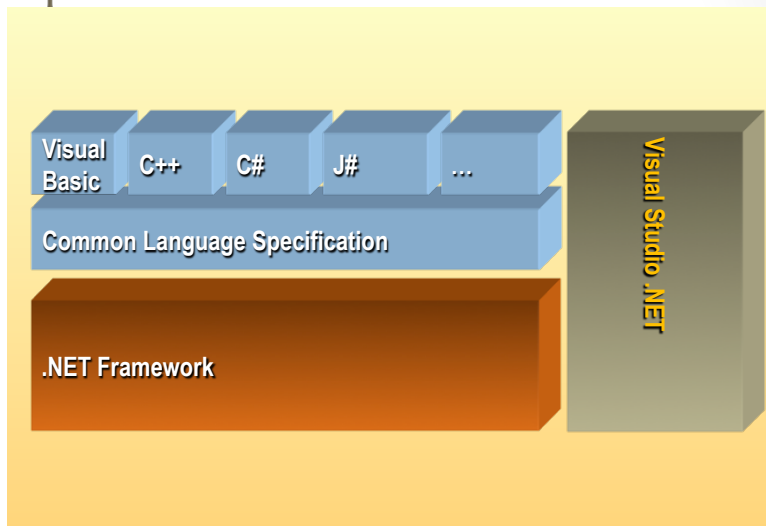
- .net es una plataforma software
- Es un entorno de desarrollo independiente del lenguaje, que permite escribir programas de forma sencilla, e incluso permite combinar código escrito en diferentes lenguajes.
- No está orientado a un Hardware/Sistema Operativo concreto, sino a cualquier plataforma para la que .net esté desarrollado.

Introducción a .Net

2

Arquitectura .Net

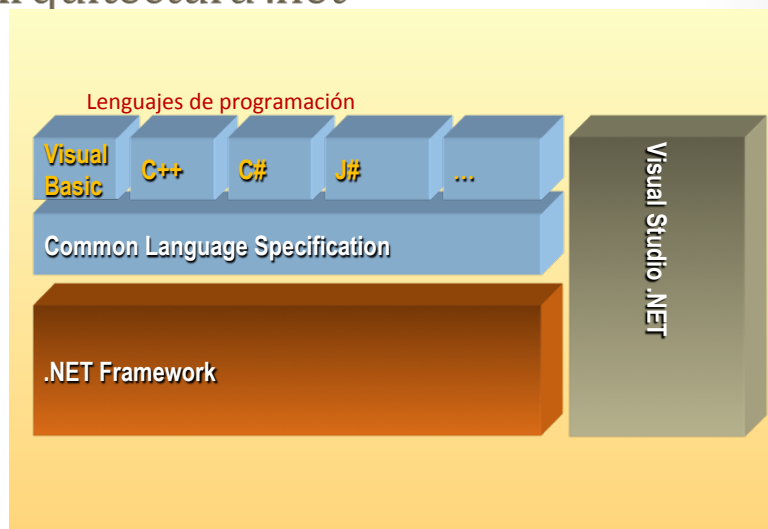
Arquitectura .net



Visual Studio .net

- **Visual Studio .NET** ofrece un entorno de desarrollo para desarrollar aplicaciones que se ejecutan sobre el .NET Framework.
- Proporciona las tecnologías fundamentales para simplificar la creación e implantación de
 - Aplicaciones y Servicios Web
 - Aplicaciones basadas en Windows

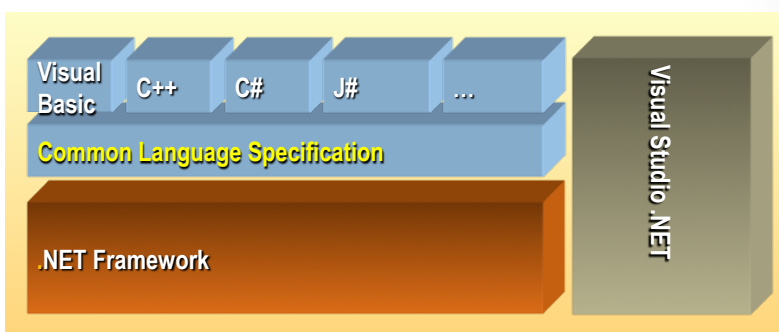
Arquitectura .net



Lenguajes de programación

- **C#**
 - C# ha sido diseñado específicamente para la plataforma .NET y es el primer lenguaje moderno orientado a componentes de la familia de C y C++.
 - Puede incrustarse en páginas ASP.NET.
 - Algunas de las principales características de este lenguaje incluyen clases, interfaces, delegados, espacios de nombres, propiedades, indexadores, eventos, sobrecarga de operadores, versionado, atributos, código inseguro, creación de documentación en formato XML
- Visual Basic .net
- C++
- J#
- Lenguajes de terceros

CLS

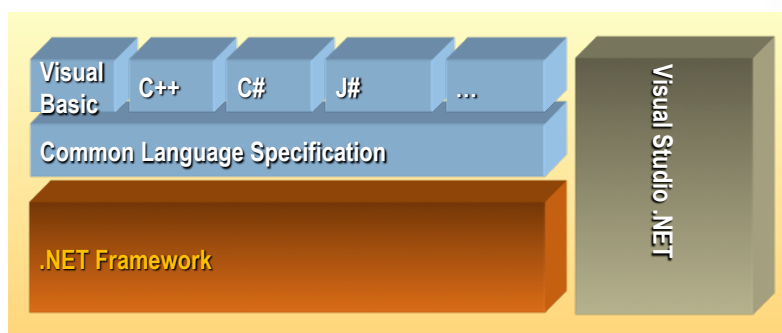


- La especificación **Common Language Specification (CLS)** define los mínimos estándares que deben satisfacer los lenguajes y desarrolladores si desean que sus componentes y aplicaciones sean ampliamente utilizados por otros lenguajes compatibles con .NET.

Common Language Specification (II)

- La especificación **CLS** permite a los desarrolladores de .NET crear aplicaciones como parte de un equipo que utiliza múltiples lenguajes con la seguridad de que no habrá problemas con la integración de los diferentes lenguajes.
- La especificación **CLS** también permite a los desarrolladores de .NET heredar de clases desarrolladas en lenguajes diferentes.

.net framework



- Es el motor de ejecución
- Proporciona un conjunto de servicios comunes para los proyectos generados en .net, con independencia del lenguaje.

.net Framework (II)

- **Extensible**
 - La jerarquía del .NET Framework no queda oculta al desarrollador. Podemos acceder y extender clases .NET (a menos que estén selladas) utilizando herencia. También podemos implementar herencia multilenguaje.
- **Fácil de usar por los desarrolladores**
 - En el .NET Framework, el código está organizado en espacios de nombres jerárquicos y clases. El Framework proporciona un sistema de tipos común, denominado sistema de tipos unificado, que utiliza cualquier lenguaje compatible con .NET. En el sistema de tipos unificado, todo es un objeto.

Introducción a .Net

3

Arquitectura
.net
Framework

Partes del .NET Framework.

El **Common Language Runtime (CLR)** es el núcleo de la plataforma .NET. Es el motor encargado de gestionar la ejecución de las aplicaciones para ella desarrolladas y a las que ofrece numerosos servicios que simplifican su desarrollo y favorecen su fiabilidad y seguridad.

Common Language Runtime (CLR)

- **Ejecución multiplataforma:** El CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET. Es decir, cualquier plataforma para la que exista una versión del CLR podrá ejecutar cualquier aplicación .NET.

Hasta ahora solo se han desarrollado CLR para todas las versiones de Windows, existe la posibilidad de desarrollar una versión para sistemas como Unix o Linux debido a que la arquitectura del CLR es abierta.

Proyecto Mono: <http://go-mono.com>



Common Language Runtime (CLR)

- **Integración de lenguajes:**

Desde cualquier lenguaje para el que exista un compilador que genere código para la plataforma .NET es posible utilizar código generado para la misma usando cualquier otro lenguaje tal y como si de código escrito usando el primero se tratase.

La integración de lenguajes provee que es posible escribir una clase en C# que herede de otra escrita en Visual Basic.NET que, a su vez, herede de otra escrita en C++ *con extensiones gestionadas*.

Microsoft Intermediate Language (MSIL)

- Los compiladores que generan código para la plataforma .NET generan código escrito en el lenguaje intermedio conocido como **Microsoft Intermediate Language (MSIL)**

Common Language Runtime (CLR)

- **Gestión de memoria:**

El CLR incluye un **recolector de basura** que evita que el programador tenga que tener en cuenta cuándo ha de destruir los objetos que dejen de serle útiles.

Gracias a este recolector se evitan errores de programación muy comunes como:

- Intentos de borrado de objetos ya borrados.
- Agotamiento de memoria por olvido de eliminación de objetos inútiles o
- Solicitud de acceso a miembros de objetos ya destruidos.

Common Language Runtime (CLR)

- **Seguridad de tipos:** El CLR facilita la detección de errores de programación difíciles de localizar comprobando que toda conversión de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que los tipos origen y destino sean compatibles.

Common Language Runtime (CLR)

- **Tratamiento de excepciones:** En el CLR todos los errores que se puedan producir durante la ejecución de una aplicación se propagan de igual manera: mediante *excepciones*.

Biblioteca de clases



System	System.Security	System.Runtime.InteropServices
System.Net	System.Text	System.Globalization
System.Reflection	System.Threading	System.Configuration
System.IO	System.Diagnostics	System.Collections

Biblioteca de clases

- La biblioteca de clases del .NET Framework expone características del entorno de ejecución y proporciona en una jerarquía de objetos otros servicios de alto nivel que todo programador necesita. Esta jerarquía de objetos se denomina **espacio de nombres**.
- La biblioteca de clases del .NET Framework proporciona numerosas y potentes características nuevas para los desarrolladores
 - Por ejemplo, el espacio de nombres Collections añade numerosas posibilidades nuevas, como clasificación, colas, pilas y matrices de tamaño automático.
 - La clase de sistema Threading también ofrece nuevas posibilidades para crear verdaderas aplicaciones multi-hilo.

Biblioteca de clases (II)

- **Espacios de nombres System**
 - El espacio de nombres System contiene clases fundamentales y clases base que definen tipos de datos valor y referencia comúnmente utilizados, eventos y descriptores de eventos, interfaces, atributos y procesamiento de excepciones.

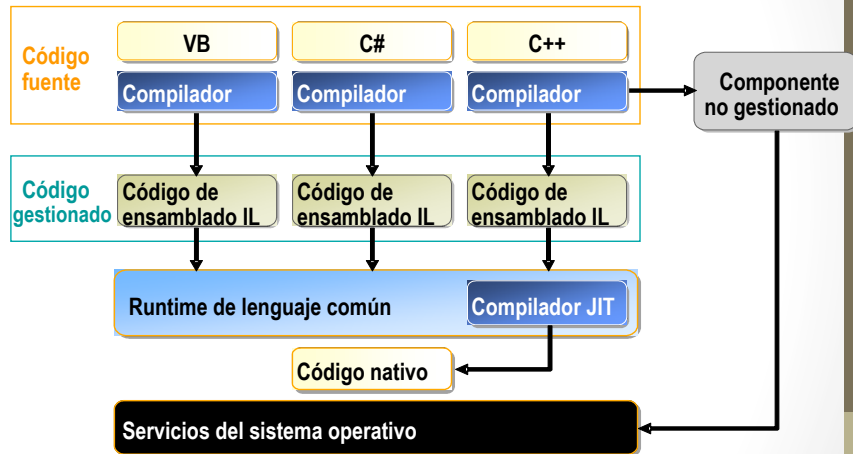
Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura ADO.NET.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Windows.Forms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.Xml	Acceso a datos en formato XML.

Introducción a .Net

4

El modelo de ejecución

El modelo de ejecución



Programación en C#

4

Introducción a C#

Introducción a C#

- C#
 - Lenguaje diseñado específicamente para .NET
 - Diseñado desde cero sin ningún condicionamiento
 - Microsoft lo describe como
 - Sencillo
 - Moderno
 - Orientado a objetos
 - Seguro en cuanto a tipos
 - Derivado de C y C++ (y a JAVA aunque Microsoft no lo diga)

Introducción a C# (II)

- Ventajas
 - Integrado con modernas herramientas de desarrollo
 - Fácil de integrar con Visual Basic
 - Tiene el alto rendimiento y permite el acceso a memoria de bajo nivel de C++

Programación en C#

5

Mi primer programa con C#

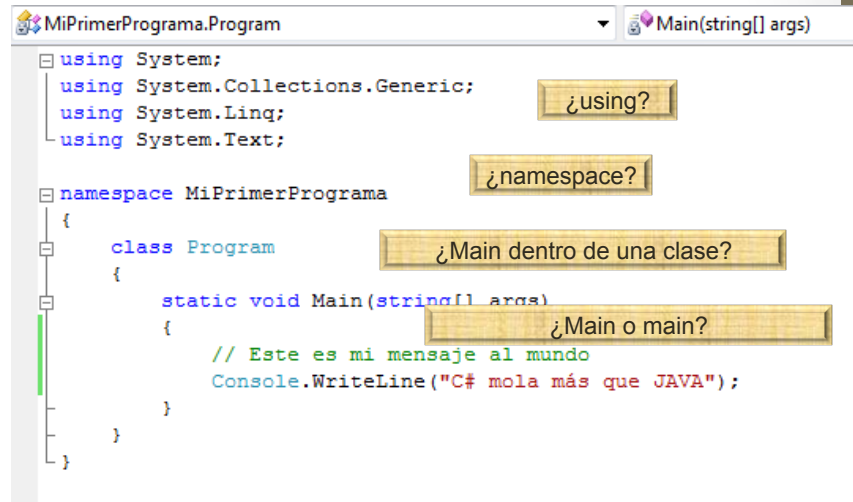


Ejercicio 1

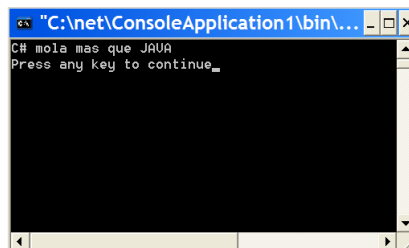
Primer Programa

- Escribir un programa en C# que muestre un mensaje en pantalla.

Demo : MiPrimerPrograma



Demo : MiPrimerPrograma



El programa principal : *Main*

- Al escribir Main hay que:
 - Utilizar una “M” mayúscula, como en “Main”
 - Designar un **Main** como el punto de entrada al programa
 - Declarar **Main** como **static void Main**
- Un Main puede pertenecer a múltiples clases
 - En ese caso se debe especificar cual es el primer punto de entrada
- La aplicación termina cuando Main acaba o ejecuta un return

La clase: *class*

- Una aplicación C# es una colección de clases, estructuras y tipos
- Una clase es un conjunto de datos y métodos
- Una aplicación C# puede incluir muchas clases
- Sintaxis :

```
class nombre
{
    ...
}
```

Los espacios de nombres :*namespace*

- .NET Framework ofrece muchas clases de utilidad
 - Organizadas en espacios de nombres (namespace)
- System es el espacio de nombres más utilizado
- Se hace referencia a clases por su espacio de nombres
- La sentencia using

```
System.Console.WriteLine("Hola, mundo");
```

```
using System;  
...  
Console.WriteLine("Hola, mundo");
```

Los espacios de nombres :*using*

- La palabra using referencia los espacios de nombres utilizados.
- En caso de no referenciar un espacio de nombres, al utilizar sus métodos se debe especificar el path completo.
- La sentencia using

```
System.Console.WriteLine("Hola, mundo");
```

```
using System;  
...  
Console.WriteLine("Hola, mundo");
```

Los comentarios

- Los comentarios son importantes
 - Una aplicación con los comentarios adecuados permite a un desarrollador comprender perfectamente la estructura de la aplicación
- Comentarios de una sola línea

```
// Obtener el nombre del usuario
Console.WriteLine("¿Cómo se llama? ");
name = Console.ReadLine( );
```

```
/* Encontrar la mayor raíz
de la ecuación cuadrática */
x = (...);
```

Los comentarios (II)

- **/// Comentarios que permiten generar documentación automática del proyecto.**

```
namespace ConsoleApplication11
{
    /// <summary>
    /// La clase 1 permite mostrar mensajes
    /// </summary>
    class Class1
    {
        /// <summary>
        /// Punto de entrada principal de la aplicación.
        /// </summary>
```

Programación en C#

6

Aspectos básicos
del lenguaje

Programación en C#

6.1

Sistema de Tipos
Comunes
(CTS)

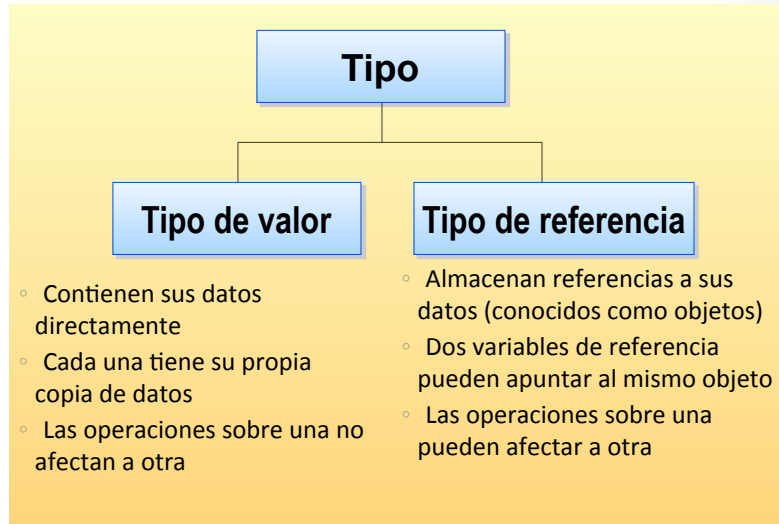
CTS

- ▶ Cada variable tiene un tipo de datos que determina los valores que se pueden almacenar en ella.
- ▶ C# es un lenguaje de especificaciones seguras (type-safe), lo que significa que el compilador de C# garantiza que los valores almacenados en variables son siempre del tipo adecuado.
- ▶ El runtime de lenguaje común incluye un sistema de tipos comunes (Common Type System, CTS) que define un conjunto de tipos de datos predefinidos que se pueden utilizar para definir variables.

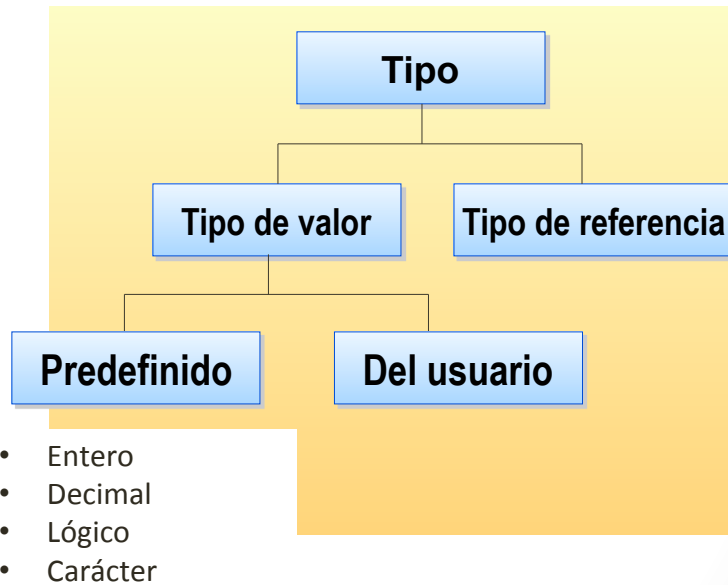
CTS (II)

- ▶ Al definir una variable es necesario elegir el tipo de datos correcto para ella. El tipo de datos determina los valores permitidos para esa variable, los cuales a su vez determinan las operaciones que se pueden efectuar sobre ella.
- ▶ El CTS es una parte integral del runtime de lenguaje común y es compartido por los compiladores, las herramientas y el propio runtime.
- ▶ Es el modelo que define las reglas que sigue el runtime a la hora de declarar, usar y gestionar tipos.
- ▶ El CTS establece un marco que permite la integración entre lenguajes, la seguridad de tipos y la ejecución de código con altas prestaciones.

CTS



CTS



Entero

sbyte	SByte	8 bits con signo
short	Int16	16 bits con signo
int	Int32	32 bits con signo
long	Int64	64 bits con signo
byte	Byte	8 bits sin signo
ushort	UInt16	16 bits sin signo
uint	UInt32	32 bits sin signo
ulong	UInt64	64 bits sin signo

El tipo byte no es equivalente al char.

Decimales

float	Single	32 bits. Precisión simple (7 dígitos significativos)
double	Double	64 bits. Precisión doble (15 dígitos significativos)
decimal	Decimal	Alta precisión (28 dígitos significativos)

Por defecto un número no entero es double

Explícitamente
34.5F (float)
34.5M (decimal)

Lógico

bool	Bool	<i>true o false</i>
------	------	---------------------

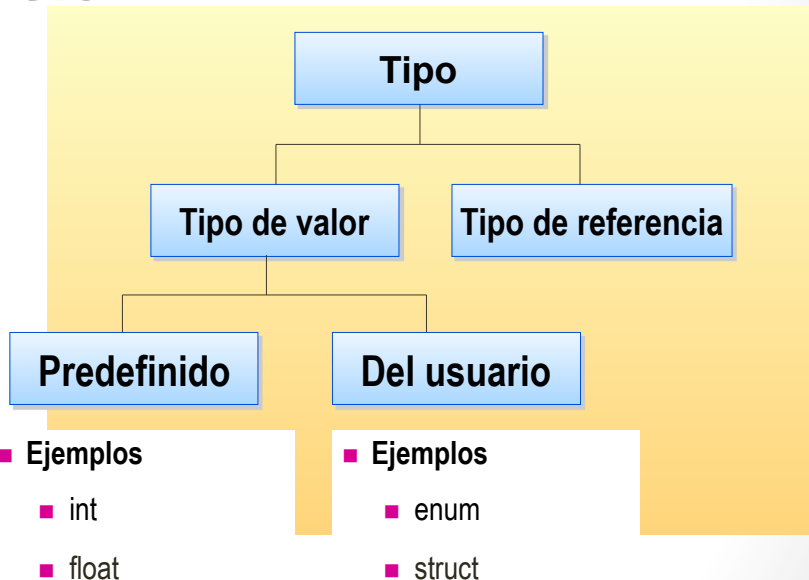
No se pueden convertir tipos bool a entero o viceversa.

Carácter

char	Char	<i>Carácter Unicode de 16 bits</i>
------	------	------------------------------------

Valores Unicode ‘\u0041’
Valores hexadecimales ‘\x0041’

CTS



Tipos enumerados definidos por el usuario :enum

```
enum Colores {amarillo, azul, rojo};
```

Definición

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

```
{
```

```
Colores colorPantalon = Colores.amarillo;
```

Uso

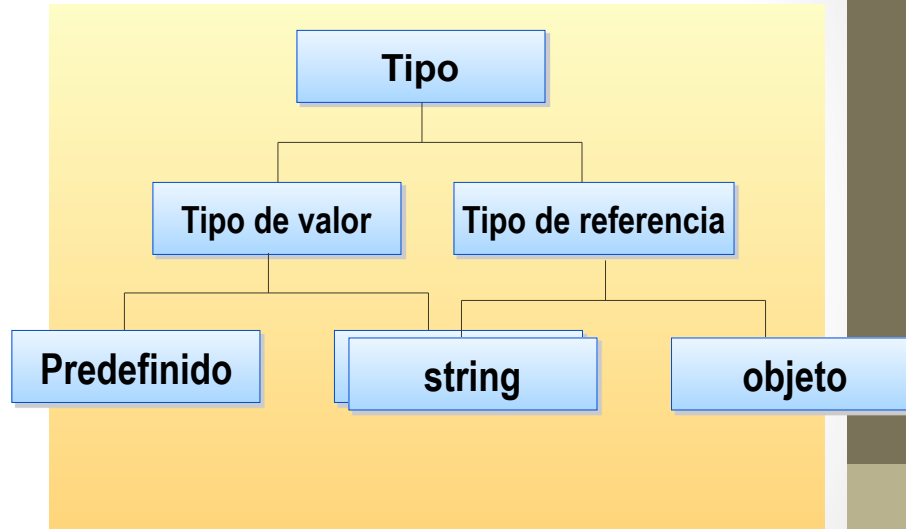
```
Console.WriteLine(colorPantalon);
```

Visualización

```
}
```

```
amarillo
```

CTS



Tipos referencia

- **Object**
 - Es un tipo del que derivan el resto de tipos
- **String**
 - Facilita la gestión de cadenas
 - Permite operaciones del tipo
 - Asignación directa
 - Concatenación (+)

El tipo string

```
static void Main(string[] args){
    string cadena= "Mi primera cadena";
    Console.WriteLine(cadena);
}
```

```
string cadena = "Mi primera cadena";
```

Mi primera cadena

```
string cadena = "Mi primera\ncadena";
```

Mi primera
cadena

```
string cadena = @"Mi primera\ncadena";
```

Mi primera\ncadena

El tipo string (2)

```
static void Main(string[] args){
    string cadena= "Mi primera cadena";
    Console.WriteLine(cadena);
}
```

```
string cadena = "\"Hola\"";
```

"Hola"

```
string cadena = "Hola " + 2;
```

Hola 2

```
string cadena = 2 + "Hola";
```

2Hola

Programación en C#

6.2

Expresiones y Operadores

Expresiones. Operadores

- Símbolos utilizados en las expresiones

Operadores	
<ul style="list-style-type: none"> • Incremento / decremento • Aritméticos • Relacionales • Igualdad • Lógicos • Asignación 	<pre> ++ -- * / % + - < > <= >= == != && ! = *= /= %= += -= </pre>



Ejercicio 2

Entrada por teclado

- Escribir un programa en C# que lea dos números por teclado.

Demo : Entrada por teclado

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MiPrimerPrograma
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            string cadena;
            Console.WriteLine("Introduce el primer número: ");
            cadena = Console.ReadLine();
            a = int.Parse(cadena);
            Console.WriteLine("Introduce el segundo número: ");
            cadena = Console.ReadLine();
            b = int.Parse(cadena);
        }
    }
}
```

Definición de variables

Salida por pantalla

Entrada por teclado y conversión de datos

Programación en C#

6.3

Variables y Constantes

Declaración de variables

- Se suelen declarar por tipo de dato y nombre de variable:

```
int objetoCuenta;
```

- --o--

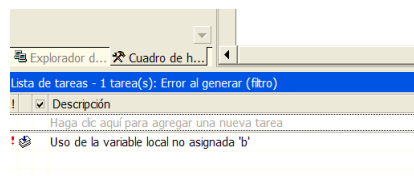
```
int objetoCuenta, empleadoNúmero;
```

```
int objetoCuenta,  
    empleadoNúmero;
```


Declaración de variables (II)

Pregunta. ¿Qué ocurre si se intenta acceder a una variable no inicializada?

```
static void Main(string[] args)
{
    int a,b;
    a = b + 2;
}
```



!!!! El compilador C# exige que cualquier variable esté inicializada antes de ser usada

Declaración de variables (III)

- C# es mucho más seguro que
 - C++ Deja al programador la tarea de garantizar que usa variables inicializadas. (Los compiladores modernos emiten mensajes de aviso pero permiten generar el ejecutable.
 - VB inicializa a 0 por defecto las variables.

Declaración de variables (IV)

- ▶ Reglas
 - Usa letras, el signo de subrayado y dígitos
- ▶ Recomendaciones
 - Evita poner todas las letras en mayúsculas
 - Evita empezar con un signo de subrayado
 - Evita el uso de abreviaturas
 - Use PascalCasing para nombres con varias palabras

Declaración de variables (V)

- ▶ *PascalCasing*
 - ▶ Consiste en marcar con mayúsculas las fronteras de las palabras
 - ▶ EstaEsUnaVariable
 - ▶ La primera letra es maúscula
- ▶ *CamelCasing* es lo mismo pero iniciando la palabra en minúscula
 - ▶ estaEsUnaVariable
- ▶ Hay autores que recomiendan utilizar PascalCasing variables públicas y camelCasing para privadas.

Declaración de variables (VI)

- Asignar valores a variables ya declaradas:

```
int empleadoNumero;  
empleadoNumero = 23;
```

- Inicializar una variable cuando se declara:

```
int empleadoNumero = 23;
```

- También es posible inicializar valores de caracteres:

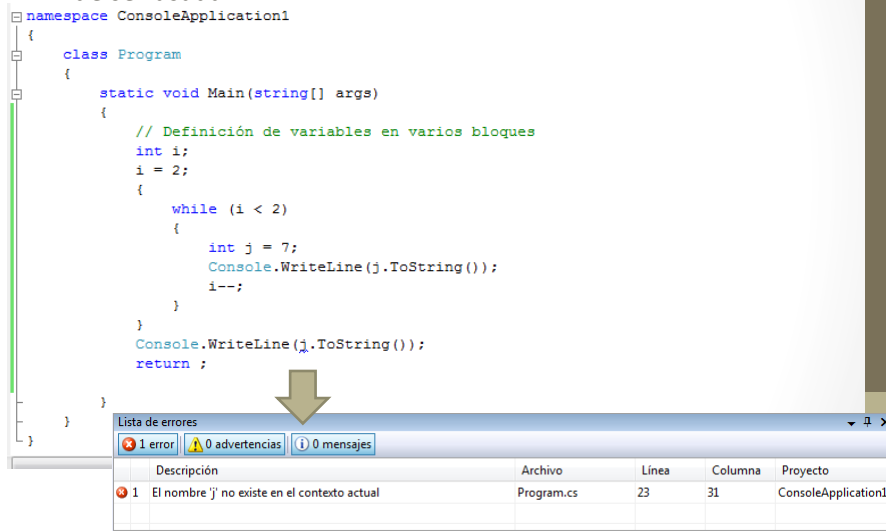
```
char inicialNombre = 'J';
```

Ámbito de las variables

- El ámbito de una variable coincide con su clase contenedora
- El ámbito de una variable finaliza con la llave que cierra el bloque o método en el que la variable ha sido declarada.
- Las variables pueden ser declaradas dentro de un bucle, siendo solo visibles en el mismo.
 - Esto cumple el estándar ANSI de C++
 - Las versiones anteriores de Microsoft C++ no lo cumplían

Ámbito de las variables (II)

- El compilador C# exige que cualquier variable esté inicializada antes de ser usada



```

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Definición de variables en varios bloques
            int i;
            i = 2;
            {
                while (i < 2)
                {
                    int j = 7;
                    Console.WriteLine(j.ToString());
                    i--;
                }
            }
            Console.WriteLine(j.ToString());
            return ;
        }
    }
}

```

Lista de errores

Descripción	Archivo	Línea	Columna	Proyecto
1 El nombre 'j' no existe en el contexto actual	Program.cs	23	31	ConsoleApplication1

Constantes

- Un constante es una variable que no puede cambiar su valor
- Se definen como las variables (con la palabra *const*)
- Se deben inicializar obligatoriamente en la inicialización

```
const int valor = 6378;
```

Constantes en C# (II)

- Las constantes en C#
 - El valor de la constante debe ser calculado en tiempo de compilación
 - (No se puede asignar a partir de una variable)
 - Esto se gestiona con variables del tipo readonly

Programación en C#

6.4

Entrada/Salida Por Consola

Entrada/salida de consola

- Permite acceder a las secuencias estándar de entrada, salida y error
- Sólo tiene sentido para aplicaciones de consola
 - Entrada estándar: teclado
 - Salida estándar: Pantalla
 - Error estándar: Pantalla

Ojo: La clase Console solo se debe utilizar para aplicaciones en línea de comandos. Para aplicaciones windows se debe utilizar el espacio de nombre System.Windows.Forms

Entrada/salida de consola (II)

- Se basan en el uso de la clase Console
- `Console.Read()`
 - Lee un flujo de entrada y lo devuelve como un *int*
- `Console.ReadLine()`
 - Lee una cadena de texto entera
- `Console.Write()`
 - Permite escribir en la pantalla
- `Console.WriteLine`
 - Escribe en pantalla añadiendo carácter retorno de carro o fin de línea

Entrada/salida de consola (III)

```
static void Main(string[] args)
{
    int a, b;
    string cadena;
    Console.Write ("Introduce el primer número: ");
    cadena = Console.ReadLine();
    a = int.Parse(cadena);
    Console.Write("Introduce el segundo número: ");
    cadena = Console.ReadLine();
    b = int.Parse(cadena);
}
```

Parámetros del writeline (II)

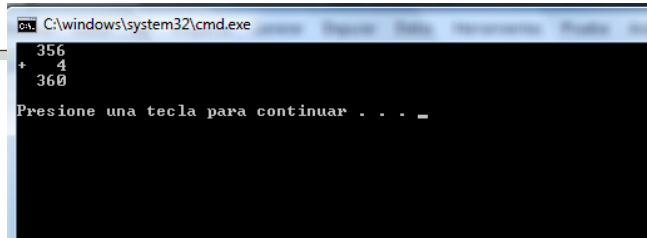
- Cadena que contiene entre llaves los parámetros que se indican a continuación separados por comas

```
static void Main(string[] args)
{
    int a, b;
    a = 3;
    b = 4;
    Console.WriteLine("La suma de {0} y {1} es {2} ",a,b, a+b);
    Console.WriteLine("La suma de " + a + " y " + b + " es " + (a+b));
}
```

Formateo de salida

- La salida se puede formatear utilizando el formato {n.m} donde n es el índice del parámetro y w el ancho de la salida

```
static void Main(string[] args)
{
    int a, b;
    a = 356;
    b = 4;
    Console.WriteLine(" {0,4}\n+{1,4}\n {2,4}\n",a,b, a+b);
}
```

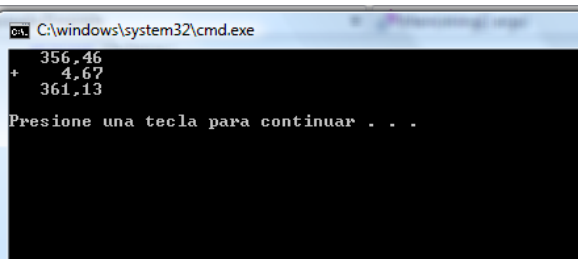


```
C:\windows\system32\cmd.exe
356
+ 4
360
Presione una tecla para continuar . . .
```

Precisión

- Se pueden indicar los decimales utilizando F y un número

```
static void Main(string[] args)
{
    double a, b;
    a = 356.459;
    b = 4.67;
    Console.WriteLine(" {0,8:F2}\n+{1,8:F2}\n {2,8:F2}\n",a,b, a+b);
}
```



```
C:\windows\system32\cmd.exe
356.46
+ 4.67
361.13
Presione una tecla para continuar . . .
```




Ejercicio 3

Suma y Cuadrado de dos números

- Escribir un programa en C# que lea dos números por teclado. Luego deberá mostrar el resultado de la suma de ambos y el cuadrado de dicha suma.

Demo : Entrada por teclado

```
static void Main(string[] args)
{
    int a, b;
    decimal suma;
    float cuadrado;
    string cadena;
    Console.Write("Introduce el primer número: ");
    cadena = Console.ReadLine();
    a = int.Parse(cadena);
    Console.Write("Introduce el segundo número: ");
    cadena = Console.ReadLine();
    b = int.Parse(cadena);
    suma = a + b;
    cuadrado = suma * suma;
}
```

Conversión de datos sin pérdida de información

Conversión de datos con pérdida de información

Lista de errores

1 error 0 advertencias 0 mensajes

Descripción

1 No se puede convertir implícitamente el tipo 'decimal' en 'float'. Ya existe una conversión explícita (compruebe si le falta una conversión)

Programación en C#

6.5

Conversión de Tipos

Conversión de tipos

- Implícita
 - Automática, cuando no hay posible pérdida de información.

```
int x = 2;  
long l = 234;  
double dob = 45.67;  
  
dob = x;  
l = x;
```

Conversión de tipos (II)

- Explícita. Se debe indicar que se desea realizar una conversión en la que puede haber pérdida de información.

```
int x = 2;
long l = 234;
double dob = 45.67;

x = l;
l = dob;
```

Lista de tareas - 2 tarea(s): Error al generar (filtro)

!	✓	Descripción
		Haga clic aquí para agregar una nueva tarea
!		No se puede convertir implícitamente el tipo 'long' a 'int'
!		No se puede convertir implícitamente el tipo 'double' a 'long'

Conversión de tipos (III)

- Explícita. Correcta utilizando *casting*

```
int x = 2;
long l = 234;
double dob = 45.67;

x = (int) l;
l = (long) dob;
```

Conversión de tipos (IV)

- ¿Es correcto el siguiente programa?

```
static void Main(string[] args)
{
    float numero;
    numero = 28.67;
}
```

- ✖ 1 El literal de tipo double no se puede convertir implícitamente en el tipo 'float'; utilice un sufijo 'F' para crear un literal de este tipo

```
static void Main(string[] args)
{
    float numero;
    numero = 28.67F;
}
```

Demo : Entrada por teclado

```
namespace MiPrimerPrograma
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            decimal suma;
            float cuadrado;
            string cadena;
            Console.WriteLine("Introduce el primer número: ");
            cadena = Console.ReadLine();
            a = int.Parse(cadena);
            Console.WriteLine("Introduce el segundo número: ");
            cadena = Console.ReadLine();
            b = int.Parse(cadena);
            suma = a + b;
            cuadrado = (float) (suma * suma);
        }
    }
}
```

Conversión de datos

Programación en C#

6.6

Estructuras de control

Estructuras de control

- Condicional
 - if/else
 - switch/case
- Repetición
 - while
 - do/while
 - for
 - foreach
- Cambio de secuencia
 - return
 - break
 - continue

Condicional Simple

□ if

```
if (saldo > reintegro)
    Console.WriteLine("OK");
```

□ if else

```
if (saldo > reintegro)
    Console.WriteLine("OK");
else
    Console.WriteLine("No OK");
```

□ if else if

```
if (saldo > reintegro)
    Console.WriteLine("OK");
else if (saldo < reintegro)
    Console.WriteLine("No OK");
else
    Console.WriteLine("a 0");
```

Condicional Simple (II)

¿Cómo funcionará?

```
if (saldo = reintegro)
    Console.WriteLine("a 0");
else if (saldo < reintegro)
    Console.WriteLine("No OK");
else
    Console.WriteLine("OK");
```

Lista de tareas - 1 tarea(s): Error al generar (filtro)

☒ Descripción

Haga clic aquí para agregar una nueva tarea

No se puede convertir implícitamente el tipo 'int' a 'bool'

SOLUCIÓN

```
if (saldo == reintegro)
    Console.WriteLine("a 0");
else if (saldo < reintegro)
    Console.WriteLine("No OK");
else
    Console.WriteLine("OK")
```

Condicional múltiple

```
switch (saldo)
{
    case 1 : saldo *= 2;
        break;
    case 2 : saldo *= 3;
        break;
    default : saldo = 0;
        break;
}
```

BREAK. Casi obligatorio.

Condicional múltiple (II)

```
switch (saldo) {
    case 1 : saldo *= 2;
    case 2 : saldo *= 3;
        break;
    default : saldo = 0;
        break;
}
```

Error

SOLUCIÓN

```
switch (saldo) {
    case 1 : saldo *= 2;
        goto case 2;
    case 2 : saldo *= 3;
        break;
    default : saldo = 0;
        break;
}
```

Condicional múltiple (III)

```
switch (saldo) {  
    case 1 :  
  
    case 2 : saldo *= 3;  
        break;  
    default : saldo = 0;  
        break;  
}
```

Correcto

Condicional múltiple (IV)

Permite utilizar etiquetas tipo string

```
string nombre= "Pedro";  
switch (nombre)  
{  
    case "Pedro" : Console.WriteLine("Hola Pedro");  
        break;  
    case "Juan" : Console.WriteLine("Hola Juan");  
        break;  
}
```


Estructuras de repetición

□ for

```
for (i = 0; i <= 10 ; i++)  
    suma+= i;
```

□ while

```
while ( i <= 10)  
{  
    suma+= i;  
    i++;  
}
```

□ do

```
i = 0;  
do  
{  
    suma += i;  
    i++;  
}  
while (i < 10);
```

Estructuras de repetición (II)

□ foreach

```
int [ ] mivector = {1,2,3};  
foreach(int algo in mivector)  
    Console.Write (algo);
```

Permite recorrer todos los elementos de un contenedor

Es necesario definir una variable del tipo de los elementos del contenedor

Cambio de secuencia:return

- Return devuelve el control a la rutina llamante a la actual

```
int valor, suma;
Console.Write("Introduce un número : ");
valor = int.Parse(Console.ReadLine());
suma = 0;
while (valor <= 10)
{
    if (valor == 3)
        return;
    suma += valor;
    valor++;
}

Console.WriteLine("El valor es " + suma);
```

¿¿¿ Qué hace el programa ???

Cambio de secuencia:break

- Break sale de la actual estructura de bucle

```
int valor, suma;
Console.Write("Introduce un número : ");
valor = int.Parse(Console.ReadLine());
suma = 0;
while (valor <= 10)
{
    if (valor == 3)
        break;
    suma += valor;
    valor++;
}

Console.WriteLine("El valor es " + suma);
```

¿¿¿ Qué hace el programa ???

Cambio de secuencia:continue

- Continue: Obliga a ejecutar la siguiente iteración del bucle

```
int valor, suma;
Console.Write("Introduce un número : ");
valor = int.Parse(Console.ReadLine());
suma = 0;
while (valor <= 10)
{
    valor++;
    if (valor == 3)
        continue;

    suma += valor;
}

Console.WriteLine("El valor es " + suma);
```

¿¿¿ Qué hace el programa ???

Programación en C#

7

Excepciones

Excepciones

- C# ofrece facilidades para la gestión de errores por medio del manejo de excepciones.
- Una excepción es un objeto que se crea cuando se produce una situación de error específica.
- Además el objeto contiene información que permite resolver el problema

Excepciones (II)

- Dos clases importantes de excepciones
 - `System.SystemException` Tienen naturaleza muy general y pueden ser lanzadas por cualquier aplicación.
 - `System.ApplicationException` Clase base de cualquier clase de excepción definida por terceros

Excepciones (III)

- La programación de excepciones se define

```
try
{
    // Código de ejecución normal
}
Catch
{
    // Gestión de errores
}
Finally
{
    // Liberación de recursos
}
```

Excepciones (IV)

- ▶ El bloque try contiene el código que forma parte del funcionamiento normal del programa
- ▶ El bloque catch contiene el código que gestiona los diversos errores que se puedan producir
- ▶ El bloque finally contiene el código que libera los recursos. Es un campo opcional.

Excepciones (V)

- Las excepciones funcionan de la siguiente forma
 1. Ejecutar instrucciones de bloque try
 1. Si hay error ir a bloque catch (2)
 2. Si no hay error ir a bloque finally (3)
 2. Ejecutar instrucciones del bloque catch
 3. Ejecutar bloque finally
 4. Fin de programa

Programación en C#

8

Colecciones de
datos

Colecciones de datos

- En .NET Framework existen tres tipos principales de colecciones:
 - Las colecciones basadas en **ICollection**
 - Las colecciones basadas en la **interfaz IList**
 - Las colecciones basadas en la **interfaz IDictionary**
- La diferencia básica entre estos tipos de colecciones es **cómo están almacenados los elementos que contienen**, por ejemplo, las colecciones de tipo **IList** (y las directamente derivadas de **ICollection**) solo almacenan un valor, mientras que las colecciones de tipo **IDictionary** guardan un valor y una clave relacionada con dicho valor.

Colecciones basadas en IList

- La **interfaz IList** se utiliza en las colecciones a las que queremos acceder mediante un índice, por ejemplo, los **arrays** realmente están basados en esta interfaz, y la única forma que tenemos de acceder a los elementos de un array, (y por extensión a los elementos de las colecciones basadas en **IList**), es mediante un índice numérico.

Colecciones basadas en IList

- Existen tres tipos principales de colecciones que implementan esta interfaz:
 - **Las de solo lectura**, colecciones que no se pueden modificar. Este tipo de colecciones suelen basarse en la clase abstracta *ReadOnlyCollectionBase*.
 - **Las colecciones de tamaño fijo**, no se pueden quitar ni añadir elementos, pero sí modificarlos. Por ejemplo, las colecciones basadas en *Array* son de tamaño fijo.
 - **Las de tamaño variable** permiten cualquier tipo de adición, eliminación y modificación. La mayoría de las colecciones suelen ser de este tipo, es decir, nos permiten dinámicamente añadir o eliminar elementos.

Colecciones basadas en IList

- Existe un gran número de colecciones en .NET que implementan esta interfaz, (sobre todo las colecciones basadas en controles), entre las que podemos destacar las siguientes:
- **ArrayList**, la colección "clásica" para este tipo de interfaz. Contiene todos los miembros habituales en este tipo de colecciones.
- **CollectionBase**, una clase abstracta para poder crear nuestras propias colecciones basadas en *IList*.
- **StringCollection**, una colección especializada que solo puede contener valores de tipo cadena.

Colección Arraylist

- Esta es una clase que representa una lista de datos.
- El ArrayList puede aumentar o disminuir su tamaño dinámicamente de una manera eficiente. Con un array de datos no era posible aumentar la capacidad del vector ya que dicho parámetro es especificado en el momento de crear la instancia del objeto.
- El ArrayList a diferencia, brinda la posibilidad de aumentar o disminuir su tamaño dinámicamente según sea necesario.

Colección Arraylist

- Al igual que ocurre con los arrays, el índice inferior es siempre el cero y los elementos se almacenan de forma consecutiva, es decir, si añadimos dos elementos a una colección de tipo ArrayList (y a las que implementen la interfaz IList), el primero ocupará la posición cero y el segundo la posición uno.
- Para crear una instancia de este objeto, se debe utilizar la clase ArrayList incluida en el espacio de nombre **System.Collections** como se muestra a continuación.
- **ArrayList arrayList=new ArrayList();**

Colección ArrayList

- El constructor de la clase ArrayList acepta también un parámetro tipo entero que indica la capacidad inicial del objeto que se esta creando.
- Si es necesario agregar un objeto a la colección, se debe utilizar el método **Add**, el cual inserta el nuevo elemento en la última posición, o el método **Insert** el cual lo inserta en la posición indicada.

Colección ArrayList

```
//ArrayList
Console.WriteLine("ArrayList");
ArrayList arrayList = new ArrayList();
arrayList.Add("hola1");
arrayList.Add("hola2");
arrayList.Add("hola3");
arrayList.Add("hola4");
arrayList.Add("hola5");
arrayList.Add("hola6");
arrayList.Add("hola7");
arrayList.Add("hola8");
arrayList.Add("hola9");
```

Colección Arraylist

- **Todos los objetos almacenados en un Arraylist son tratados como objetos**, por lo tanto es posible agregar todo tipo de datos, es decir, se puede agregar enteros, cadenas de texto, objetos de clases propias, etc.
- Y a diferencia de los array, **no todos los elementos deben ser del mismo tipo de dato**. Esto en algunas ocasiones puede ser una ventaja ya que permite almacenar gran variedad de información en una sola colección, sin embargo, por razones de rendimiento (cast, boxing, unboxing; convertir un tipo por valor en uno por referencia cuando va a guardarlo en la colección (boxing), y el proceso inverso cuando lo queremos recuperar (unboxing)), hay ocasiones en las que es preferible utilizar las colecciones genéricas.

Colección Arraylist

- Si es necesario quitar elementos de la colección, se debe usar el método **remove**, **removeAt** o **RemoveRange**, los cuales eliminan el objeto pasado como parámetro, o un elemento en una posición específica, o un grupo de elementos respectivamente.

Colección Arraylist

- Las propiedades más utilizadas de esta colección son : **Count y Capacity**.
- La primera sirve para conocer la cantidad actual de elementos que contiene la colección.
- La segunda indica la capacidad máxima actual de la colección para almacenar elementos. Es necesario tener presente que la capacidad de la colección, aumenta en caso de ser necesario al insertar un elemento, con lo que se garantiza el redimensionamiento automático.

Colección Arraylist

- La capacidad de una colección nunca podrá ser menor a la cantidad total de elementos contenidos, por lo que si se modifica manualmente la propiedad Capacity y se le asigna un valor menor que el valor devuelto por la propiedad Count, obtendremos una excepción de tipo **ArgumentOutOfRangeException** .

```
//ArrayList
ArrayList arrayList = new ArrayList();
arrayList.Add("hola1");
arrayList.Add("hola2");
arrayList.Add("hola3");
arrayList.Add("hola4");
arrayList.Add("hola5");
arrayList.Add("hola6");
arrayList.Add("hola7");
arrayList.Add("hola8");
arrayList.Add("hola9");

Console.WriteLine("Tamaño: " + arrayList.Count);
Console.WriteLine("Capacidad: " + arrayList.Capacity);

arrayList.Remove("hola1");

Console.WriteLine("Tamaño despues de eliminar 1 elemento: " + arrayList.Count);
Console.WriteLine("Capacidad despues de eliminar 1 elemento: " + arrayList.Capacity);

arrayList.Capacity -= 1;
Console.WriteLine("Capacidad tras disminuirla manualmente: " + arrayList.Capacity);
```

ex C:\WINDOWS\system32\cmd.exe

```
Tamaño: 9
Capacidad: 16
Tamaño despues de eliminar 1 elemento: 8
Capacidad despues de eliminar 1 elemento: 16
Capacidad tras disminuirla manualmente: 15
Presione una tecla para continuar . . .
```

Colección ArrayList

- Para acceder a los elementos contenidos por la colección se puede hacer mediante el uso de índices o mediante la instrucción foreach.

```
foreach (string s in arrayList)
{
    Console.WriteLine(s);
}

for (int i = 0; i < arrayList.Count; i++)
{
    Console.WriteLine(arrayList[i]);
}
```

Cuál elegir?

- Determinar qué tipo de colección usar en un caso específico, es tarea del desarrollador y se debe evaluar las condiciones para determinar la manera más eficiente de administrar los recursos.
- Si es un escenario donde no conocemos el tamaño que tendrá la colección y si además será muy probable que el tamaño varíe, entonces será recomendable bajo todas las demás circunstancias usar un **ArrayList** en lugar de un array debido a que el ArrayList brinda la posibilidad de redimensionarlo automáticamente.
- Sin embargo, para escenarios donde se conoce de antemano la cantidad total de elementos a almacenar y si todos son del mismo tipo, se debe usar el **array convencional** ya que los objetos son almacenados en su tipo de datos nativo y no es necesario hacer conversiones.