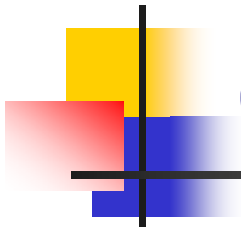
Tema 2: Clase y objetos en C#

Programación Orientada a Objetos

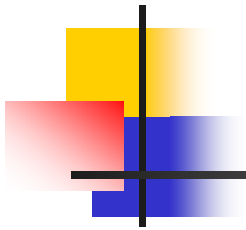
Curso 2008/2009

Begoña Moros Valle



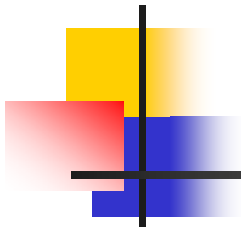
Contenido

- Introducción.
- Clases.
- Propiedades.
- Visibilidad.
- Espacios de nombres.
- Ensamblados.
- Tipos del lenguaje.
- Construcción de objetos.
- Estructuras.
- Asignación y copia.
- Identidad e igualdad.
- Métodos y mensajes.
- Paso de parámetros.
- Operadores.
- Instancia actual.
- Método Main.



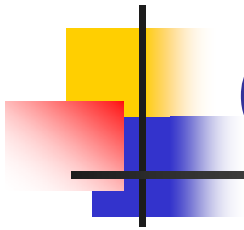
Introducción

- C# es un lenguaje creado por **Microsoft** y liderado por **Anders Heljsberg**.
- Es un **lenguaje orientado a objetos puro** inspirado en C++, Java, Delphi y Eiffel.
- Las aplicaciones C# son ejecutadas en un entorno controlado llamado **CLR** (*Common Language Runtime*).
- El lenguaje está **estandarizado** en ECMA e ISO.
- Actualmente está en la versión 3.0.



Plataforma .NET

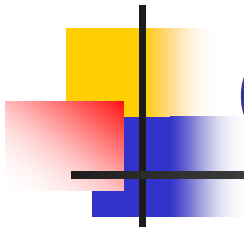
- El compilador de C# genera **código intermedio** para la **plataforma .NET**.
- El código intermedio es ejecutado por una máquina virtual: **CLR**
- C# es sólo uno de los lenguajes de la plataforma .NET: C++, VB.NET, Eiffel.NET, etc.
- La plataforma .NET está ligada a los sistemas operativos **Windows**.
- **Proyecto Mono**:
  - Implementación de .NET en otros sistemas operativos.
  - Incluye un compilador para C#.



Clases

- En C# los elementos que definen una clase son:
  - **Atributos**, **métodos** y **constructores** (= Java y C++)

- La declaración de una clase comparte aspectos en común con Java y C++:
  - La declaración de una clase incluye la definición e implementación (= Java).
  - Un fichero de código fuente (extensión .cs) puede contener la declaración de varias clases (= C++).



Clases

- C# añade dos nuevos tipos de declaraciones:
  - **Propiedades**:
    - Representan características de los objetos que son accedidas como si fueran atributos.
    - Ocultan el uso de métodos get/set.
    - Una propiedad puede representar un atributo calculado.
  - **Eventos**:
    - Notificaciones que envía un objeto a otros objetos cuando se produce un cambio de estado significativo.
- Propiedades y eventos son el soporte para el **Desarrollo de Software basado en Componentes**.

Clase Cuenta 1/4

```
public class Cuenta {
        // Constante
        private const int MAX_OPERACIONES = 20;

              // Atributo de clase
        private static int ultimoCodigo = 0;

              // Atributos de instancia
        private int codigo;
        private double saldo = 100;
        private readonly Persona titular;
        private EstadoCuenta estado;
        private double[] ultimasOperaciones;
        …
```

Clase Cuenta 2/4

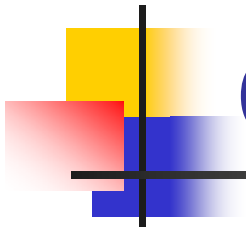
```
public class Cuenta
{
      …
      // Constructor
      public Cuenta(Persona titular, double saldo)
      {
          this.codigo = ++ultimoCodigo;
          this.titular = titular;
          this.saldo = saldo;
          estado = EstadoCuenta.OPERATIVA;
          ultimasOperaciones = new double[MAX_OPERACIONES];
      }
      …
}
```

Clase Cuenta 3/4

```
public class Cuenta
{       …
                // Propiedades
        public double Saldo
        {
            get { return saldo; }
        }
        public Persona Titular
        {
            get { return titular; }
        }
        public int Codigo
        {
            get { return codigo; }
        }
}
```

Clase Cuenta 4/4

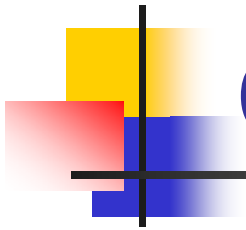
```
public class Cuenta
{ …
                // Métodos de instancia
        public void Ingreso(double cantidad) {
            saldo = saldo + cantidad;
        }
        public void Reintegro(double cantidad){
            if (cantidad <= saldo)
                saldo = saldo - cantidad;
        }

        // Métodos de clase
        public static int GetNumeroCuentas() {
            return ultimoCodigo;
        }
    }
```



Clases

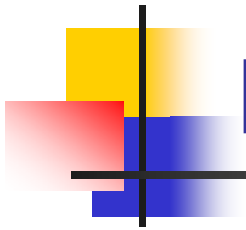
- Los **miembros** de una clase pueden ser de **instancia** (por defecto) o de **clase**, utilizando el modificador **`static`** (= Java y C++).

- Los **atributos de sólo lectura** se marcan utilizando el modificador **`readonly`** (= `final` de Java y `const` de C++):
  - `readonly Persona titular;`



Clases

- Las **constantes** se declaran `const` (= `final static` de Java y `const static` de C++):
  - `private const int MAX_OPERACIONES = 20;`
- Está permitida la **inicialización de los atributos** en la declaración (= Java):
  - `private double saldo = 100;`
- Los **atributos no inicializados** en la declaración o en los constructores toman el valor por defecto de su tipo de datos (= Java).



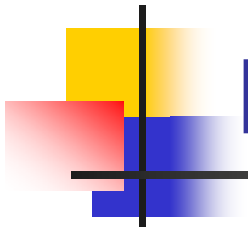
Propiedades

- **Declaración** de propiedades:

```
public double Saldo
{
    get { return saldo; }
}
```

- Se usan como atributos, pero el acceso se realiza invocando a métodos get/set:

```
Console.WriteLine("Saldo de la cuenta: " + cuenta.Saldo);
```



Propiedades

- Los métodos get/set pueden realizar cálculos:

```
public double SaldoDolar
{
   get { return saldo * Banco.CambioDolar(); }
}
```

- El acceso a la propiedad oculta el cálculo:

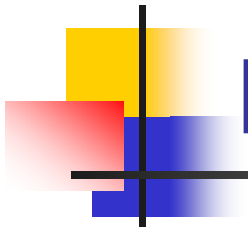
```
Console.WriteLine("Saldo en dólares: " + cuenta.SaldoDolar );
```

Propiedades

- En la definición de un método set, el identificador **`value`** representa el valor que va a ser asignado:

```
public double Saldo
{
    get { return saldo; }
    private set { saldo = value; }
}
```

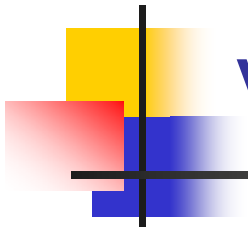
- Es posible indicar un nivel de visibilidad distinto para cada uno de los métodos.



Propiedades

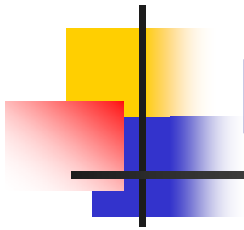
- **Declaración automática de propiedades**:
  - Evitamos tener que declarar el atributo.
  - Los métodos get/set sólo consultan y modifican la propiedad.

```
public double Saldo
{
      get;
      private set;
}
```



Visibilidad

- El **nivel de visibilidad** se especifica para cada declaración (= Java):
  - **public**: visible para todo el código.
  - **private**: visible sólo para la clase.
  - **protected**: visibilidad para la clase y los subtipos.
  - **internal**: visibilidad para el “ensamblado”.
  - **protected internal**: visibilidad para la clase y subtipos dentro del mismo ensamblado.
- Por defecto, las declaraciones en una clase son privadas (= C++).



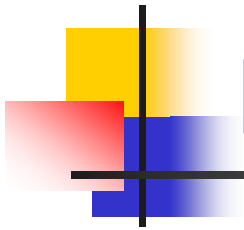
Espacios de nombres

- Un espacio de nombres (**`namespace`**) es un mecanismo para agrupar un conjunto de declaraciones de tipos relacionadas (= C++)
- **Evita la colisión de los nombres** de identificadores.
- Se declaran con **`namespace`** y pueden estar definidos en varios ficheros de código fuente.
- Los espacios de nombres pueden estar **anidados**.
- Son diferentes a los paquetes de Java.

Espacios de nombres

- Para hacer uso de un tipo declarado en un espacio de nombre se califica su nombre:
  - `GestionCuentas.Cuenta`
- Podemos indicar que se usan todas las declaraciones de un espacio de nombres con `using`.

```
using System;
using System.Text;

namespace GestionCuentas
{
    enum EstadoCuenta { … }
    class Cuenta { … }
}
```



Ensamblados

- C# define un nivel de visibilidad entre los tipos que forman parte del mismo “ensamblado”:

  ➔ visibilidad **`internal`**.

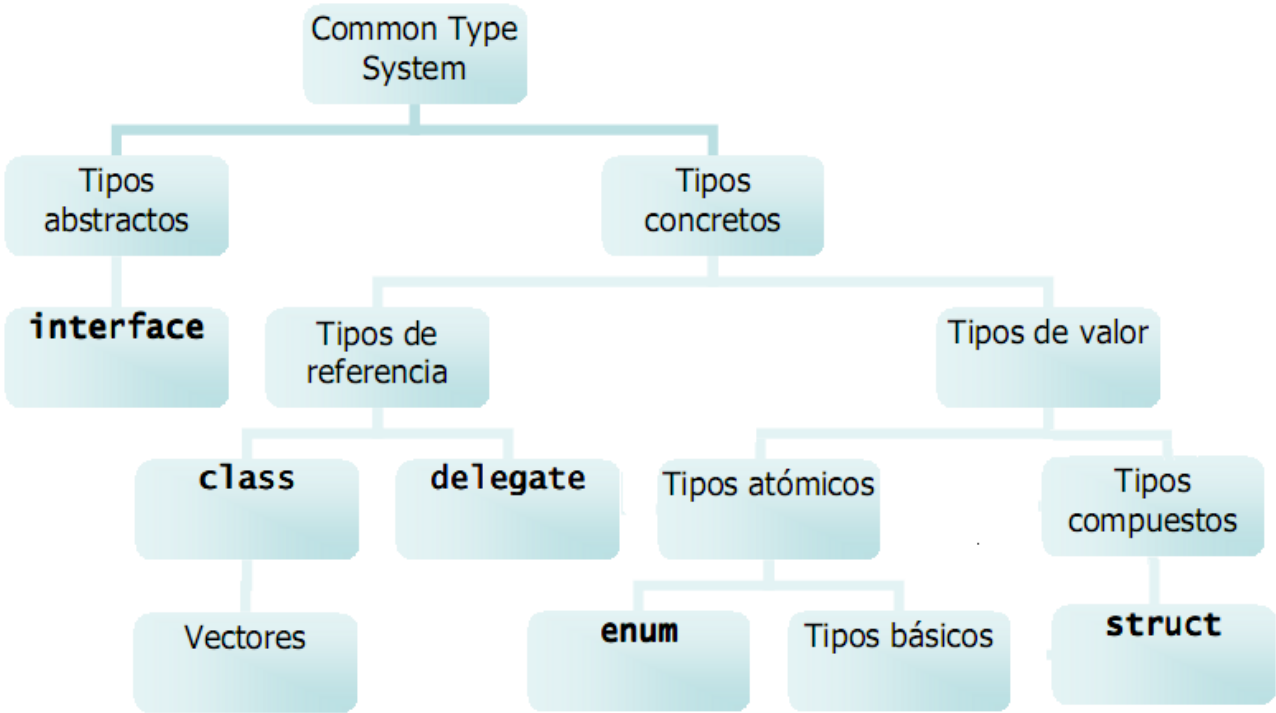
- **Ensamblado**: unidad de empaquetado de software en la plataforma .NET
  - Un fichero ejecutable es un de ensamblado.

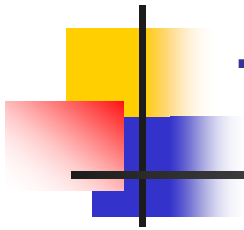
  ➔ Un ensamblado es un **componente software**.

- Visibilidad de los tipos: **`internal`** o **`public`**.
- Por defecto, la visibilidad es **`internal`**.

Tipos del lenguaje

- Corresponden con los tipos de la plataforma .NET: ***Common Type System*** (CTS):



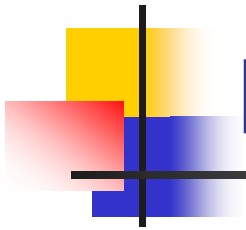


Tipos del lenguaje

- C# es un lenguaje **orientado a objetos puro**.

  ➔ Todos los tipos definen objetos.

- Se distinguen dos **tipos de datos**:
  - Tipos con **semántica referencia**: clases, interfaces, arrays y “delegados”. Aceptan el valor `null`.
  - Tipos con **semántica por valor**: tipos primitivos, enumerados y estructuras.
- Por tanto, los **tipos primitivos** son objetos:
  - Podemos aplicar métodos sobre los tipos primitivos como `ToString` o `Equals`.
  - Tipos: `char`, `int`, `long`, `float`, `double`, `bool`, etc.



Enumerados

- Los enumerados son objetos con **semántica valor**.

- **Declaración** de un enumerado:

```
enum EstadoCuenta { OPERATIVA, INMOVILIZADA, NUMEROS_ROJOS }
```

- Al igual que en C++, las etiquetas del enumerado representan valores enteros.

Enumerados

- **Uso** de un enumerado:

```
public class Cuenta
{    …
     private EstadoCuenta estado;

     public Cuenta(Persona titular, double saldo)
     { …
          estado = EstadoCuenta.OPERATIVA;
     }
```

- Si no se inicializa un enumerado, toma como **valor por defecto** la primera etiqueta.

Arrays

- Los arrays son objetos con **semántica referencia**.
- Se declaran y usan igual que en Java:

```
public class Cuenta
{
    …
    private double[] ultimasOperaciones;

    public Cuenta(Persona titular, double saldo)
    {
        …
        ultimasOperaciones = new double[MAX_OPERACIONES];
    }
```

Construcción de objetos

- **Declaración y construcción** de objetos

```
Persona persona;

persona = new Persona("34565433", "Juan González");

Cuenta cuenta = new Cuenta(persona, 300);
```

- La declaración de una variable (por valor o referencia) no la inicializa.

- Los objetos se crean con el operador **new**.

Constructores

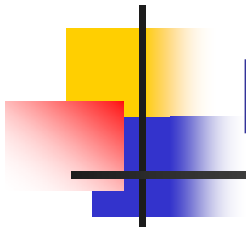
- **Declaración** de constructores (= C++ y Java):
  - Tienen el nombre de la clase y no declaran tipo de retorno.
  - Se permite **sobrecarga**.
  - Si no se define un constructor, el compilador incluye el **constructor por defecto** (vacío y sin argumentos).

```
public Cuenta(Persona titular, double saldo)
{
      this.codigo = ++ultimoCodigo;
      this.titular = titular;
      this.saldo = saldo;
      estado = EstadoCuenta.OPERATIVA;
      ultimasOperaciones = new double[MAX_OPERACIONES];
}
```

Constructores

- Los constructores se pueden **reutilizar** con la palabra clave **`this`** (= Java)
- En relación a Java, cambia la ubicación de la llamada **`this`**: justo después de la declaración de los parámetros.

```
public Cuenta(Persona titular, double saldo)
{ … }

public Cuenta(Persona titular): this(titular, 200)
{

}
```



Destructores

- El CLR de .NET incorpora un mecanismo de recolección de memoria dinámica: **Garbage Collector** (= Java)
- Se puede declarar el método **`Finalize`**`()` para liberar recursos que quedan fuera del entorno de ejecución (= método `finalize()` de Java).
- Por tanto, no existe el operador `delete` para liberar memoria dinámica.

Estructuras

- Construcción para definir **objetos** cuya semántica de almacenamiento es **por valor**.
- En relación a las clases, se diferencian:
  - **No pueden heredar** de otra estructura (ni clase).
  - No se puede definir un **constructor sin parámetros**: el compilador siempre genera uno.
  - **Un constructor debe inicializar todos los atributos** de la estructura. Además, no se puede aplicar ningún método ni usar una propiedad antes de la inicialización.
  - No se puede realizar **inicialización explícita de atributos** de instancia.
  - El método **`Equals`** por defecto realiza una **igualdad superficial**.

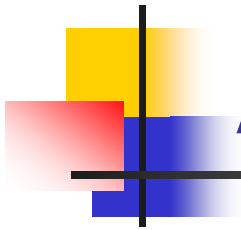
Estructuras

```
public struct Punto {
    private int x;
    private int y;

    public int X { get { return x; } }
    public int Y { get { return y; } }

    public Punto(int x, int y {
        this.x = x;
        this.y = y;
    }
    public void desplaza(int enX, int enY){
        x = x + enX;
        x = x + enY;
    }
}
```

Estructuras

- La semántica valor implica que la declaración de la variable reserva la memoria.
- Sin embargo, se inicializa con el operador **new**.
- La **asignación** realiza una copia superficial (= C++).

```
Punto punto; // No está inicializada
punto = new Punto(2, 3);
Console.WriteLine("Punto X: " + punto.X); // 2

Punto punto2 = new Punto(8, 7);
punto = punto2;
Console.WriteLine("Punto X: " + punto.X); // 8
```



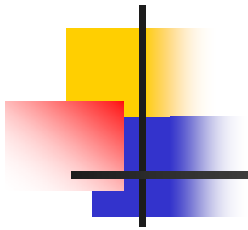
Asignación y copia

- **Operador de asignación** (=)
  - Entre **tipos referencia** (clases, interfaces): se copia el identificador de objeto (= Java).
  - Entre **tipos valor** (estructuras): se realiza una copia superficial.
- C# permite la **redefinición de operadores**. Sin embargo, no se puede redefinir el operador de asignación.
- Para copiar objetos por referencia se recomienda definir el método **`Clone`** (= Java).

Método Clone

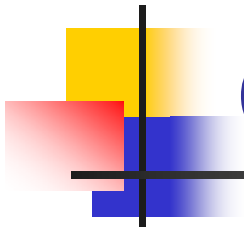
- Hay que implementar la interfaz **`ICloneable`** que define el método **`Clone`**`()`.
- De la clase **`object`** se hereda el método protegido **`MemberwiseClone`**`()` que realiza una copia superficial del objeto receptor.
- En C# no podemos cambiar el tipo de retorno (no se define la regla covariante).

```
public class Cuenta: ICloneable
{  …
    // Realiza una copia superficial
    public object Clone() {
      return this.MemberwiseClone();
    }
}
```



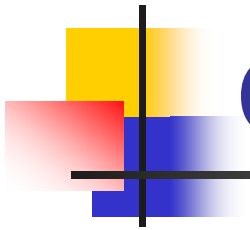
Identidad e Igualdad

- **Operadores de igualdad** (== y !=)
  - **Tipos referencia**: consulta la identidad (= Java).
  - **Tipos valor**: no está disponible (= C++)
- **Redefinición operador** (== y !=)
  - **Tipos valor**: recomendable, ya que no está disponible.
  - **Tipos referencia**: no deberían redefinirse.
- Todos los objetos disponen del método **`Equals`**:
  - **Tipos referencia**: consulta la identidad de los objetos.
  - **Tipos valor**: realiza igualdad superficial de los campos.
- El método **`Equals`** puede redefinirse en clases (= Java) y estructuras.



Operadores

- Al igual que en C++, es posible redefinir gran parte de los operadores (==, !=, <, etc.)
- Sin embargo, en C# **no podemos redefinir el operador de asignación** (=).
- Los operadores se declaran como **métodos de clase**.
- Se utiliza como nombre de método **`operator`** seguido del operador:
  - `operator==`, `operador<`, etc.
- Algunos operadores deben declararse en pareja: == y !=, < y >, etc.



Operadores

```
public static bool operator> (Cuenta cuenta1, Cuenta cuenta2)
{
    return (cuenta1.saldo > cuenta2.saldo);
}

public static bool operator< (Cuenta cuenta1, Cuenta cuenta2)
{
    return (cuenta1.saldo  < cuenta2.saldo);
}
```

```
Cuenta c1 = new Cuenta(persona, 100);
Cuenta c2 = new Cuenta(persona, 200);

Console.WriteLine (c1 > c2); // False
```

Operadores implícitos

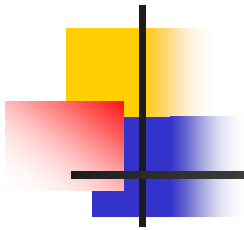
- C# no permite definir el operador =, pero ofrece la alternativa de crear **operadores implícitos**:

```
// A partir de una persona crea una cuenta
public static implicit operator Cuenta (Persona titular)
{
    return new Cuenta(titular, 500);
}
// Si es asignado a un double, se toma el saldo
public static implicit operator double (Cuenta cuenta)
{
    return cuenta.Saldo;
}
```

Operadores implícitos

- Ante una asignación en la que interviene el tipo `Cuenta`, el compilador comprueba si se ha definido un operador implícito.
- En el ejemplo, se realiza asignación `Cuenta = Persona` y `double = Cuenta`.

```
Cuenta cuenta = persona;
Console.WriteLine(cuenta.Saldo); // 500

cuenta.Ingreso(300);
double valor = cuenta;
Console.WriteLine(valor); //800
```



Métodos y mensajes

- Al igual que en Java y C++, los métodos definidos en una clase son los mensajes aplicables sobre los objetos de la clase.
- Está permitida la **sobrecarga** de métodos.
- La **aplicación de métodos** y el acceso a los miembros de un objeto se realiza siempre utilizando la notación punto “.”
  - `cuenta.Ingreso(200); // Referencia`
  - `punto.Desplaza(2,3); // Valor`
- Si no se indica el objeto receptor, la llamada se realiza sobre la instancia actual.

Paso de parámetros

- Paso de parámetros **por valor**, **por referencia** y **de salida**:

```
void metodo(int valor, ref int referencia, out int salida)
{
   valor++; // Se incrementa la copia

   referencia++; // Se incrementa el parámetro real

   salida = 1; // Es obligatorio asignar un valor
               // antes de usarlo
}
```

Paso de parámetros

- **Parámetro por valor** (= Java y C++)
  - Copia el parámetro real sobre el parámetro formal.
- **Paso por referencia**:
  - Se utiliza el modificador **`ref`** para declarar y usar el parámetro.
  - El parámetro formal es una referencia a la variable usada como parámetro real (= C++)
- **Parámetros de salida**:
  - Se utiliza el modificador **`out`** para declarar y usar el parámetro.
  - Parecido a un parámetro por referencia, salvo que es obligatorio asignarle un valor antes de utilizarlo.
  - Resultan útiles para ampliar los valores de retorno de un método.

Paso de parámetros

- Para realizar el paso de parámetros por referencia hay que utilizar la palabra clave **`ref`**.
- Asimismo, para el parámetro de salida **`out`**.

```
int intValor = 3;
int intReferencia = 3;
int intSalida;

cuenta.Metodo(intValor, ref intReferencia, out intSalida);

Console.WriteLine("Por valor = " + intValor); // 3

Console.WriteLine("Por referencia = " + intReferencia); // 4

Console.WriteLine("Salida = " + intSalida); // 1
```

Paso de objetos como parámetro

```
public void Transferencia (Cuenta emisor, Cuenta receptor,
                                              double cantidad) {
    // Cambia el estado de los parámetros reales
    emisor.Reintegro(cantidad);
    receptor.Ingreso(cantidad);

    // No se ve afectado el parámetro real
    receptor = null;
}
```

- **Paso de las referencias por valor** (= Java)
- El estado de los objetos `emisor` y `receptor` cambia.
- La variable utilizada en el paso del parámetro receptor no cambia, ya que se asigna a `null` una copia.

Paso de objetos como parámetro

- **Paso por referencia del parámetro**

```
public void Transferencia (Cuenta emisor, ref Cuenta receptor,
                           double cantidad) {
    // Cambia el estado de los parámetros reales
    emisor.Reintegro(cantidad);
    receptor.Ingreso(cantidad);
        // El parámetro real cambia!
    receptor = null;
}
```

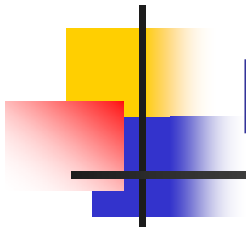
```
Cuenta emisor = new Cuenta(persona, 1000);
Cuenta receptor = new Cuenta(persona, 200);
banco.Transferencia(emisor, ref receptor, 100);
Console.WriteLine("Receptor nulo: "
                  + (receptor == null)); // True
```

Instancia actual

- Al igual que en C++ y Java, la palabra clave **`this`** referencia a la instancia actual.

- Uso de la referencia **`this`**:
  - Evitar el ocultamiento de atributos en los métodos.
  - Dentro de un método, hacer referencia al objeto receptor en un paso de parámetros a otro método.

```
public void Trasladar (Oficina oficina) {
      this.oficina.RemoveCuenta(this);
      oficina.AddCuenta(this);
}
```



Método Main

- C# es menos rígido que Java para la definición del punto de entrada a la aplicación.
- Puede haber **sólo un punto de entrada** (= C++)
- Sólo exige declarar en una clase un **método de clase** con nombre **`Main`**, sin importar la visibilidad.
- Opcionalmente puede tener un parámetro con los **argumentos del programa**.
- **Ejemplos**:
  - `static void Main(string[] args)`
  - `public static void Main()`