Programación Orientada a Objetos C#

Félix Gómez Mármol

3º Ingeniería Informática
Julio de 2004

Índice general

Índice General **4**

2. Clases y Objetos **5**
2.1. Clases . . . 5
2.1.1. Estructura . . . 5
2.1.2. Ocultación de la información . . . 7
2.1.3. Relaciones entre clases: Cliente-Servidor y Herencia . . . 8
2.1.4. Visibilidad . . . 9
2.2. Objetos . . . 10
2.3. Mensajes . . . 10
2.3.1. Sintaxis. Notación punto . . . 10
2.3.2. Semántica . . . 10
2.4. Semántica referencia versus semántica almacenamiento . . . 11
2.5. Creación de objetos . . . 11
2.5.1. Destructores . . . 12
2.6. Semántica de la asignación e igualdad entre objetos . . . 12
2.7. Genericidad . . . 14
2.8. Definición de una clase "Lista Doblemente Enlazada" . . . 14

3. Diseño por Contrato: Asertos y Excepciones **17**
3.1. Contrato software . . . 17
3.2. Clases y Corrección del software: Asertos . . . 17
3.3. Excepciones en C# . . . 18
3.3.1. Lanzamiento de excepciones . . . 20
3.3.2. Bloques `try` . . . 21
3.3.3. Manejadores . . . 22
3.4. Conversión de asertos en excepciones . . . 22

4. Herencia **25**
4.1. Introducción . . . 25
4.2. Doble aspecto de la herencia . . . 26
4.3. Polimorfismo . . . 26
4.3.1. Sobrecarga . . . 27
4.3.2. Regla de aplicación de propiedades . . . 29
4.3.3. Estructuras de datos polimórficas . . . 29
4.3.4. Operadores `is` y `as` . . . 29
4.4. Ligadura Dinámica . . . 30
4.5. Clases Diferidas . . . 31
4.5.1. Interfaces . . . 32
4.6. Herencia, reutilización y extensibilidad del software . . . 34
4.7. Herencia múltiple . . . 37

4.7.1. Problemas: Colisión de nombres y herencia repetida 37
4.7.2. Ejemplos de utilidad 38
4.8. C#, Java y C++: Estudio comparativo 39

Índice de Figuras **41**

Índice de Tablas **43**

Índice de Códigos **45**

Bibliografía **47**

Tema 2

Clases y Objetos

2.1. Clases

Definición 2.1 *Una clase es una implementación total o parcial de un tipo abstracto de dato (TAD).*

Sus características más destacables son que se trata de entidades sintácticas y que describen objetos que van a tener la misma estructura y el mismo comportamiento.

2.1.1. Estructura

Los componentes principales de una clase, que a partir de ahora llamaremos *miembros*, son:

- **Atributos**, que determinan una estructura de almacenamiento para cada objeto de la clase, y
- **Métodos**, que no son más que operaciones aplicables sobre los objetos.

Ejemplo 2.1 *La clase mostrada en el código 2.1, llamada Luna, convierte a kilómetros la distancia de la Tierra a la Luna en millas.*

```
1  //Distancia hasta la Luna convertida a kilómetros
2  using System;
3
4  public class Luna {
5     public static void Main()
6     {
7        int luna = 238857;
8        int lunaKilo;
9
10       Console.WriteLine("De la Tierra a la Luna = " + luna
11                         + " millas");
12       lunaKilo = (int)(luna * 1.609);
13       Console.WriteLine("Kilómetros = "
14                         + lunaKilo + "km.");
15    }
16 }
```

Código 2.1: Clase Luna

Tipo	Bytes	Rango de Valores
`short`	2	(-32768, 32767)
`ushort`	2	(0, 65535)
`int`	4	(-2147483648, 2147483647)
`uint`	4	(0, 4294967295)
`long`	8	(-9223372036854775808, 9223372036854775807)
`ulong`	8	(0, 18446744073709551615)

Tabla 2.1: Tipos enteros primitivos

Tipo	Bytes	Rango de Valores
`float`	4	$(\pm 3{,}4 \times 10^{38})$ 7 dígitos significativos
`double`	8	$(\pm 1{,}7 \times 10^{38})$ de 15 a 16 dígitos significativos
`decimal`	16	$(10^{-28}, 7{,}9 \times 10^{+28})$ de 28 a 29 dígitos significativos

Tabla 2.2: Tipos flotantes primitivos

Tipo	Bytes	Rango de Valores
`byte`	1	(-128,127)
`ubyte`	1	(0,255)
`bool`	1	{true, false}
`char`	2	Tabla ASCII

Tabla 2.3: Otros tipos primitivos

Tipos de datos primitivos

Las tablas 2.1, 2.2 y 2.3 muestran los tipos primitivos soportados por C#.

Para declarar un tipo consistente en un conjunto etiquetado de constantes enteras se emplea la palabra clave `enum` (por ejemplo, `enum Edades {Félix = 21, Alex, Alberto = 15}`).

Palabras reservadas

La tabla 2.4 muestra las palabras reservadas de C#.

abstract	do	implicit	private	this
as	double	in	protected	throw
base	else	int	public	true
bool	enum	interface	readonly	try
break	event	internal	ref	typeof
byte	explicit	is	return	unit
case	extern	lock	sbyte	ulong
catch	false	long	sealed	unchecked
char	finally	namespace	set	unsafe
checked	fixed	new	short	ushort
class	float	null	sizeof	using
const	for	object	stackalloc	value
continue	foreach	operator	static	virtual
decimal	get	out	string	void
default	goto	override	struct	volatile
delegate	if	params	switch	while

Tabla 2.4: Palabras reservadas

2.1.2. Ocultación de la información

En ocasiones conviene ocultar ciertas características (atributos y/o métodos) de una clase al exterior. Por ejemplo, en una relación entre clases de Cliente-Servidor, el servidor debería ocultar los aspectos de implementación al cliente.

Para llevar a cabo esto, C# proporciona tres tipos de acceso: público, privado y protegido.

Un miembro con acceso privado (opción por defecto) sólo es accesible desde otros miembros de esa misma clase, mientras que uno con acceso público es accesible desde cualquier clase. Por último, un miembro protegido sólo es accesible por miembros de la misma clase o bien por miembros de alguna de las subclases.

Ejemplo 2.2 *En el código 2.2 se muestra un ejemplo de ocultación de información, mediante la clase Punto.*

```
1  public class Punto {
2     private double x, y;       //O simplemente double x, y;
3     public void SetPunto(double u, double v)
4     { x = u; y = v; }
5  }
6  . . .
7  class Test {
8     public void Prueba(Punto w)
9     {
10       . . .
11       w.SetPunto(4.3,6.9);  //Correcto
12       w.x = 5.5;            //Error sintáctico
13       . . .
14    }
15 }
```

Código 2.2: Ejemplo de Ocultación de Información

A diferencia de C++, en C# es posible declarar un miembro como de sólo lectura mediante la palabra clave `readonly`.

La diferencia entre `readonly` y `const` es que con la primera opción la inicialización tiene lugar en tiempo de ejecución, mientras que con la segunda opción la inicialización se da en tiempo de compilación.

Una forma de simular el efecto de `const` mediante `readonly` es usando además la palabra clave `static`.

Ejemplo 2.3 *En el siguiente código, n y m podríamos decir que son "estructuralmente equivalentes":*

```
public const n;
public static readonly m;
```

Otra práctica interesante consiste en declarar un atributo como privado y, mediante otro atributo público tener acceso de lectura y/o escritura sobre el atributo privado. Para ello nos valemos de los modificadores de acceso `get` y `set`.

Ejemplo 2.4 *En el código 2.3 se muestra un ejemplo de ocultación de información mediante el uso de* `get` *y* `set`*.*

Obsérvese que el identificador `value` es siempre un objeto del mismo tipo que el atributo que lo contiene.

```
1  public class CuentaPalabras {
2     private string m_file_output; //Atributo privado
3
4     public string OutFile //Atributo público asociado
5     {
6        get { return m_file_output; } //Acceso de lectura
7        set
8        { //Acceso de escritura
9           if ( value.Length != 0 )
10             m_file_output = value;
11       }
12    }
13 }
```

Código 2.3: Otro ejemplo de Ocultación de Información

2.1.3. Relaciones entre clases: Cliente-Servidor y Herencia

Un cliente de una clase debe ver a ésta como un TAD, es decir, conocer su especificación, pero sin importarle su implementación (siempre que ésta sea eficiente y potente).

El creador de la clase, en cambio, debe preocuparse de que dicha clase sea lo más reutilizable, adaptable, y eficiente, así como que pueda ser usada por el mayor número de clientes posible.

Definición 2.2 *Una clase A se dice que es cliente de una clase B, si A contiene una declaración en la que se establezca que cierta entidad (atributo, parámetro, variable local) e es de tipo B.*

Definición 2.3 *Una clase A se dice que hereda de otra clase B, si A es una versión especializada de B (herencia de especialización), o bien si A es una implementación de B (herencia de implementación).*

La figura 2.1.a muestra la relación de clientela entre A y B, mientras que en la 2.1.b se observa la relación de herencia entre dichas clases.

(a) (b)

Figura 2.1: Clientela y Herencia entre A y B

Ejemplo 2.5 *El siguiente código muestra la relación de clientela vista en la figura 2.1.a:*

```
class A {
. . .
B conta;
. . .
}
```

Por su parte, el siguiente código muestra la relación de herencia mostrada en la figura 2.1.b:

```
class A: B { . . . }
```

Por último cabe decir que la herencia es una decisión de diseño más comprometedora que la clientela, pues en ésta última se puede cambiar la implementación de la clase que se emplea en el cliente, sin afectar a éste (en nuestro ejemplo, se puede modificar la implementación de B sin que esto afecte a A).

2.1.4. Visibilidad

Además de lo comentado en el apartado 2.1.2 acerca de la ocultación de información, en esta sección trataremos la cuestión de las clases anidadas.

Definición 2.4 *Una clase anidada no es más que una clase que se declara dentro de otra y que tiene visibilidad sobre todas las propiedades de los objetos de la clase que la incluye.*

Ejemplo 2.6 *El código 2.4 muestra un ejemplo de clases anidadas.*
El resultado mostrado por pantalla sería:

```
IntAnidado(6) = 17
```

```
1  public class Uno {
2     public int c;
3  }
4
5  public class Dos {
6     public int c;
7     public class Tres {
8        public int IntAnidado(int e)
9        {
10          Dos d = new Dos();
11          Uno u = new Uno();
12          u.c = 5 + e;
13          d.c = c = e;
14          return u.c + c;
15       }
16       private int c;
17    }
18    public Tres y;
19 }
20
21 public class PruebaAnidada {
22    public static void Main()
23    {
24       Dos x = new Dos();
25       x.y = new Tres();
26       Console.WriteLine("IntAnidado(6) = " + IntAnidado(6));
27    }
28 }
```

Código 2.4: Ejemplo de Clases Anidadas

2.2. Objetos

Definición 2.5 *Un objeto es una instancia de una clase, creada en tiempo de ejecución y formada por tantos campos como atributos tenga la clase.*

Dichos campos son simples si corresponden a atributos de tipos primitivos (véase tablas 2.1, 2.2 y 2.3), mientras que se dice que son compuestos si sus valores son subobjetos o referencias.

Mientras exista, cada objeto se identifica unívocamente mediante su *identificador de objeto (oid)*.

Una posible clasificación de los objetos podría ser:

- Objetos **externos**: Son aquellos que existen en el dominio de la aplicación. Por ejemplo, *Producto, Socio, Comercial, Descuento, ...*
- Objetos **software**:
  - Procedentes del análisis: objetos del dominio.
  - Procedentes del diseño/implementación: TDA's, patrones de diseño y GUI.

El estado de un objeto viene dado por la lista de pares atributo/valor de cada campo. Su modificación y consulta se realiza mediante mensajes.

En cada instante de la ejecución de una aplicación Orientada a Objetos (en adelante, OO) existe un objeto destacado sobre el que se realiza algún tipo de operación. Este objeto recibe el nombre de **instancia actual**.

Para hacer referencia a la instancia actual en C# se emplea la palabra reservada `this`.

2.3. Mensajes

Los mensajes son el mecanismo básico de la computación OO y consisten en la invocación de la aplicación de un método sobre un objeto.

Constan de tres partes: objeto receptor, identificador del método y los argumentos de éste último.

2.3.1. Sintaxis. Notación punto

Como se ha podido observar en algunos de los ejemplos vistos hasta ahora, la forma en que se invoca a un método es mediante el operador punto, siguiendo una sintaxis como la que a continuación se muestra:

```
receptor.método(argumentos)
```

2.3.2. Semántica

La diferencia entre un mensaje y la invocación a un procedimiento es que en éste último todos los argumentos reciben el mismo trato, mientras que en los mensajes uno de esos argumentos, a saber, el objeto receptor, recibe un "trato especial".

Ejemplo 2.7 *En el mensaje* `w.SetPunto(4.3,6.9)` *lo que se está queriendo decir es que se aplique el método* `SetPunto` *sobre el objeto receptor* `w`*, efectuando el correspondiente paso de parámetros.*

Cuando un mensaje no especifica al objeto receptor la operación se aplica sobre la instancia actual.

Si no se incluye nada, el paso de parámetros es "por valor". Para especificar un paso de parámetros "por referencia" se emplea la palabra clave `ref` delante del parámetro en cuestión.

2.4. Semántica referencia versus semántica almacenamiento

En C# se tiene semántica referencia (figura 2.4[1]) para cualquier entidad asociada a una clase, aunque también existen los tipos primitivos (figura 2.2[2]), como ya hemos visto.

Toda referencia puede estar, o bien no ligada (con valor `null`), o bien ligada a un objeto (mediante el operador `new`, o si se le asigna una referencia ya ligada). Pero ojo, la asignación no implica copia de valores sino de referencias (⇒ Aliasing).

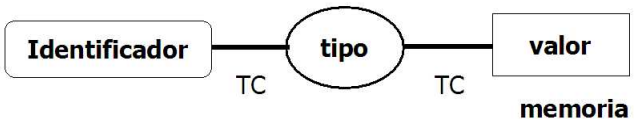

Figura 2.2: Variables en los lenguajes tradicionales

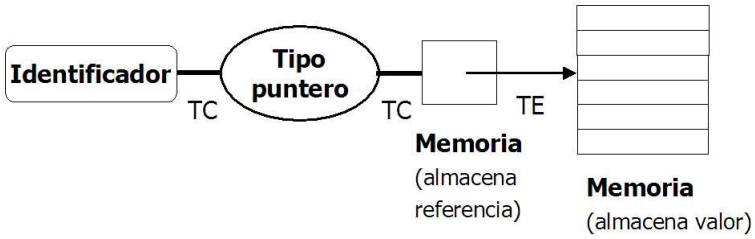

Figura 2.3: Variables tipo "puntero" en los lenguajes tradicionales

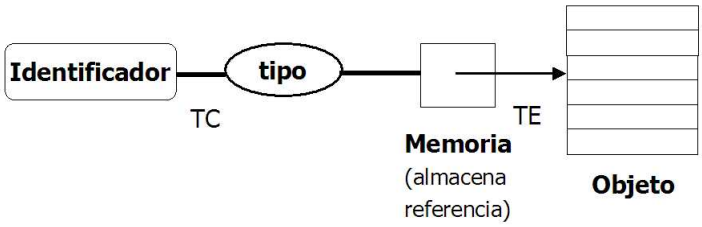

Figura 2.4: Variables en los lenguajes OO

Algunas ventajas de los tipos referencia son:

- Son más eficientes para manejar objetos.
- Constituyen un soporte para definir estructuras de datos recursivas.
- Dan soporte al polimorfismo.
- Los objetos son creados cuando son necesarios.
- Se permite la compartición de un objeto.

2.5. Creación de objetos

En C# existe un mecanismo explícito de creación de objetos mediante la instrucción de creación `new` y los llamados métodos *constructores* (que deben tener el mismo nombre que la clase en la que se definen).

Estos constructores se encargan de inicializar los atributos con valores consistentes. Sin embargo también se pueden inicializar con unos valores por defecto, mediante los *constructores por defecto* (aquellos que no tienen argumentos).

[1] TE→Tiempo de Ejecución
[2] TC→Tiempo de Compilación

Ejemplo 2.8 *El código 2.5 muestra un ejemplo de una clase con un constructor y un constructor por defecto.*

```
1  public class Contador {
2     private int conta;
3
4     public Contador() { conta = 0; } //Constructor por defecto
5     public Contador(int i) { conta = i % 100; } //Constructor
6     public int Conta
7     {
8        get { return conta; }
9        set { conta = value % 100; }
10    }
11    public void Click() { conta = (conta + 1) % 100; }
12 }
```

Código 2.5: Ejemplo de Constructores

2.5.1. Destructores

Un destructor es un método cuyo nombre es el nombre de la clase precedido de una tilde ∼. Mientras que los constructores sí pueden tener modificadores de acceso, los destructores carecen de ellos.

Cuando un objeto es recogido por el recolector de basura[3] se llama implícitamente a su destructor. No obstante, en la mayoría de los casos las clases no necesitan destructores, pues con el recolector de basura es suficiente para la finalización.

2.6. Semántica de la asignación e igualdad entre objetos

Definición 2.6 *Una copia se dice que es superficial si se copian los valores de cada campo del objeto origen en el objeto destino y ambos comparten referencias (figura 2.5).*

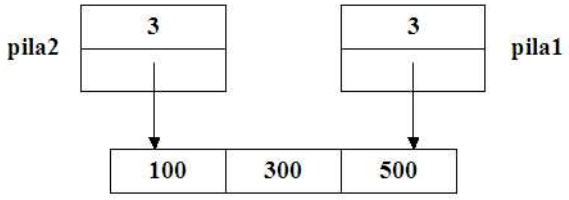

Figura 2.5: Ejemplo de copia superficial

Definición 2.7 *Una copia se dice que es profunda si se crea un objeto con una estructura idéntica al objeto origen y sin compartición de referencias (figura 2.6).*

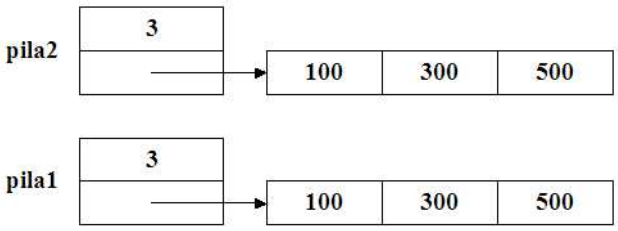

Figura 2.6: Ejemplo de copia profunda

[3] *garbage collector*

Como ya se dijo anteriormente, la asignación implica compartición de referencias si se tiene semántica referencia, mientras que si se tiene semántica almacenamiento entonces se copian los valores. A este tipo de copia se le llama *copia superficial* y está soportada en C#.

Si se tiene semántica referencia y se realiza una copia superficial, los cambios hechos en una variable tienen repercusión en su(s) correspondiente(s) copia(s).

Para evitar este efecto debemos realizar una copia profunda, lo cual puede llevarse a cabo, por ejemplo, implementando el método `Clone()` de la interfaz `ICloneable`.

Ejemplo 2.9 *El código 2.6 muestra un ejemplo de una clase que implementa el método* `Clone()` *de la interfaz* `ICloneable` *para conseguir una copia profunda.*

```
1  public class matrix : ICloneable
2  {
3  . . .
4     public matrix( int row, int col )
5     {
6        m_row = ( row <= 0 ) ? 1 : row;
7        m_col = ( col <= 0 ) ? 1 : col;
8
9        m_mat = new double[ m_row, m_col ];
10    }
11
12    public object Clone()
13    {
14       matrix mat = new matrix( m_row, m_col );
15       for ( int i = 0; i < m_row; ++i )
16          for ( int j = 0; j < m_col; ++j )
17             mat.m_mat[i,j] = m_mat[i,j];
18       return mat;
19    }
20 }
```

Código 2.6: Ejemplo de implementación del método `Clone()`

Nota.- La línea 6 (equivalentemente la línea 7) del código 2.6 es equivalente al siguiente código:

```
if (row <= 0) m_row = 1;
else m_row = row;
```

Definición 2.8 *Dos variables se dice que son idénticas si ambas referencian al mismo objeto (figura 2.7).*

En C# esto se expresa de la siguiente forma: `oa == ob`

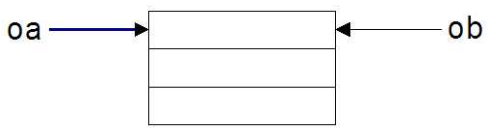

Figura 2.7: Ejemplo de variables idénticas

Definición 2.9 *Dos variables se dice que son iguales si ambas referencian a objetos con valores iguales en sus campos (figuras 2.5 y 2.7).*

En C# esto se expresa de la siguiente forma: `oa.Equals(ob)`

De esta última definición se deduce que siempre que haya identidad, también habrá igualdad superficial (figura 2.7), pero no necesariamente a la inversa (figura 2.5).

2.7. Genericidad

C++ hace uso de plantillas, que son altamente eficientes y muy potentes, para implementar clases contenedoras genéricas. Dichas plantillas están propuestas para Java y C#, pero aún están en una fase experimental.

C# consigue "genericidad" al estilo de Java, a saber, mediante el empleo de la clase raíz `Object`, con los consabidos inconvenientes:

- Necesidad de conversiones explícitas de tipo.
- Imposibilidad de asegurar homogeneidad.

2.8. Definición de una clase "Lista Doblemente Enlazada"

```
1  using System;
2  public class DListElement
3  {
4     public DListElement(object val) { data = val; }
5
6     public DListElement(object val, DListElement d)
7     {
8        data = val;
9        next = d;
10       d.previous = this;
11    }
12
13    public DListElement Next
14    { get { return next; } set { next = value; } }
15
16    public DListElement Previous
17    { get { return previous; } set { previous = value; } }
18
19    public DListElement Data
20    { get { return data; } set { data = value; } }
21
22    private DListElement next, previous;
23    private object data;
24 }
25
26 public class DList
27 {
28    public DList() { head = null; }
29    public DList(object val)
30    {
31       head = new DListElement(val);
32       head.Next = head.Previous = null;
33    }
34
```

```
35    public bool IsEmpty() { return head == null; }
36
37    public void Add(object val)
38    {
39       if (IsEmpty())
40       {
41          head = new DListElement(val);
42          head.Next = head.Previous = null;
43       }
44       else
45       {
46          DListElement h = new DListElement(val,head);
47          head.Previous = h;
48          head = h;
49       }
50    }
51
52    public object Front()
53    {
54       if (IsEmpty())
55          throw new System.Exception("Empty DList");
56       return head.Data;
57    }
58
59    public DListElement Find (object val)
60    {
61       DListElement h = head;
62
63       while (h != null)
64       {
65          if (val.Equals(h.Data))
66             break;
67          h = h.Next;
68       }
69       return h;
70    }
71
72    private DListElement Delete(DListElement h)
73    {
74       if ( h == null ) return null;
75
76       DListElement np = h.Next, pp = h.Previous;
77
78       if ( np != null )
79          np.Previous = pp;
80       else
81          head = null;
82       if ( pp != null )
83          pp.Next = np;
84       return h.Next;
85    }
```

```
86
87      public DListElement Delete(object v)
88      { return Delete(Find(v)); }
89
90      public override string ToString()
91      { return ToStringRecursive(head); }
92
93      static string ToStringRecursive(DListElement first)
94      {
95          if ( first == null )
96              return "";
97          else
98              return (first.Data + "\n")
99                + ToStringRecursive(first.Next);
100     }
101     private DListElement head;
102 }
```

Código 2.7: Lista genérica doblemente enlazada

A continuación destacaremos algunos aspectos de este código:

- Obsérvese el uso de get y set en las líneas 13 a 20 (en métodos declarados como públicos) y las declaraciones privadas de las líneas 22 y 23. Es este un ejemplo de ocultación de información.
- Otro ejemplo de ocultación de información es la declaración como privados de los métodos `Delete` y `ToStringRecursive` de las líneas 72 y 93, respectivamente.
- Podemos ver la implementación de un constructor por defecto y un constructor en las líneas 28 y 29, respectivamente, así como la sobrecarga de constructores en las líneas 4 y 6.
- Como se observa en la línea 23 de este código los datos almacenados en la lista son de tipo `object`. Como ya se dijo anteriormente, así es como se consigue "genericidad" en C#.
- En la línea 65 vemos un ejemplo de uso del método `Equals`. El empleo del operador == en esta línea provocaría una ejecución incorrecta, distinta a la esperada.
- En la línea 55 se ve un ejemplo del uso de excepciones, las cuales trataremos en el siguiente tema; y en la línea 90 vemos cómo se sobrescribe el método heredado `ToString()`[4].

[4]Todo lo relacionado con la herencia se verá en profundidad en el tema 4

Tema 3

Diseño por Contrato: Asertos y Excepciones

En este tema se describe el manejo de excepciones en C#. Las excepciones son habitualmente condiciones inesperadas de error que provocan la finalización del programa en el que tienen lugar, con un mensaje de error. C#, sin embargo permite al programador intentar recuperarse frente a estas situaciones y así continuar con la ejecución del programa.

3.1. Contrato software

Desde un punto de vista una excepción se puede decir que se basa en el incumplimiento del *contrato software* entre el cliente y el servidor de una clase.

Según este modelo, el usuario (cliente) debe garantizar que se cumplen las condiciones necesarias para una correcta aplicación del software. A estas condiciones se le conocen como **precondiciones**.

El proveedor (servidor), por su parte, debe garantizar que, si el cliente cumple con las precondiciones, el software realizará la tarea deseada satisfactoriamente. Si esto es así, se dice que se cumplen las **postcondiciones**.

3.2. Clases y Corrección del software: Asertos

La corrección de los programas se puede ver en parte como una prueba de que la ejecución terminará proporcionando una salida correcta, siempre que la entrada sea correcta.

Establecer una prueba formal completa de corrección del software es algo muy deseable, pero por desgracia, no se lleva a cabo en la mayoría de las ocasiones. De hecho, la disciplina consistente en plantear asertos apropiados frecuentemente consigue que el programador advierta y evite bastantes errores de programación que antes podrían pasar desapercibidos.

En C# la clase `Debug` contiene el método `Assert()`, invocado de la siguiente manera:

`Assert(`*expresión booleana*`,`*mensaje*`);`

Si la expresión *expresión booleana* se evalúa como falsa, la ejecución proporciona una salida de diagnóstico, con el mensaje *mensaje*. Los asertos se habilitan definiendo la variable `DEBUG`.

Para acceder a dicha salida de diagnóstico se emplean los *listener*.

Ejemplo 3.1 *En el código 3.1 se muestra un ejemplo del uso de asertos, así como del empleo de listeners para visualizar los mensajes de diagnóstico por pantalla.*

```
1  #define DEBUG
2  using System;
3  using System.Diagnostics;
4
5  public class AssertSqrRoot {
6     public static void Main()
7     {
8        Debug.Listeners.Clear();
9        Debug.Listeners.Add(new
10                    TextWriterTraceListener(Console.Out));
11
12       double x;
13       string datos;
14
15       Console.WriteLine("Introduzca un número real positivo:");
16       datos = Console.readLine();
17       x = double.Parse(datos);
18       Console.WriteLine(x);
19       Debug.Assert(x > 0, "Valor no positivo");
20       Console.WriteLine("La raíz cuadrada es " + Math.Sqrt(x));
21    }
22 }
```

Código 3.1: Ejemplo del uso de asertos

El uso de asertos reemplaza el uso *ad hoc* de comprobaciones condicionales con una metodología más uniforme. El inconveniente de los asertos es que no permiten una estrategia de reparación o reintento para continuar con la ejecución "normal" del programa.

3.3. Excepciones en C#

C# contiene la clase estándar `System.Exception`, que es el objeto o la clase base de un objeto lanzado por el sistema o por el usuario cuando ocurre un error en tiempo de ejecución.

El mecanismo de manejo de excepciones en C# es sensible al contexto. Y dicho contexto no es más que un bloque `try`, en el cual se declaran los manejadores (*handlers*) al final del mismo mediante la palabra clave `catch`.

Un código de C# puede alcanzar una excepción en un bloque `try` mediante la expresión `throw`. La excepción es entonces tratada por alguno de los manejadores que se encuentran al final del bloque `try`.

Ejemplo 3.2 *El código 3.2 muestra un ejemplo del uso de excepciones, con un bloque* `try`*, la expresión* `throw` *y los manejadores definidos mediante la palabra clave* `catch`*.*

```
1  using System;
2
3  public class LanzaExcepcion {
4     public static void Main()
5     {
6        try {
7           double x;
8           string datos;
```

```
9           Console.WriteLine("Introduzca un double:");
10          datos = Console.readLine();
11          x = double.Parse(datos);
12          Console.WriteLine(x);
13          if (x > 0)
14            throw(new System.Exception());
15          else Console.WriteLine("La raíz cuadrada es "
16                                        + Math.Sqrt(x));
17      }
18      catch(Exception e)
19       { Console.WriteLine("Lanzada excepción" + e); }
20   }
21 }
```

Código 3.2: Ejemplo del uso de excepciones

La tabla 3.1 muestra algunas excepciones estándar en C#, mientras que en la tabla 3.2 podemos observar las propiedades estándar que toda excepción debe tener.

SystemException	Clase base para las excepciones del sistema
ApplicationException	Clase base para que los usuarios proporcionen errores de aplicación
ArgumentException	Uno o más argumentos son inválidos
ArgumentNullException	Pasado null no permitido
ArgumentOutOfRangeException	Fuera de los valores permitidos
ArithmeticException	Valor infinito o no representable
DivideByZeroException	Auto-explicativa
IndexOutOfRangeException	Índice fuera de los límites del array
InvalidCastException	Cast no permitido
IOException	Clase base en el espacio de nombres System.IO para las excepciones de E/S
NullReferenceException	Intento de referenciar a null
OutOfMemoryException	Ejecución fuera de la memoria *heap* (montón)

Tabla 3.1: Algunas Excepciones estándar

HelpLink	Obtiene o establece un enlace a un fichero de ayuda
InnerException	Obtiene la instancia de `Exception` que causó la excepción
Message	Texto que describe el significado de la excepción
StackTrace	Traza de la pila cuando se llamó a la excepción
Source	Aplicación u objeto que generó la excepción
TargetSize	Método que lanzó la excepción

Tabla 3.2: Propiedades de las excepciones

3.3.1. Lanzamiento de excepciones

Mediante la expresión `throw` se lanzan excepciones, las cuales deben ser objetos de la clase `Exception`. El bloque `try` más interno en el que se lanza una excepción es empleado para seleccionar la sentencia `catch` que procesará dicha excepción.

Si lo que queremos es relanzar la excepción actual podemos emplear un `throw` sin argumentos en el cuerpo de un `catch`. Esta situación puede ser útil si deseamos que un segundo manejador llamado desde el primero realice un procesamiento más exhaustivo de la excepción en cuestión.

Ejemplo 3.3 *El código 3.3 muestra un ejemplo del relanzamiento de excepciones.*

```
1  using System;
2
3  public class ReLanzaExcepcion {
4     public static void LanzaMsg()
5     {
6        try {
7        . . .
8        throw new Exception("Lanzada en LanzaMsg");
9        . . .
10       }
11       catch(Exception e)
12       {
13       . . .
14       Console.WriteLine("Primera captura " + e);
15       throw;  //Relanzamiento
16       }
17    }
18
19    public static void Main()
20    {
21       try {
22       . . .
23       LanzaMsg();
24       . . .
25       }
26       catch(Exception e)
27       { Console.WriteLine("Excepción recapturada " + e); }
28    }
29 }
```

Código 3.3: Ejemplo del relanzamiento de excepciones

Conceptualmente, el lanzamiento de una excepción *pasa* cierta información a los manejadores, pero con frecuencia éstos no precisa de dicha información. Por ejemplo, un manejador que simplemente imprime un mensaje y aborta la ejecución no necesita más información de su entorno.

Sin embargo, el usuario probablemente querrá información adicional por pantalla para seleccionar o ayudarle a decidir la acción a realizar por el manejador. En este caso sería apropiado "empaquetar" la información en un objeto *derivado* de una clase `Exception` ya existente.

Ejemplo 3.4 *El código 3.4 muestra un ejemplo del uso de objetos para empaquetar información que se pasa a los manejadores de excepciones.*

El resultado mostrado por pantalla sería algo como:

```
Out of bounds with last char z
```

```
1  using System;
2
3  public class Stack {
4     public char[] s = new char[100];
5  }
6
7  public class StackError: Exception {
8     public StackError(Stack s, string message)
9       { st = s; msg = message; }
10    public char TopEntry() { return st.s[99]; }
11    public string Msg
12      { set { msg = value; } get { return msg; }}
13    private Stack st;
14    private string msg;
15 }
16
17 public class StackErrorTest {
18    public static void Main()
19    {
20       Stack stk = new Stack;
21       stk.s[99] = 'z';
22       try {
23          throw new StackError(stk,"Out of bounds");
24       }
25       catch(StackError se)
26       {
27          Console.WriteLine(se.Msg + " with last char "
28                            + se.TopEntry());
29       }
30    }
31 }
```

Código 3.4: Uso de excepciones con más información

3.3.2. Bloques `try`

Un bloque `try` es el contexto para decidir qué manejador invocar para cada excepción que se dispara. El orden en el que se definen los manejadores determina el orden en el que se intenta invocar cada manejador que puede tratar la excepción que haya saltado.

Una excepción puede ser tratada por una manejador en concreto si se cumplen alguna de estas condiciones:

1. Si hay una coincidencia exacta.
2. Si la excepción lanzada ha sido derivada de la clase base del manejador.

Es un error listar los manejadores en un orden en el que se impida la ejecución de alguno de ellos, como muestra el siguiente ejemplo.

Ejemplo 3.5 *Según el siguiente código:*

```
catch(ErrorClaseBase e)
catch(ErrorClaseDerivada e)
```

nunca se ejecutaría el cuerpo del segundo `catch`*, pues antes se encontraría una coincidencia con el primer* `catch`*, según lo dicho en las condiciones anteriores.*

3.3.3. Manejadores

La sentencia `catch` parece la declaración de un método con un solo argumento y sin valor de retorno. Incluso existe un `catch` sin ningún argumento, el cual maneja las excepciones no tratadas por los manejadores que sí tienen argumento.

Cuando se invoca a un manejador a través de una expresión `throw`, nos salimos del bloque `try` y automáticamente se llama a los métodos (incluidos los destructores) que liberan la memoria ocupada por todos los objetos locales al bloque `try`.

La palabra clave `finally` después de un bloque `try` introduce otro bloque que se ejecuta después del bloque `try` independientemente de si se ha lanzado una excepción o no.

3.4. Conversión de asertos en excepciones

La lógica de las excepciones es más dinámica ya que los manejadores pueden recibir más información que los asertos, como ya hemos visto anteriormente.

Los asertos simplemente imprimen un mensaje por pantalla, mientras que con las excepciones, además de imprimir más información, se puede decidir (en algunos casos) entre continuar la ejecución del programa o abortarla de inmediato.

Ejemplo 3.6 *El código 3.5 es una reescritura del código 3.1 que se muestra en el ejemplo 3.1 del uso de asertos, pero ahora mediante excepciones.*

En este ejemplo se ve cómo el programa hace uso del mecanismo de excepciones para llamarse a sí mismo recursivamente hasta que el usuario introduzca un número válido para la entrada.

```
1  using System;
2
3  public class ExAssert {
4     public static void MyAssert(bool cond,
5                                 string message, Exception e)
6     {
7        if (!cond) {
8           Console.Error.WriteLine(message);
9           throw e;
10       }
11    }
12 }
13
14 class ExceptionSqrRoot {
15    public static void ConsoleSqrt()
16    {
17       double x;
18       string datos;
19       Console.WriteLine("Introduzca un double positivo:");
```

```
20          datos = Console.readLine();
21          x = double.Parse(datos);
22          Console.WriteLine(x);
23          try {
24             ExAssert.MyAssert(x > 0, "Valor no positivo: x = "
25                                + x.ToString(), new Exception());
26             Console.WriteLine("La raíz cuadrada es " + Math.Sqrt(x
                  ));
27          }
28          catch(Exception e)
29          {
30             Console.WriteLine(e);
31             ExceptionSqrRoot.ConsoleSqrt(); //Se vuelve a intentar
32          }
33       }
34
35       public static void Main()
36       {
37          Console.WriteLine("Probando Raíces Cuadradas");
38          ConsoleSqrt();
39       }
40    }
```

Código 3.5: Ejemplo del uso de excepciones en vez de asertos

Tema 4

Herencia

4.1. Introducción

La herencia es un potente mecanismo para conseguir reutilización en el código, consistente en derivar una nueva clase a partir de una ya existente. A través de la herencia es posible crear una jerarquía de clases relacionadas que compartan código y/o interfaces.

En muchas ocasiones distintas clases resultan ser variantes unas de otras y es bastante tedioso tener que escribir el mismo código para cada una. Una clase derivada hereda la descripción de la clase base, la cual se puede alterar añadiendo nuevos miembros y/o modificando los métodos existentes, así como los privilegios de acceso.

Ejemplo 4.1 *En la figura 4.1 se muestra un ejemplo de una jerarquía de clases.*

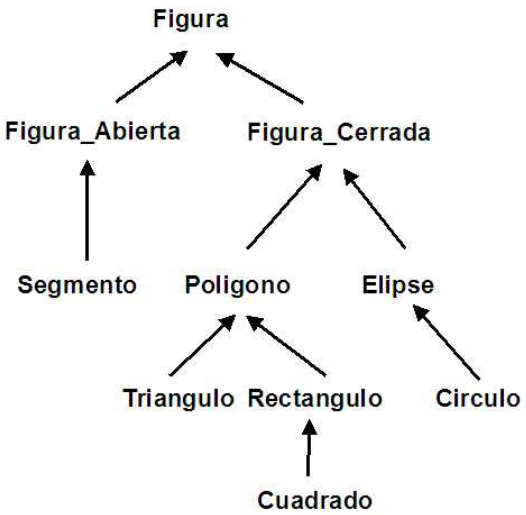

Figura 4.1: Jerarquía de clases

Definición 4.1 *Si B hereda de A entonces B incorpora la estructura (atributos) y comportamiento (métodos) de A, pero puede incluir adaptaciones:*

- *B puede añadir nuevos atributos.*
- *B puede añadir nuevos métodos.*
- *B puede redefinir métodos de A (bien para extender el método original, bien para mejorar la implementación).*
- *B puede implementar un método diferido en A.*

La tabla 4.1 muestra los métodos públicos y protegidos de la clase `object`, raíz en toda jerarquía de clases de C#.

bool Equals(object o)	Devuelve `true` si dos objetos son equivalentes, si no, devuelve `false`
void Finalize()	Equivalente a escribir un destructor
int GetHashCode()	Proporciona un entero único para cada objeto
Type GetType()	Permite averiguar dinámicamente el tipo de objeto
object MemberwiseClone()	Permite clonar un objeto
bool ReferenceEquals (object a, object b)	Devuelve `true` si los objetos son la misma instancia
string ToString()	Devuelve un *string* que representa al objeto actual

Tabla 4.1: Propiedades de las excepciones

4.2. Doble aspecto de la herencia

Como ya se dijo en la definición 2.3, la herencia es, a la vez, un mecanismo de especialización y un mecanismo de implementación.

Esto es, una clase B puede heredar de otra clase A, por ser B una especialización de A. Sin embargo, puede ocurrir que el motivo por el cual B hereda de A sea la necesidad de construir una implementación "distinta" (si se quiere, más refinada) de A.

Ejemplo 4.2 *La herencia entre clases que se observa en la figura 4.1, así como en las figuras 4.2.a y 4.2.b sirve como mecanismo de especialización.*

Por otra parte, los ejemplos de las figuras 4.2.c y 4.2.d muestran la herencia como mecanismo de implementación.

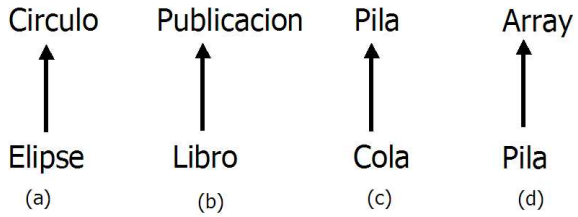

Figura 4.2: Herencia de especialización y de implementación

4.3. Polimorfismo

Definición 4.2 *Podríamos definir el polimorfismo como la capacidad que tiene una entidad para referenciar en tiempo de ejecución a instancias de diferentes clases.*

C# soporta el polimorfismo, lo cual quiere decir que sus entidades (las cuales poseen un único tipo estático y un conjunto de tipos dinámicos) pueden estar conectadas a una instancia de la clase asociada en su declaración o de cualquiera de sus subclases.

Podríamos categorizar el polimorfismo en dos tipos:

✠ Real { Paramétrico / Inclusión (basado en la herencia) } y ✠ Aparente → Sobrecarga

Ejemplo 4.3 *Dada la siguiente declaración, siguiendo la jerarquía de clases de la figura 4.1,*

```
Poligono p;  Triangulo t;  Rectangulo r;  Cuadrado c;
```

podemos decir que:
te(p) = Poligono *ctd(p) = {Poligono, Triangulo, Rectangulo, Cuadrado}*
te(t) = Triangulo *ctd(t) = {Triangulo}*
te(r) = Rectangulo *ctd(r) = {Rectangulo, Cuadrado}*
te(c) = Cuadrado *ctd(c) = {Cuadrado}*
donde "te" significa "tipo estático" y "ctd" significa "conjunto de tipos dinámicos".
Y por lo tanto las siguientes asignaciones serían válidas:

```
p = t;      p = r;      r = c;      p = c;
```

Pero no lo serían, por ejemplo: `t = r;` `r = p;` `c = p;` `c = t;`

4.3.1. Sobrecarga

Definición 4.3 *La sobrecarga consiste en dar distintos significados a un mismo método u operador.*

C# soporta la sobrecarga de métodos y de operadores. Uno de los ejemplos más claros es el del operador '+'.

Ejemplo 4.4 *La expresión* `a + b` *puede significar desde concatenación, si* `a` *y* `b` *son de tipo* `string`*, hasta una suma entera, en punto flotante o de números complejos, todo ello dependiendo del tipo de datos de* `a` *y* `b`*.*

Sin embargo, lo realmente interesante es poder redefinir operadores cuyos operandos sean de tipos definidos por el usuario, y C# permite hacer esto.

Pero conviene tener en cuenta que este tipo de actuación sólo es provechosa en aquellos casos en los que existe una notación ampliamente usada que conforme con nuestra sobrecarga (más aún, si cabe, en el caso de los operadores relacionales <, >, <= y >=).

La tabla 4.2 muestra los operadores unarios que en C# se pueden sobrecargar, mientras que la tabla 4.3 hace lo propio con los operadores binarios.

$+$	$-$
$!$	$\sim$
$++$	$--$
$true$	$false$

Tabla 4.2: Operadores unarios "sobrecargables"

$+$	$-$	$*$	$/$
$\%$	$\&$	$\mid$	$\wedge$
$<<$	$>>$	$==$	$!=$
$<$	$>$	$>=$	$<=$

Tabla 4.3: Operadores binarios "sobrecargables"

Cuando se sobrecarga un operador tal como '+', su operador de asignación asociado, '+=', también es sobrecargado implícitamente. Nótese que el operador de asignación no puede sobrecargarse y que los operadores sobrecargados mantienen la precedencia y asociatividad de los operandos.

Tanto los operadores binarios como los unarios sólo pueden ser sobrecargados mediante métodos estáticos.

Ejemplo 4.5 *El código 4.1 muestra un ejemplo de sobrecarga de operadores, tanto unarios, como binarios.*

Obsérvese como en la línea 32 se vuelve a definir el operador '', previamente sobrecargado por el método de la línea 29, pero con los parámetros en orden inverso. Esta práctica común evita que el operador sobrecargado tenga un orden prefijado en sus operandos.*

El resultado mostrado por pantalla tras ejecutarse el método `Main()` *sería:*

```
Initial times
day 00 time:00:00:59
day 01 time:23:59:59
One second later
day 00 time:00:01:00
day 02 time 00:00:00
```

```
1  using System;
2
3  class MyClock {
4     public MyClock() { }
5     public MyClock(unit i) { Reset(i); }
6
7     public void Reset(unit i) {
8        totSecs = i;
9        secs = i % 60;
10       mins = (i / 60) % 60;
11       hours = (i / 3600) % 24;
12       days = i / 86400;
13    }
14
15    public override string ToString() {
16       Reset(totSecs);
17       return String.Format("day {0:D2} time:{1:D2}:{2:D2}:{3:D2
            }",
18                days, hours, mins, secs);
19    }
20
21    public void Tick() { Reset(++totSecs); }
22
23    public static MyClock operator++(MyClock c)
24        { c.Tick(); return c; }
25
26    public static MyClock operator+(MyClock c1, MyClock c2)
27      { return new MyClock(c1.totSecs + c2.totSecs); }
28
29    public static MyClock operator*(uint m, MyClock c)
30      { return new MyClock(m * c.totSecs); }
31
32    public static MyClock operator*(MyClock c, uint m)
33      { return (m * c); }
34
35    private uint totSecs = 0, secs = 0, mins = 0, hours = 0,
         days = 0;
36 }
```

```
37
38 class MyClockTest {
39    public static void Main() {
40       MyClock a = new MyClock(59), b = new MyClock(172799);
41       Console.WriteLine("Initial times\n" + a + '\n' + b);
42       ++a; ++b;
43       Console.WriteLine("One second later\n" + a + '\n' + b);
44    }
45 }
```

Código 4.1: Ejemplo de Sobrecarga de Operadores

4.3.2. Regla de aplicación de propiedades

Definición 4.4 *Un tipo T1 es compatible con otro tipo T2, si la clase de T1 es la misma que la de T2 o una subclase de T2.*

Definición 4.5 Regla de la Asignación [1]
Una asignación `x = y` *o una invocación* `r(..,y,..)` *a una rutina* `r(..,T x,..)`*, será legal si el tipo de* `y` *es compatible con el tipo de* `x`*.*

Definición 4.6 Regla de Validez de un Mensaje [2]
Un mensaje `ox.r(y)`*, supuesta la declaración* `X x`*, será legal si:*

1. `X` *incluye una propiedad con nombre final* `r`*.*
2. *Los argumentos son compatibles con los parámetros y coinciden en número.*
3. *Y* `r` *está disponible para la clase que incluye el mensaje.*

4.3.3. Estructuras de datos polimórficas

La lista doblemente enlazada implementada en el código 2.7 es un claro ejemplo de estructura de datos polimorfa.

Si en la línea 23 de dicho código en vez de poner `private object data` escribiéramos `private Figura data`, siguiendo el ejemplo de jerarquía de clases de la figura 4.1, en un instante dado podríamos tener una lista como la que se observa en la figura 4.3.

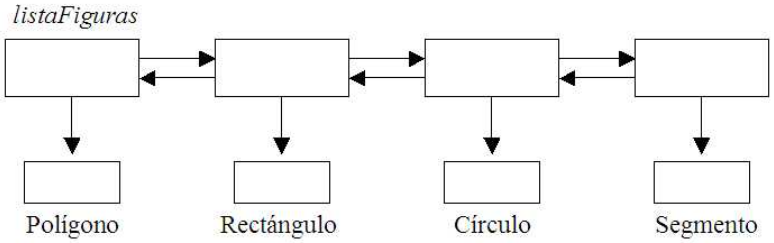

Figura 4.3: Ejemplo de estructura de datos polimorfa

4.3.4. Operadores `is` y `as`

El operador `is`, cuya sintaxis es *expresión* `is` *tipo*, devuelve `true` si se puede hacer un "cast" de la expresión *expresión* al tipo *tipo*.

El operador `as`, con idéntica sintaxis devuelve la expresión *expresión* convertida al tipo *tipo*, o bien, si la conversión no está permitida, devuelve `null`.

[1] En adelante, RA
[2] En adelante, RVM

4.4. Ligadura Dinámica

Como ya dijimos en la definición 4.1, si una clase B hereda de otra clase A, B puede redefinir métodos de A e implementar métodos que en A sean diferidos.

Entonces cuando tengamos un objeto de la clase A y una invocación a un método redefinido en la clase B, la versión del método que se ejecute dependerá del tipo dinámico del objeto de la clase A. En esto consiste la **ligadura dinámica**.

Ejemplo 4.6 *En la figura 4.4 se muestra una jerarquía de clases en las que algunas de ellas redefinen el método* `f` *de sus ancestras. Las letras griegas representan la versión de cada método.*

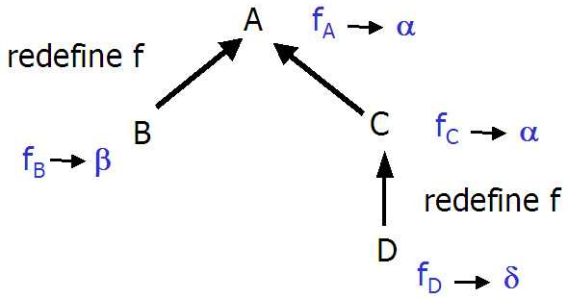

Figura 4.4: Herencia y redefinición de métodos

Dadas las siguientes declaraciones:

```
A oa;   B ob = new B();   C oc = new C();   D od = new D();
```

A continuación se muestra qué versión del método `f` *se ejecutaría, dependiendo del tipo dinámico del objeto* `oa`*:*

`oa = ob; oa.f` $\rightarrow \beta$
`oa = oc; oa.f` $\rightarrow \alpha$
`oa = od; oa.f` $\rightarrow \delta$

En C#, como en C++, la ligadura es estática por defecto, siendo necesario incluir la palabra clave `virtual` en aquellos métodos que vayan a ser redefinidos y a los cuales sí se aplicará ligadura dinámica.

Aquellas subclases que quieran redefinir un método declarado como virtual en la clase base deberán emplear para ello la palabra clave `override`, como ya hemos visto en varios ejemplos con el método `ToString()`.

Nótese la diferencia entre un método redefinido y un método sobrecargado. Los métodos sobrecargados son seleccionados en tiempo de compilación basándonos en sus prototipos (argumentos y valor de retorno) y pueden tener distintos valores de retorno.

Un método redefinido, por contra, es seleccionado en tiempo de ejecución basándonos en el tipo dinámico del objeto receptor y no puede tener distinto valor de retorno del que tenga el método al que redefine.

Ejemplo 4.7 *El código 4.2 muestra un ejemplo de redefinición de un método heredado de la clase base.*

Obsérvese el uso de la palabra clave `virtual` *en la línea 4, así como de la palabra clave* `override` *en la línea 9. Obsérvese también como el valor devuelto por el método redefinido de la línea 9 es el mismo que el del método original de la línea 4, a saber,* `void`*.*

El resultado mostrado por pantalla tras ejecutarse el método `Main()` *sería:*

```
Dentro de la clase base
Dentro de la clase derivada
Dentro de la clase derivada
```

```
1  using System;
2
3  class ClaseBase {
4     public virtual void Status()
5       { Console.WriteLine("Dentro de la clase base"); }
6  }
7
8  class ClaseDerivada : ClaseBase {
9     public override void Status()
10      { Console.WriteLine("Dentro de la clase derivada"); }
11 }
12
13 class LigaduraDinamicaTest {
14    public static void Main()
15    {
16       ClaseBase b = new ClaseBase();
17       ClaseDerivada d = new ClaseDerivada();
18
19       b.Status();
20       d.Status();
21       b = d;
22       b.Status();
23    }
24 }
```

Código 4.2: Redifinición de un método heredado

4.5. Clases Diferidas

Definición 4.7 *Una clase diferida o abstracta es aquella que contiene métodos abstractos, esto es, métodos que deben ser implementados en las subclases. No se pueden crear instancias de una clase abstracta.*

Para declarar en C# un método como abstracto se emplea la palabra clave `abstract`, y para implementarlo en alguna subclase utilizamos la palabra clave `override`.

Si una clase hereda de otra clase abstracta y no implementa todos sus métodos abstractos, entonces sigue siendo abstracta.

Ejemplo 4.8 *El código 4.3 muestra un ejemplo de una clase abstracta* `Figura`*, cuyo método abstracto* `Area()` *es implementado en las subclases* `Rectangulo` *y* `Circulo`*, pero no en la subclase* `Triangulo`*.*

Es por ello que la subclase `Triangulo` *sigue siendo abstracta. Para ser más preciso, es parcialmente abstracta (ver definición 4.8).*

Sin embargo el método abstracto `Perimetro()` *es implementado en las tres subclases.*

```
1  using System;
2
3  abstract class Figura {
4     abstract public double Area();
5     abstract public double Perimetro();
6  }
```

```
7  class Rectangulo : Figura {
8     public Rectangulo(double a, double b)
9       { altura = a; base = b; }
10    public override double Area()
11      { return (base * altura); }
12    public override double Perimetro()
13      { return ((base * 2) + (altura * 2)); }
14    private double altura, base;
15 }
16
17 class Circulo : Figura {
18    public Circulo(double r)
19      { radio = r; }
20    public override double Area()
21      { return (Math.PI * radio * radio); }
22    public override double Perimetro()
23      { return (Math.PI * radio * 2); }
24    private double radio;
25 }
26
27 abstract class Triangulo : Figura {
28    public Triangulo(double l1, double l2, double l3)
29      { lado1 = l1; lado2 = l2; lado3 = l3; }
30    public override double Perimetro()
31      { return (lado1 + lado2 + lado3); }
32    private double lado1, lado2, lado3;
33 }
```

Código 4.3: Clase abstracta Figura

Definición 4.8 *Una clase parcialmente abstracta es aquella que contiene métodos abstractos y efectivos.*

Para referenciar al constructor de la clase base añadimos al final del constructor de la clase derivada lo siguiente: `: base(`*argumentos*`)`. El constructor de la clase base es invocado antes de ejecutar el constructor de la clase derivada.

4.5.1. Interfaces

Definición 4.9 *Una clase que únicamente contiene métodos abstractos recibe el nombre de interface. Pero no confundamos, no es lo mismo que una clase abstracta, pues una clase abstracta puede estar parcialmente implementada.*

Además una clase sólo puede heredar de una única clase abstracta, mientras que en C# está permitida la herencia múltiple de interfaces[3].

Todos los métodos de una interface, además de ser abstractos como ya hemos dicho, deben ser públicos. Por lo tanto las palabras clave `abstract` y `public` se pueden omitir.

Para declarar una interface se emplea la palabra clave `interface`; y para implementar una interface se emplea la notación vista hasta ahora para la herencia.

Una interface puede heredar de otra interface; sin embargo, no puede contener miembros de datos, sólo métodos.

[3] Lo veremos en profundidad más adelante

Ejemplo 4.9 *En el código 4.4 se muestra un ejemplo de una interface* `IPersona`*, de la cual heredan otras dos interfaces* `IEstudiante` *e* `IProfesor`*. Las tres son implementadas por la clase* `Ayudante`*.*

Obsérvese el convenio de nombrar a las interfaces con una 'I' mayúscula delante del nombre, a diferencia de las clases, que no la llevan.

La figura 4.5 muestra el esquema de herencia e implementación de las interfaces y la clase en cuestión.

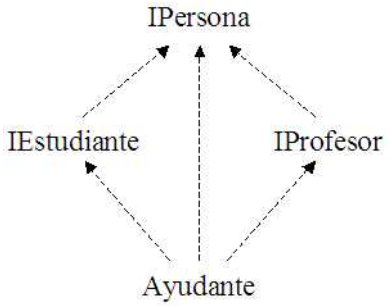

Figura 4.5: Herencia e implementación de interfaces

```
1  interface IPersona {
2     string Nombre { get; set; }
3     string Telefono { get; set; }
4     int Edad { get; set; }
5     bool esMayorDeEdad();
6  }
7
8  interface IEstudiante : IPersona {
9     double NotaMedia { get; set; }
10    int AñoMatricula { get; set; }
11 }
12
13 interface IProfesor : IPersona {
14    double Salario { get; set; }
15    string Nss { get; set; } //N° seguridad social
16 }
17
18 class Ayudante : IPersona, IEstudiante, IProfesor {
19    //Métodos y propiedades requeridos por IPersona
20    public string Name
21      { get { return name; } set { name = value; } }
22    public string Telefono
23      { get { return telefono; } set { telefono = value; } }
24    public int Edad
25      { get { return edad; } set { edad = value; } }
26    public bool esMayorDeEdad()
27      { return (edad >= 18); }
28
29    //Métodos y propiedades requeridos por IEstudiante
30    public double NotaMedia
31      { get { return notaMedia; } set { notaMedia = value; } }
32    public int AñoMatricula
33      { get { return añoMatricula; } set { añoMatricula = value;
          } }
```

```
34
35    //Métodos y propiedades requeridos por IProfesor
36    public double Salario
37      { get { return salario; } set { salario = value; } }
38    public string Nss
39      { get { return nss; } set { nss = value; } }
40
41    //Constructores para Ayudante
42    public Ayudante(string nom, string tlf, int ed, double nm,
43                    int am, double sal, string numSec)
44    {
45       nombre = nom; telefono = tlf; edad = ed;
46       notaMedia = nm; añoMatricula = am;
47       salario = sal; nss = numSec;
48    }
49
50    //Otras implementaciones de constructores y métodos
51    ...
52
53    public override ToString()
54    {
55       return String.Format
56         ("Nombre: {0:-20}  Teléfono: {1,9}  Edad: {2,2}" +
57          "\nSeguridad Social: {3}  Salario: {4:C}" +
58          "\nNota media: {5}  Año de matriculación: {6}",
59          nombre, telefono, edad, nss, salario, notaMedia,
               añoMatricula);
60    }
61
62    private string nombre;
63    private string telefono;
64    private int edad;
65    private double notaMedia;
66    private int añoMatricula;
67    private double salario;
68    private string nss;
69 }
```

Código 4.4: Herencia Múltiple mediante interfaces

4.6. Herencia, reutilización y extensibilidad del software

A continuación enumeramos cuáles son los requerimientos que debe cumplir un módulo para facilitar la reutilización:

1. Variación en tipos. Conseguida gracias a la genericidad.
2. Variación en estructuras de datos y algoritmos. Conseguida gracias a la ligadura dinámica y el polimorfismo.
3. Independencia de la representación. Conseguida gracias a la ligadura dinámica y el polimorfismo.

4. Captura de similitudes entre un un subgrupo de un conjunto de posibles implementaciones. Conseguida gracias a la herencia.

5. Agrupación de rutinas relacionadas. Conseguida mediante la construcción de clases.

Existen dos aspectos importantes en la reutilización de código: por una parte, la creación de componentes reutilizables, y por otra parte, la utilización de dichos componentes.

En C# existe una gran colección (aún en crecimiento) de elementos reutilizables. Nosotros trataremos ahora brevemente el contenido de `System.Collections` y la interface `Collections.IEnumerable`.

La implementación de la interface `IEnumerate` por parte de una clase contenedora dota a ésta de la posibilidad de emplear la instrucción `foreach`, cuya sintaxis es:

`foreach (`*Tipo nombreVariable* `in` *nombreArray*`)`

Mediante la instrucción `foreach` tenemos en cada iteración en la variable *nombreVariable* el contenido de la clase contenedora. Obsérvese que este valor no puede ser modificado, sólo puede consultarse.

Para construir un iterador necesitamos alguna manera de avanzar el iterador, devolver el siguiente valor en la colección y comprobar en cada instante si ya hemos iterado sobre todos los elementos de la colección.

Para ello la interface `IEnumerable` declara el método `IEnumerator GetEnumerator()`. No obstante, para realizar una implementación concreta deben implementarse (valga la redundancia) los métodos `MoveNext()` y `Reset()`, así como el método de acceso `Current`, todos ellos de la interface `IEnumerator`.

Ejemplo 4.10 *El código 4.5 muestra un ejemplo de una clase que implementa la interface* `IEnumerable`*, de modo que se podrá emplear la instrucción* `foreach` *con objetos de dicha clase (como puede observarse en la línea).*

Se trata de un array de 10 enteros cuyo iterador sólo indexa los elementos que ocupan posiciones impares (teniendo en cuenta que la primera posición es la 0, la segunda la 1, ...).

El resultado mostrado por pantalla sería:

```
1 3 5 7 9 11 13 15 17 19
Iterando...
3 7 11 15 19
```

```
1  using System;
2  using System.Collections;
3
4  class ArrayImpar : IEnumerable {
5     public int [] a = new int[10];
6     public IEnumerator GetEnumerator()
7       { return (IEnumerator) new ArrayImparEnumerator(this); }
8
9     public int this [int i]
10      { get { return a[i]; } set { a[i] = value; }
11 }
12
13 private class ArrayImparEnumerator : IEnumerator
14 {
15    public ArrayImparEnumerator(ArrayImpar a)
```

```
16    {
17       this.a = a;
18       indice = -1;
19    }
20
21    public void Reset() { indice = -1; }
22
23    public bool MoveNext()
24    {
25       indice += 2;
26       return (indice <= 9);
27    }
28
29    publoc object Current { get { return a[indice]; } }
30
31    private int indice;
32    private ArrayImpar a;
33 }
34
35 class ArrayImparTest {
36    public static void Main()
37    {
38       ArrayImpar ai = new ArrayImpar();
39       for (int i = 0; i < 10; i++)
40       {
41          ai.a[i] = 2 * i + 1;
42          Console.Write(ai.a[i] + " ");
43       }
44       Console.WriteLine("\nIterando...");
45       foreach (int elemento in ai)
46          Console.Write(elemento + " ");
47    }
48 }
```

Código 4.5: Implementación de un iterador

Tratando ahora la cuestión referente a la ocultación de información en los casos en los que existe herencia diremos que en tales situaciones se viola el *principio de caja negra*, pues puede ocurrir que una subclase apoye algunos de sus métodos en la implementación concreta conocida de la clase base.

Éste es un factor negativo para la extensibilidad, pues si modificamos la clase base, todas sus subclases que hayan actuado de la manera anteriormente indicada se verán afectadas y probablemente también deberán modificar su implementación.

Además de los tres tipos de acceso vistos en la sección 2.1.2 existe otro más: `internal`. Una propiedad con este modificador de acceso sólo puede ser accedida desde código perteneciente al mismo ensamblado en el que se haya definido dicha propiedad.

La herencia privada, que está permitida en C++, no es soportada en C#. Sin embargo, sí se puede inhabilitar la herencia de una clase mediante la palabra clave `sealed`. Este tipo de clases, llamadas *selladas*, pueden ser tratadas de una forma más eficiente por parte del compilador, ya que sus métodos no pueden ser redefinidos y por tanto se declaran automáticamente como no virtuales.

4.7. Herencia múltiple

Definición 4.10 *Se dice que existe herencia múltiple de clases (también puede ser de interfaces) cuando una clase hereda directamente de más de una clase.*

Ejemplo 4.11 *La figura 4.6.(a) muestra un ejemplo de herencia simple, mientras que la figura 4.6.(b) hace lo propio con la herencia múltiple.*

(a) (b)

Figura 4.6: Herencia simple y múltiple

En C#, a diferencia de C++, no está permitida la herencia múltiple de clases. Sin embargo, al igual que en Java, este hecho se solventa mediante la herencia múltiple de interfaces y simple de clases.

Ejemplo 4.12 *En el código 4.4 así como en la figura 4.5 se muestra un claro ejemplo de herencia múltiple de interfaces.*

4.7.1. Problemas: Colisión de nombres y herencia repetida

Aunque la herencia múltiple nos ofrece varias ventajas a la hora de programar según la metodología OO, también implica algunos inconvenientes.

A continuación explicaremos dos de ellos: la colisión de nombres y la herencia repetida (aunque ésta última, como veremos más adelante, no se da en C#).

- La **colisión de nombres** tiene lugar cuando una clase hereda de dos o más clases un método con el mismo nombre y diferente implementación.

Ejemplo 4.13 *En la figura 4.7 se muestra un ejemplo de colisión de nombres debida a la herencia múltiple.*

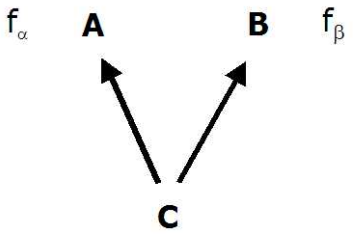

Figura 4.7: Herencia múltiple y colisión de nombres

- La **herencia repetida** se da cuando una misma propiedad es heredada por distintos caminos más de una vez.

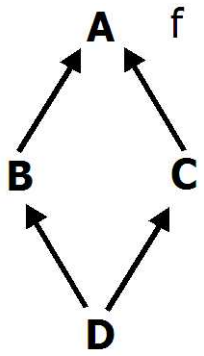

Figura 4.8: Herencia múltiple y herencia repetida

Ejemplo 4.14 *En la figura 4.8 se muestra un ejemplo de herencia repetida debida a la herencia múltiple.*

En C#, como ya hemos dicho, no hay herencia repetida, pero sí puede darse colisión de nombres en el caso en el que una interface hereda de dos o más interfaces un método con el mismo nombre.

En este caso existen dos posibilidades:

1. Sobrecarga, si difieren en la signatura y/o en el valor de retorno.
2. Compartición, si tienen la misma signatura y valor de retorno.

4.7.2. Ejemplos de utilidad

Ejemplo 4.15 *En el código 4.6 se muestra un ejemplo en el que una clase* `C` *hereda de dos interfaces distintas* `A` *y* `B` *un método con el mismo nombre* `f(object o)`*, resultando en una ambigüedad de nombres.*

```
1  interface A {
2     void f(object o);
3  }
4
5  interface B {
6     void f(object o);
7  }
8
9  class C : A, B {
10    //A.f() o B.f()??
11    public void g() { f(); }
12 }
```

Código 4.6: Ambigüedad de nombres debida a la herencia múltiple

Podemos resolver la potencial ambigüedad de nombres dentro de la clase `C` *incluyendo declaraciones explícitas para cada instancia de interface heredada. Es más, también podemos implementar un método* `f()` *en* `C` *para el caso en el que se manipulen directamente objetos de dicha clase. Estos cambios pueden observarse en el código 4.7.*

```
1  class C : A, B {
2     public void f(string s) { ... }
3     public void A.f(object o) { ... }
4     public void B.f(object o) { ... }
5  }
```

Código 4.7: Solución a la ambigüedad de nombres

Ejemplo 4.16 *Sea la jerarquía mostrada en la figura 4.9. Las letras enmarcadas representan interfaces y las no enmarcadas, clases.*

En esta circunstancia podríamos pensar que se daría un caso de herencia repetida, pero no es así.

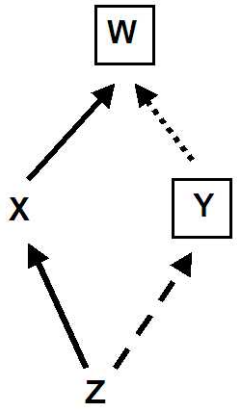

Figura 4.9: Herencia múltiple sin herencia repetida (I)

La clase `Z` *no necesita implementar los métodos de la interface* `W`*, pues ya hereda dichas implementaciones de la clase* `X`*. No obstante sí deberá implementar aquellos métodos de la interface* `Y` *que no estén en la interface* `W`*.*

Ejemplo 4.17 *Sea ahora la jerarquía mostrada en la figura 4.10. Al igual que en el ejemplo anterior, podríamos pensar que bajo estas premisas nos encontramos ante un caso de herencia múltiple. Pero, al igual que en el ejemplo anterior esto no es así.*

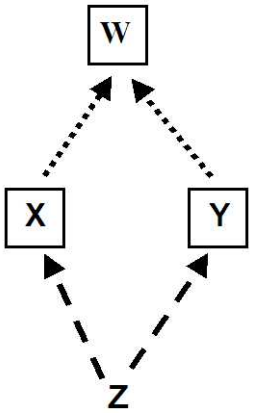

Figura 4.10: Herencia múltiple sin herencia repetida (II)

Si bien es cierto que la clase `Z` *hereda los métodos de la interface* `W` *por dos vías, no resulta esto ser un problema, pues como ya vimos en el ejemplo 4.15, se puede solucionar fácilmente.*

Otra solución distinta a la expuesta en el ejemplo 4.15 pero igualmente válida consiste en implementar una única vez cada método "repetido".

4.8. C#, Java y C++: Estudio comparativo

Además de las continuas referencias que durante todo el texto se han hecho hacia otros lenguajes OO como Java y C++, a continuación se muestran sólo algunas comparaciones entre dichos lenguajes y C#.

- C# es muy parecido a Java. Normalmente el recolector de basura (*garbage collector*) se encarga automáticamente de los objetos que ya no son necesarios (por ejemplo, los no

referenciados por ninguna variable). Toda clase tiene como clase base de su jerarquía a la clase `object`.

- En C# todo es un objeto. Los tipos primitivos tales como `int` pueden ser tratados como objetos. No es este el caso de Java donde los tipos primitivos son estrictamente "tipos valor". C# permite paso por referencia en los tipos primitivos.
- C# incorpora la instrucción `foreach` junto con la interface `IEnumerator`, lo cual permite la construcción de un iterador. C# tiene verdaderos arrays multidimensionales.
- C# ha retenido, aunque de una forma más lógica y simple, la habilidad de C++ para sobrecargar operadores. En C# existe el concepto de propiedades, lo cual proporciona los métodos de acceso `get` y `set` para consultar y modificar los valores de los miembros de datos de una clase.
- Un programador de C# puede convertir rápidamente su código a Java, pero no a la inversa. De hecho Java es conceptualmente un subconjunto de C#.
- C++ es muy complejo e inseguro. Ofrece demasiadas oportunidades al programador de manejar erróneamente los punteros en aras de la eficiencia.
- C++ es dependiente del sistema. No está preparado para la Web. No maneja la memoria como lo hacen C# o Java. El mero hecho de que el núcleo de C# esté desprovisto del tipo puntero simplifica enormemente su compresión por parte de un estudiante de POO.

Índice de Figuras

2.1. Clientela y Herencia entre A y B 8
2.2. Variables en los lenguajes tradicionales 11
2.3. Variables tipo “puntero” en los lenguajes tradicionales 11
2.4. Variables en los lenguajes OO 11
2.5. Ejemplo de copia superficial 12
2.6. Ejemplo de copia profunda 12
2.7. Ejemplo de variables idénticas 13

4.1. Jerarquía de clases 25
4.2. Herencia de especialización y de implementación 26
4.3. Ejemplo de estructura de datos polimorfa 29
4.4. Herencia y redefinición de métodos 30
4.5. Herencia e implementación de interfaces 33
4.6. Herencia simple y múltiple 37
4.7. Herencia múltiple y colisión de nombres 37
4.8. Herencia múltiple y herencia repetida 38
4.9. Herencia múltiple sin herencia repetida (I) 39
4.10. Herencia múltiple sin herencia repetida (II) 39

Índice de Tablas

2.1. Tipos enteros primitivos 6
2.2. Tipos flotantes primitivos 6
2.3. Otros tipos primitivos 6
2.4. Palabras reservadas 6

3.1. Algunas Excepciones estándar 19
3.2. Propiedades de las excepciones 19

4.1. Propiedades de las excepciones 26
4.2. Operadores unarios “sobrecargables” 27
4.3. Operadores binarios “sobrecargables” 27

Índice de Códigos

2.1. Clase Luna 5
2.2. Ejemplo de Ocultación de Información 7
2.3. Otro ejemplo de Ocultación de Información 8
2.4. Ejemplo de Clases Anidadas 9
2.5. Ejemplo de Constructores 12
2.6. Ejemplo de implementación del método `Clone()` 13
2.7. Lista genérica doblemente enlazada 14
3.1. Ejemplo del uso de asertos 18
3.2. Ejemplo del uso de excepciones 18
3.3. Ejemplo del relanzamiento de excepciones 20
3.4. Uso de excepciones con más información 21
3.5. Ejemplo del uso de excepciones en vez de asertos 22
4.1. Ejemplo de Sobrecarga de Operadores 28
4.2. Redifinición de un método heredado 31
4.3. Clase abstracta Figura 31
4.4. Herencia Múltiple mediante interfaces 33
4.5. Implementación de un iterador 35
4.6. Ambigüedad de nombres debida a la herencia múltiple 38
4.7. Solución a la ambigüedad de nombres 38

Bibliografía

[Lip02] Stanley B. Lippman. *C# Primer. A Practical Approach.* Addison Wesley, 2002.

[Mol04] J. García Molina. Apuntes de programación orientada a objetos. 2004.

[Poh03] Ira Pohl. *C# by Dissection.* Addison Wesley, University of California, 2003.