Contenido

Introducción.

**Clases y Objetos en C#:**

Módulos: Clases, Estructuras, Espacios de nombres, Ensamblados.

Tipos del lenguaje.

Definición y semántica de los operadores.

Métodos y mensajes. Paso de parámetros.

Construcción de objetos.

**Herencia en C#:**

Polimorfismo y ligadura.

Clase object.

Casting y Compatibilidad de tipos.

Clases abstractas.

Interfaces.

Herencia múltiple.

Iteradores.

**Corrección y Robustez en C#:** asertos y excepciones



Introducción

C

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

# es un lenguaje creado por **Microsoft**

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.



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 y Objetos en C#

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

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.



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 ensamblado.

Un ensamblado es un **componente software**.

Visibilidad de los tipos: **internal** o **public**.

Por defecto, la visibilidad es **internal**.

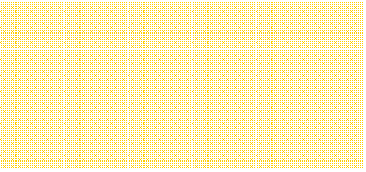
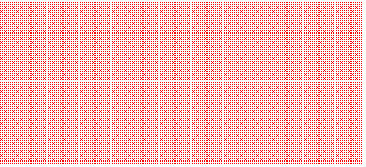
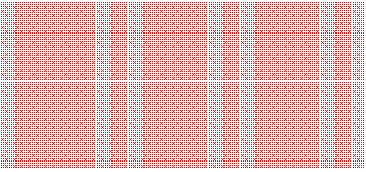
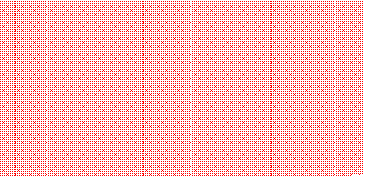
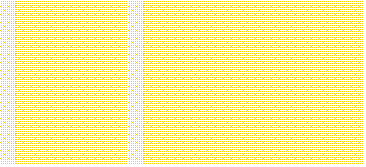
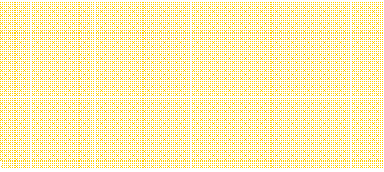
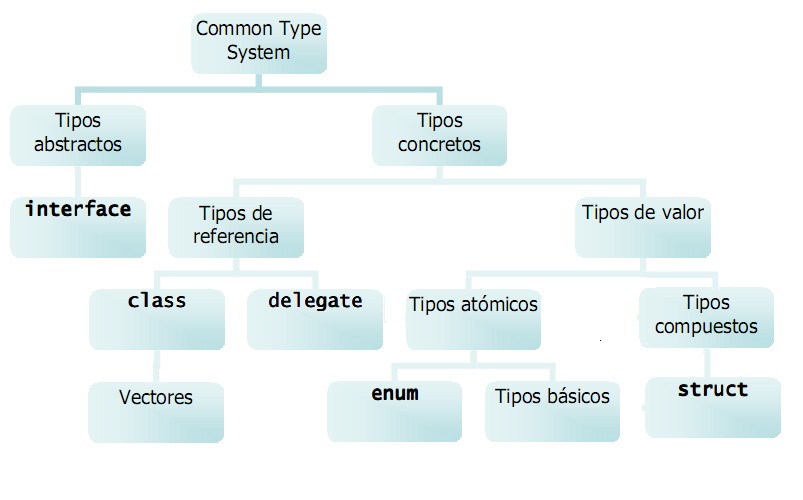


Tipos del lenguaje

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.

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**



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** | | |  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PA |  |  |  |  |
| Paso de objetos como parámetro |
| **Public void Transferencia (Cuenta emisor, Cuenta receptor, doublé 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.



**receptor**:

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**()



Herencia en C#

Herencia.

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | |  |  |
|  |  |  | |

Polimorfismo y ligadura.

Clase object.

Casting.

Compatibilidad de tipos.

Clases abstractas.

Interfaces.

Boxing y unboxing.

Herencia múltiple.

Genericidad.

Estrategias – Delegados.

Iteradores.



Herencia en C#

La herencia en C# comparte características tanto con Java como con C++:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Herencia **simple** ( = Java )

Herencia **pública** ( = Java )

Todos las clases heredan directa o indirectamente de **object** ( = Java )

La aplicación de métodos puede resolverse por **ligadura** estática o dinámica ( = C++)

Por defecto, no se aplica ligadura dinámica ( = C++)



Herencia y constructores

**Los constructores no se heredan** ( = Java y C++ )

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

El constructor de la clase hija tiene que invocar al de la clase padre utilizando la palabra clave **base**.

Si no invoca al constructor del padre, el compilador añade **base()**.

La llamada al constructor se realiza justo después de la lista de parámetros ( = C++)

Redefinición de métodos y ligadura dinámica



|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

La aplicación de un **método o propiedad** sólo se resolverá mediante ligadura dinámica si:

Se declara con el modificador **virtual** en la clase padre ( = C++)

Se utiliza el modificador **override** para el método redefinido en la clase hija.

En un **refinamiento**, se llama a la versión del padre con **base** ( = super de Java).



Redefinición de métodos

Si se define un método con la misma declaración que otro método **virtual** de la clase padre, podemos indicar que no lo redefinimos con el modificador **new**:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Se entiende que se define un método con la misma signatura, pero con **distinto significado**.

**No se aplicaría ligadura dinámica**.

Redefinición y visibilidad



|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Si el método redefinido es **virtual**:

No se puede modificar su nivel de visibilidad (distinto a Java y C++)

Si el método redefinido **no es virtual**:

Podemos cambiar la visibilidad, aumentarla o reducirla.



Restringir la herencia

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Al redefinir un **método virtual**, podemos indicar que no se pueda redefinir en los subtipos con el modificador **sealed** (= final de Java)

**Ejemplo**:

Podríamos definir como **sealed** la redefinición de

Intereses/get en DepositoEstructurado.

Impediría que DepositoGarantizado pudiera cambiar la implementación.

Una **clase** se puede definir como **sealed** indicando que no se puede heredar de ella (= final de Java)



Polimorfismo y ligadura

El **polimorfismo** está permitido sólo para **entidades de tipos referencia** (clases, interfaces).

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

La **ligadura dinámica** sólo se aplica en tipos referencia y en métodos declarados con el modificador **virtual** ( = C++)

Se aplica la versión del tipo dinámico, si la clase del objeto ha redefinido el método con **override**.

La **ligadura estática** se aplica en el resto de casos.



Clase object

La clase **object** representa la raíz de la jerarquía de tipos en C# y .NET

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Define **métodos básicos** para la plataforma:

public virtual bool **Equals**(object otro)

public static bool **ReferenceEquals**

(object obj1, object obj2)

Comprueba siempre la identidad de objetos referencia y es aplicable a referencias nulas.



Clase object

**Métodos básicos**:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

public virtual String **ToString**()

public Type **GetType**()

Equivalente al getClass() de Java.

Para preguntar por el tipo de una variable se utiliza **typeof**(var).

public virtual int **GetHashCode**()

protected object **MemberwiseClone**()

Realiza una copia superficial del objeto receptor de la llamada.



Casting

Se puede aplicar un casting entre tipos compatibles:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**estructurado = (DepositoEstructurado)deposito;**

Sin embargo, para los tipos referencia se define el operador **as**.

**estructurado = deposito as DepositoEstructurado;**

Devuelve null si la conversión no es correcta.

Similar al dynamic\_cast de C++.



Compatibilidad de tipos

Se define el operador **is** para consultar la compatibilidad de tipos (= instanceof d Java):

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**if (deposito is DepositoEstructurado)**

**{**

**// El casting va a ser correcto**

**estructurado = (DepositoEstructurado)deposito;**

**}**



Clases abstractas

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Las clases pueden declararse como abstractas utilizando el modificador **abstract** .

**Métodos y propiedades** se declaran abstractos con

abstract.

Si una **subclase** no implementa una declaración abstracta, debe declararse como abstracta.

Una clase abstracta define un tipo, pero no se pueden construir objetos.

Una clase es abstracta si define un **concepto abstracto** del cual no está permitido crear objetos.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  | Clases abstractas | | |
|  |  |  |  |  |  |  |
| **public** | | | | **abstract class** | **ProductoFinanciero** |
| **{** | |  |  |  |  |
|  |  |  |  | **public Persona** | **Titular { get; private set; }** |
|  |  |  |  | **public ProductoFinanciero(Persona titular) { Titular = titular;**  **}**  **public abstract double Beneficio { get; }** | |
|  |  |  |  | **public double get {** | **Impuestos {** |
|  |  |  |  | **return** | **Beneficio \* 0.18;** |
|  |  |  |  | **}** |  |
|  |  |  |  | **}** |  |
| **}** | |  |  |  |  |



Interfaces

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

C# define el concepto de interfaz similar al de Java.

Permite definir **propiedades y métodos**, pero no constantes.

Una clase puede implementar múltiples interfaces.

Una interfaz puede extender varias interfaces.

Los miembros de una interfaz siempre son públicos.



Interfaces – Declaración

**Declaración** de una interfaz:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**public interface Amortizable**

**{**

**bool Amortizar(double cantidad);**

**}**

Una interfaz puede **extender múltiples**

**interfaces**:

**public interface Flexible : Amortizable, Incrementable**

**{**

**void ActualizarTipoInteres(double tipo);**

**}**



Interfaces – Implementación

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **public class DepositoPenalizable : Deposito**  **{** | **, Amortizable** |  |
|
| **…** |  |  |
| **public bool Amortizar(double cantidad)**  **{**  **if (cantidad > Capital)**  **return false;** | |  |
| **Capital = Capital - cantidad;**  **return true;** |  |  |
| **}** |  |  |
| **}** |  |  |



Interfaces – Métodos repetidos

Dos interfaces puede definir métodos o propiedades con la misma signatura ( métodos repetidos )

|  |  |  |
| --- | --- | --- |
|  | |  |
|  |  |  |

Si una clase implementa las dos interfaces con métodos repetidos, sólo podremos proporcionar una **única implementación** para esos métodos

El mismo problema existe en Java.

En cambio, en C# podemos resolverlo mediante la **implementación explícita de interfaces**.



Interfaces -Implementación explícita

**Implementación explícita de una interfaz**:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

El nombre del método va acompañado del nombre de la interfaz.

No se declara visibilidad. Se asume pública.

**public class DepositoPenalizable : Deposito, Amortizable**

**{ …**

**bool Amortizable.Amortizar(double cantidad)**

**{ … }**

**}**



Interfaces -Implementación explícita

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

La implementación explícita de interfaces tiene las siguientes **limitaciones**:

El método no puede ser utilizado dentro de la clase.

El método no puede ser aplicado sobre variables del tipo de la clase (en el ejemplo, DepositoPenalizable).

El método sólo puede ser aplicable sobre variables polimórficas del tipo de la interfaz:

**DepositoPenalizable penalizable = new …;**

**penalizable.Amortizar(100); // error**

**Amortizable amortizable = penalizable;**

**amortizable.Amortizar(100);**



Interfaces y estructuras

Las estructuras pueden implementar interfaces.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**public interface Reseteable**

**{**

**void reset();**

**}**

**public struct Punto: Reseteable**

**{ …**

**// Método Interfaz Reseteable public void reset()**

**{**

**x = 0;**

**y = 0;**

**}**

**}**



Interfaces y estructuras

Asignación a una interfaz:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Punto punto = new Punto(2, 3);;** |  | |
| **Reseteable res = punto;**  **res.reset();** |
| **Punto otro = (Punto) res;** |
| **Console.WriteLine("Punto X: " +** | **otro.X);** | **// 0** |

Una interfaz es un tipo referencia, ¿cómo puede apuntar a un tipo con semántica valor?

**Boxing**



Herencia múltiple

C# es un lenguaje con herencia simple.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Al igual que en Java, podemos **simular herencia múltiple** utilizando interfaces y relaciones de clientela.

Ideposito Cuenta

**«implements» «extends»**

Barco

Deposito



**depositoImpl**

CuentaRemunerada



Iteradores

El modelo de iteradores de C# es similar al de Java.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Cualquier clase que quiera ser iterable debe implementar la **IEnumerable**:

**public interface IEnumerable<T>**

**{**

**IEnumerator<T> GetEnumerator();**

**}**



Iteradores

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | |  |  |
|  |  |  | |

Interfaz **IEnumerator**:

Diferente a un iterador de Java. No hay método remove()

Método **MoveNext**: avanza hasta el siguiente elemento, indicando si ha conseguido avanzar.

Propiedad **Current**: elemento actual.

Método **Reset**: sitúa el iterador en estado inicial, justo antes del primer elemento.

**public interface IEnumerator<T>**

**{**

**T Current { get; } bool MoveNext(); void Reset();**

**}**



Iteradores

Ejemplo de uso de un iterador:

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | |  |  |
|  |  |  | |

**public Cuenta Buscar(Test test)**

**{**

**IEnumerator<Cuenta> enumerador = cuentas.GetEnumerator();**

**while (enumerador.MoveNext())**

**{**

**Cuenta cuenta = enumerador.Current;**

**if (test(cuenta))**

**return cuenta;**

**}**

**return null;**

**}**



Iteradores

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Al igual que en Java, se puede omitir el uso de un iterador con un recorrido **foreach**:

**public Cuenta Buscar(Test test)**

**{**

**foreach (Cuenta cuenta in cuentas)**

**{**

**if (test(cuenta))**

**return cuenta;**

**}**

**return null;**

**}**



Bloques de iteración

En C# es posible definir métodos con **bloques de iteración**:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Un método que retorna un objeto iterable.

La ejecución del método se detiene cada vez que se llama a **yield return** para retornar un elemento del recorrido.

Cuando se solicita el siguiente elemento, continúa la ejecución del método hasta alcanzar el siguiente yield.

La iteración finaliza al terminar el método o ejecutar

**yield break**.



Bloques de iteración

**Ejemplo**:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Método de búsqueda que permite recorrer los elementos que cumplen una condición de búsqueda.

**public IEnumerable<Cuenta> Buscar2(Test test)**

**{**

**foreach (Cuenta cuenta in cuentas)**

**{**

**if (test(cuenta))**

**yield return cuenta;**

**}**

**yield break;**

**}**



Bloques de iteración

Uso del método de iteración:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**foreach (Cuenta cuenta**

**in banco.Buscar2(elemento => elemento.Saldo > 400))**

**{**

**Console.WriteLine("Cuenta " + cuenta);**

**}**



Implementación iteradores

Los bloques de iteración son utilizados como implementación de las clases iterables.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**Ejemplo**:

La clase **Banco** es iterable. En un recorrido retorna las cuentas que contiene.

La implementación de la interfaz se apoya en un método que usa un bloque de iteración.

C# obliga a implementar dos versiones del método

getEnumerator().

**foreach (Cuenta cuenta in banco) {**

**Console.WriteLine("Cuenta " + cuenta);**

**}**

**class Banco: IEnumerable<Cuenta> {**

**private List<Cuenta> cuentas = new List<Cuenta>();**

**private IEnumerator<Cuenta> getEnumerador() {**

**foreach (Cuenta cuenta in cuentas) {**

**yield return cuenta;**

**}**

**}**

**IEnumerator<Cuenta> IEnumerable<Cuenta>.GetEnumerator() {**

**return getEnumerador();**

**}**

**IEnumerator System.Collections.IEnumerable.GetEnumerator() {**

**return getEnumerador();**

**}**

Corrección y Robustez en C#



|  |  |  |
| --- | --- | --- |
|  | |  |
|  |  |  |

Asertos

Mecanismo de excepciones:

Declaración de excepciones

Lanzar excepciones

Manejo de excepciones

Definición de excepciones



Corrección y Robustez

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**Corrección**:

Es la capacidad de los productos software de realizar con exactitud su tarea (**cumplir su especificación**).

**Robustez**:

Es la capacidad de los productos software de reaccionar adecuadamente ante **situaciones excepcionales**.

Al igual que Java, el lenguaje ofrece **asertos** y **excepciones** como soporte de la corrección y robustez del código.

La **verificación** del código se realiza con pruebas unitarias



Asertos

La clase **System.Diagnostics.Debug** declara el **método** de clase **Assert** para evaluar asertos.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

La evaluación de asertos sólo se realiza en la ejecución de la aplicación de **depuración**.

Los asertos de C# tienen las mismas limitaciones que en C++ y Java. Simplemente son una utilidad de depuración.

**Debug.Assert(valores != null,**

**“Lista de valores no debe ser nula");**



Pruebas Unitarias

Al igual que en Java y C++, las pruebas unitarias no forman parte de la biblioteca del lenguaje.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Existen **herramientas externas** para realizar pruebas unitarias.

El entorno de desarrollo de **Microsoft Visual Studio** incluye en el espacio de nombres Microsoft.VisualStudio.TestTools.UnitTesting soporte para realizar pruebas unitarias.



Excepciones

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

El modelo de excepciones de Java comparte aspectos en común con Java y C++:

**Las excepciones son objetos** ( = Java ).

La raíz de todas las excepciones es la clase

**System.Exception** ( = Throwable de Java).

Todas las excepciones son **no comprobadas** ( = C++).

Sin embargo, se diferencia de Java y C++:

**En la declaración de un método no se puede indicar las excepciones que lanza**. Sólo podemos indicarlo en la documentación.



Excepciones

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

La clase **System.Exception** tiene las características comunes a todas las excepciones:

string **Message** { get; }: mensaje de error.

string **StackTrace** { get; }: pila de llamadas en la que se ha producido la excepción.

Las excepciones se lanzan con **throw** ( = Java y C++).



Excepciones

Una excepción se puede construir de tres formas ( = Java):

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Llamando al **constructor sin parámetros**.

Llamando al **constructor con la cadena de error**.

Llamando al constructor con la **cadena de error** y la **excepción** que ha causado el error.



Excepciones – Clasificación

Se definen dos grupos de excepciones con el propósito de distinguir excepciones predefinidas y de usuario:

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**System.SystemException**: predefinidas en .NET

**System.ApplicationException**: excepciones de usuario.

A diferencia de Java, estos dos grupos sólo representan una clasificación de excepciones.

No tiene relación con el concepto comprobada/no comprobada de Java.



Control de precondiciones

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Las excepciones predefinidas incluyen excepciones para el tratamiento de precondiciones:

**ArgumentException**: precondiciones de argumentos.

**InvalidOperationException**: precondiciones de estado.

**public void Ingreso(double cantidad){**

**if (cantidad < 0)**

**throw new ArgumentException("Cantidad negativa");**

**if (estado != EstadoCuenta.OPERATIVA)**

**throw new InvalidOperationException("Estado incorrecto");**

**saldo = saldo + cantidad;**

**}**



Excepciones de usuario

Las excepciones de usuario utilizadas para notificar el fallo en las postcondiciones heredan de System.**ApplicationException**.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

**namespace Navegador {**

**public class RedNoDisponible : ApplicationException**

**{**

**public RedNoDisponible() { }**

**public RedNoDisponible(string msg): base(msg) { }**

**public RedNoDisponible(string msg, Exception causante)**

**: base(msg, causante) { }**

**}**

**}**



Declaración de excepciones

**Las excepciones que lanza un método no son declaradas en su signatura**.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Se aconseja documentarlas en la declaración del método

**/// <summary>**

**/// Obtiene una nueva línea del fichero**

**/// </summary>**

**/// <returns>Una línea del fichero**

**/// o null si no hay disponibles</returns>**

**/// <exception cref="Navegador.RedNoDisponible">**

**/// Error producido por un fallo en la red**

**/// </exception>**

**public String leerLinea() { … }**



Excepciones y herencia

En C# no se controlan las excepciones que lanza un método.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Por tanto, no hay restricción en el lanzamiento de excepciones en la redefinición de un método en un subtipo.

Es responsabilidad del programador el correcto lanzamiento de excepciones.



Tratamiento de excepciones

Al igual que Java y C++, las excepciones pueden ser tratadas en bloques **try-catch**.

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |
|  |  |  |

Cuando ocurre una excepción se evalúan los tipos definidos en los manejadores y se ejecuta el primero cuyo tipo sea compatible ( = Java y C++)

Se puede definir un **manejador para cualquier tipo de excepción**: catch(Exception e)

Es posible **relanzar una misma excepción** que se está manejando ( = C++): throw;

Las excepciones no tratadas en un método se propagan al método que hace la llamada ( = Java y C++).

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  |  |  |  |
|  | | **public class Navegador**  **{**  **public void Visualiza(String url)**  **{**  **Conexion conexion;**  **int intentos = 0;**  **while (intentos < 20) {**  **try {**  **conexion = new Conexion(url);**  **break;**  **}**  **catch (RedNoDisponible e) { System.Threading.Thread.Sleep(1000); intentos++;**  **if (intentos == 20) throw; // relanza**  **}**  **}**  **…** | |