TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN UTN-FRC

TUP

UTN

Facultad Regional Córdoba

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

PROGRAMACIÓN II

Unidad Temática 2: Polimorfismo

Material de Estudio

1er Año - 2er Cuatrimestre

2019

V.0.1

Índice

Repaso de Herencia

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que, si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera -a la que se le suele llamar **clase hija**- será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda –a la que se le suele llamar **clase padre** o **clase base**. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <nombreHija>: <nombrePadre>
{
     <nombreHija>
}
```

A los miembros definidos en se les añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase Persona puede crearse una clase Trabajador así:

```
class Trabajador:Persona
{
     public int Sueldo;
     public Trabajador(string nombre, int edad, string nif, int sueldo):
             base(nombre, edad, nif)
     {
             Sueldo = sueldo;
     }
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que cada trabajador gane. Nótese además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

```
: base(<parametroBase>)
```

A esta estructura se le llama **inicializador base** y se utiliza para indicar cómo deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto que vale **:base()**, lo que sería incorrecto en este ejemplo debido a que Persona carece de constructor sin parámetros.

Un ejemplo que pone de manifiesto cómo funciona la herencia es el siguiente:

```
using System;

class Persona
{
     public string Nombre;       // Campo de cada objeto Persona que almacena su nombre
     public int Edad;            // Campo de cada objeto Persona que almacena su edad
     public string NIF;          // Campo de cada objeto Persona que almacena su NIF

     void Cumpleaños()   // Incrementa en uno de edad del objeto Persona
     {
             Edad++;
     }

     public Persona (string nombre, int edad, string nif) // Constructor de Persona
     {
             Nombre = nombre;
             Edad = edad;
             NIF = nif;
     }
}

class Trabajador: Persona
{
     public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

     Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)
     {       // Inicializamos cada Trabajador en base al constructor de Persona
             Sueldo = sueldo;
     }
```

```
        public static void Main()
        {
                Trabajador p = new Trabajador("Juan", 22, "77588260-Z", 100000);
                Console.WriteLine ("Nombre="+p.Nombre);
                Console.WriteLine ("Edad="+p.Edad);
                Console.WriteLine ("NIF="+p.NIF);
                Console.WriteLine ("Sueldo="+p.Sueldo);
        }
}
```

Nótese que ha sido necesario prefijar la definición de los miembros de Persona de la palabra reservada public. Esto se debe a que por defecto los miembros de una tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo public conseguimos que sean accesibles desde cualquier código, como el método Main() definido en Trabajador. public es lo que se denomina un **modificador de acceso**, concepto que se tratará más adelante en este mismo tema bajo el epígrafe titulado Modificadores de acceso.

Llamadas por defecto al constructor base

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base el compilador considerará que éste es **:base()** Por ello hay que estar seguros de que si no se incluye base en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

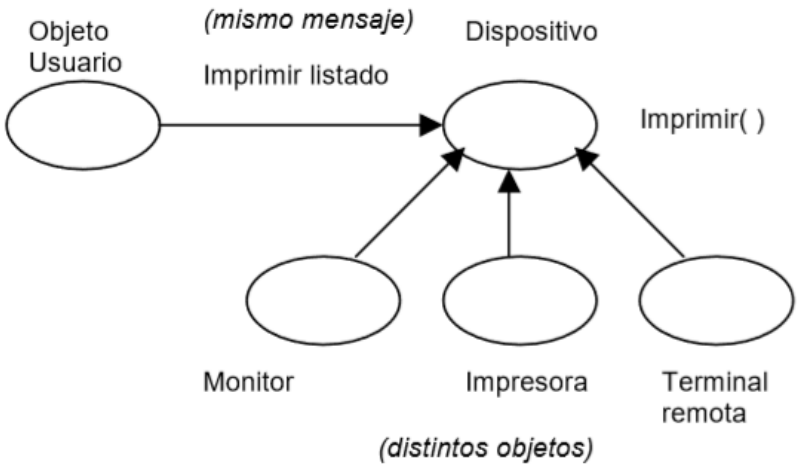
Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

```
<nombreClase>(): base()
{}
```

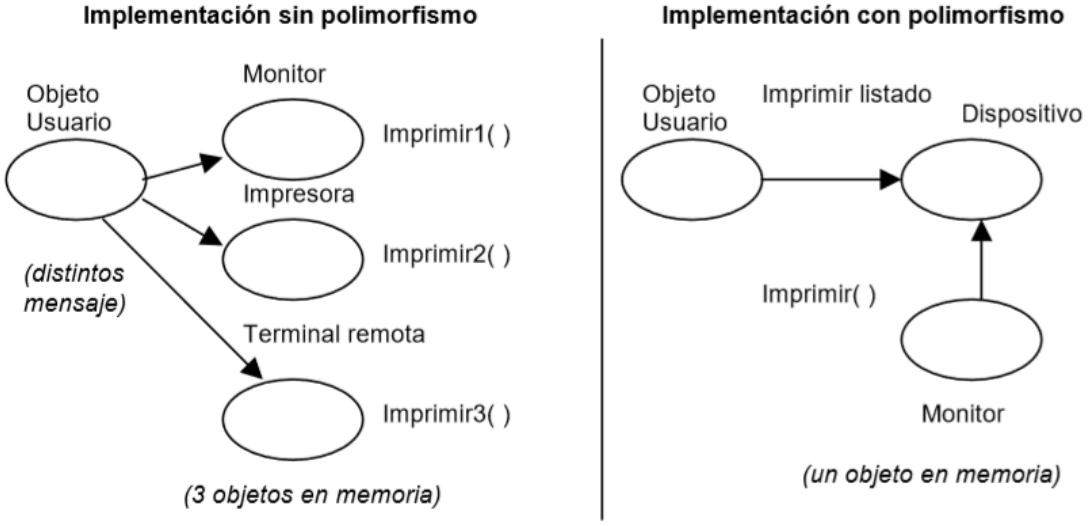
Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

Polimorfismo

El polimorfismo es otro de los pilares fundamentales de la programación orientada a objetos. Es la capacidad de los objetos de responder a los mismos mensajes de distintas formas. Por ejemplo: un usuario de un sistema tiene que imprimir un listado de sus clientes, él puede elegir imprimirlo en el monitor, en la impresora, en un archivo en disco o en una terminal remota. De esta forma el mensaje es imprimir el listado, pero la respuesta la dará el objeto que el usuario desee, el monitor, la impresora, el archivo o la terminal remota. Cada uno responderá con métodos (comportamientos) diferentes.



El polimorfismo permite mejorar la implementación de los programas. Por ejemplo, la implementación del ejemplo anterior sin usar polimorfismo, implicaría crear la cantidad de objetos que podría necesitar el usuario, si bien el mismo solo va a usar uno por vez, tiene que tenerlos a todos en la memoria para que cuando requiera el listado correspondiente, tenga acceso a cualquier salida. En cambio, si se utiliza polimorfismo, en memoria sólo se requerirá en forma permanente una referencia al dispositivo (genérico) y cuando el usuario decida imprimir el listado, elegirá en que dispositivo lo hará (monitor, impresora o terminal remota), por lo tanto, en tiempo de ejecución se decide cual es el objeto que estará activo en el momento de la impresión, dejando en memoria solo el objeto que se requiera y no los n posibles.



Otras definiciones

- Es la capacidad de almacenar objetos de un determinado tipo en variables de tipos antecesores del primero a costa, claro está, de sólo poderse acceder a través de dicha variable a los miembros comunes a ambos tipos (métodos polimórficos).
- Dos objetos son polimórficos, respecto de un conjunto de mensajes, si ambos son capaces de responderlos. En otras palabras, podemos decir que, si dos objetos son capaces de responder al mismo mensaje, aunque sea haciendo cosas diferentes, entonces son polimórficos respecto de ese mensaje.

Ejemplos

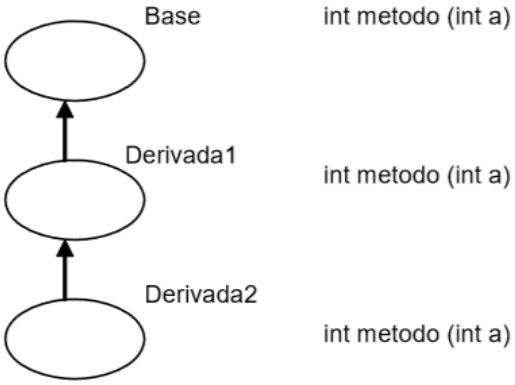
- Supongamos que en un programa debemos trabajar con figuras geométricas y construimos algunos objetos que son capaces de representar un rectángulo, un triángulo y un circulo. Si nos interesa que todos puedan dibujarse en la pantalla, podemos dotar a cada uno de un método llamado “dibujar” que se encargue de realizar esta tarea y entonces vamos a poder dibujar tanto un circulo, como un cuadrado o un triángulo simplemente enviándole al mensaje “dibujar” al objeto que queremos mostrar en la pantalla. En otras palabras, vamos a poder hacer X.dibujar() sin importar si X es un cuadrado un circulo o un triángulo, “circulo”, como los “triangulo” y los “cuadrado” son polimórficos respecto del mensaje dibujar. Es importante notar que no es lo mismo dibujar una figura que otra, seguramente, cada figura implementa esa método de modo diferente.
- Otro ejemplo: supongamos que en un sistema de un banco tenemos objetos que representas cuentas corrientes y cajas de ahorro. Si bien son diferentes, en ambas pueden hacerse depósitos y extracciones, entonces podemos hacer que cada objeto implemente un método “extraer” y uno “depositar” para que puedan responder a esos mismos mensajes. Sin duda, van a estar implementados de modo diferente porque no se hacen los mismos pasos para extraer o depositar dinero de una cuenta corriente que de una caja de ahorro, pero ambos objetos van a poder responder e esos mensajes y entonces serán polimorfitos entre sí respecto del conjunto de mensajes formado por “depositar” y “extraer”. La gran ventaja es que ahora podemos hacer X.depositar() y confiar en que se realizara un depósito en la cuenta que corresponde sin que tengamos que saber a priori si X es una cuenta corriente o una caja de ahorro.

Para aplicar polimorfismo hay ciertos conceptos que debemos aprender, ellos son:

Clases abstractas: Son clases genéricas, sirven para agrupar clases del mismo género, no se pueden instanciar, es decir no se pueden crear objetos a partir de ellas, ya que representan conceptos tan generales que no se puede definir cómo será la implementación de la misma. Es por eso que dichas clases llevan métodos que se llaman abstractos, que no tienen implementación, una clase abstracta debe tener al menos un método abstracto.

Métodos abstractos: Métodos que no tienen implementación, Se utilizan para recibir los mensajes en común. No tiene implementación porque en el nivel donde se definen no hay información suficiente para implementarlos. En los próximos niveles de la jerarquía se pueden implementar, para ello deben ser redefinidos en niveles posteriores.

Métodos redefinidos: Métodos que tienen la misma firma (tipo, nombre y argumentos) en los distintos niveles de una jerarquía.



Implementación

1. Diseñar e implementar las clases: Diseñar una jerarquía de clases con las clases que intervendrán en el polimorfismo, dicha jerarquía debe tener una clase abstracta como clase base y tantas clases derivadas de ella, como haga falta.
2. Elegir él o los métodos que recibirán el mismo mensaje, definirlo en la clase base como abstracto y redefinirlo en las clases derivadas con la implementación correspondiente.
3. En la clase usuaria, puede ser la que lleve el método main(), realizar lo siguiente:
    1. Definir una variable de la clase base abstracta
    2. Solicitarle al usuario que elija que objeto va a crear
    3. Crear el objeto de acuerdo a la opción del usuario, la referencia de dicho objeto se guarda en la variable creada en primera instancia.

Polimorfismo en C#

Métodos virtuales

Ya hemos visto que es posible definir tipos cuyos métodos se hereden de definiciones de otros tipos. Lo que ahora vamos a ver es que además es posible cambiar dicha definición en la clase hija, para lo que habría que haber precedido con la palabra reservada **virtual** la definición de dicho método en la clase padre. A este tipo de métodos se les llama **métodos virtuales**, y la sintaxis que se usa para definirlos es la siguiente:

```
virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
    <código>
}
```

Si en alguna clase hija quisiésemos dar una nueva definición del <código> del método, simplemente lo volveríamos a definir en la misma, pero sustituyendo en su definición la palabra reservada **virtual** por **override**. Es decir, usaríamos esta sintaxis:

```
override <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
    <nuevoCódigo>
}
```

Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en la clase padre el método fue definido como **virtual**. En caso contrario, el compilador considerará un error intentar redefinirlo.

El lenguaje C# impone la restricción de que toda redefinición de método que queramos realizar incorpore la partícula **override** para forzar a que el programador esté seguro de que verdaderamente lo que quiere hacer es cambiar el significado de un método heredado. Así se evita que por accidente defina un método del que ya exista una definición en una clase padre. Además, C# no permite definir un método como **override** y **virtual** a la vez, ya que ello tendría un significado absurdo: estaríamos dando una redefinición de un método que vamos a definir.

Por otro lado, cuando definamos un método como **override** ha de cumplirse que en alguna clase antecesora (su clase padre, su clase abuela, etc.) de la clase en la que se ha realizado la definición del mismo exista un método virtual con el mismo nombre que el redefinido. Si no, el compilador informará de error por intento de redefinición de método no existente o no virtual. Así se evita que por accidente un programador crea que está redefiniendo un método del que no exista definición previa o que redefina un método que el creador de la clase base no desee que se pueda redefinir.

Para aclarar mejor el concepto de método virtual, vamos a mostrar un ejemplo en el que cambiaremos la definición del método Cumpleaños() en los objetos Persona por una nueva versión en la que se muestre un mensaje cada vez que se ejecute, y redefiniremos dicha nueva versión para los objetos Trabajador de modo que el mensaje mostrado sea otro. El código de este ejemplo es el que se muestra a continuación:

```
using System;

class Persona
{
     public string Nombre;
     public int Edad;
     public string DNI;

     public virtual void Cumpleaños()
     {
             Edad++;
             Console.WriteLine(“Incrementada edad de persona”);
     }
     public Persona (string nombre, int edad, string dni) // Constructor de Persona
     {
             Nombre = nombre;
             Edad = edad;
             DNI = dni;
     }
}
class Trabajador: Persona
{
     public int Sueldo;

     Trabajador(string nombre, int edad, string dni, int sueldo): base(nombre, edad, dni)
     {       // Inicializamos cada Trabajador en base al constructor de Persona
             Sueldo = sueldo;
     }
     public override void Cumpleaños()
     {
             Edad++;
             Console.WriteLine(“Incrementada edad de trabajador”);
     }
     public static void Main()
     {
             Persona p = new Persona("Carlos", 22, "77588261-Z");
             Trabajador t = new Trabajador("Juan", 22, "77588260-Z", 100000);

             t.Cumpleaños();
             p.Cumpleaños();
     }
}
```

Nótese cómo se ha añadido el modificador **virtual** en la definición de Cumpleaños() en la clase Persona para habilitar la posibilidad de que dicho método puede ser redefinido en clase hijas de Persona y cómo se ha añado **override** en la redefinición del mismo dentro de la clase Trabajador para indicar que la nueva definición del método es una redefinición del heredado de la clase. La salida de este programa confirma que la implementación de **Cumpleaños**() es distinta en cada clase, pues es de la forma:

```
Incrementada edad de trabajador
Incrementada edad de persona
```

También es importante señalar que para que la redefinición sea válida ha sido necesario añadir la partícula **public** a la definición del método original, pues si no se incluyese se consideraría que el método sólo es accesible desde dentro de la clase donde se ha definido, lo que no tiene sentido en métodos virtuales ya que entonces nunca podría ser redefinido. De hecho, si se excluyese el modificador **public** el compilador informaría de un error ante este absurdo. Además, este modificador también se ha mantenido en la redefinición de Cumpleaños() porque toda redefinición de un método virtual ha de mantener los mismos modificadores de acceso que el método original para ser válida.

Clases abstractas

Una **clase abstracta** es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos (esto último se verá más adelante en este mismo tema) Para definir una clase abstracta se antepone **abstract** a su definición, como se muestra en el siguiente ejemplo:

```
public abstract class A
{
     public abstract void F();
}
abstract public class B: A
{
     public void G() {}
}
class C: B
{
     public override void F()
     {}
}
```

Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador **abstract** con modificadores de acceso.

La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación, sino que se deje en manos de sus clases hijas darla. No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador **abstract** y sustituyendo su código por un punto y coma (;), como se muestra en el método F() de la clase A del ejemplo (nótese que B también ha de definirse como abstracta porque tampoco implementa el método F() que hereda de A)

Obviamente, como un método abstracto no tiene código no es posible llamarlo. Hay que tener especial cuidado con esto a la hora de utilizar **this** para llamar a otros métodos de un mismo objeto, ya que llamar a los abstractos provoca un error al compilar.

Véase que todo método definido como abstracto es implícitamente virtual, pues si no sería imposible redefinirlo para darle una implementación en las clases hijas de la clase abstracta donde esté definido. Por ello es necesario incluir el modificador **override** a la hora de darle implementación y es redundante marcar un método como **abstract** y **virtual** a la vez (de hecho, hacerlo provoca un error al compilar)

Es posible marcar un método como **abstract** y **override** a la vez, lo que convertiría al método en abstracto para sus clases hijas y forzaría a que éstas lo tuviesen que re implementar si no se quisiese que fuesen clases abstractas.

Un ejemplo

A continuación, se muestra un ejemplo de cómo una variable de tipo Persona puede usarse para almacenar objetos de tipo Trabajador. En esos casos el campo Sueldo del objeto referenciado por la variable no será accesible, y la versión del método Cumpleaños() a la que se podría llamar a través de la variable de tipo Persona sería la definida en la clase Trabajador, y no la definida en Persona:

```
using System;

class Persona
{
     public string Nombre;
     public int Edad;
     public string DNI;

     public virtual void Cumpleaños() // Incrementa en uno la edad del objeto Persona
     {
        Console.WriteLine("Incrementada edad de persona");
     }
     public Persona (string nombre, int edad, string dni) // Constructor de Persona
     {
             Nombre = nombre;
             Edad = edad;
             DNI = dni;
     }
 }
class Trabajador: Persona
```

```
{
     int Sueldo;
     Trabajador(string nombre, int edad, string dni, int sueldo): base(nombre, edad, dni)
     {
        Sueldo = sueldo;
     }
     public override Cumpleaños()
     {
          Edad++;
          Console.WriteLine("Incrementada edad de trabajador");
     }
     public static void Main()
     {
             Persona p = new Trabajador("Juan", 22, "77588260-Z", 100000);
             p.Cumpleaños();
             // p.Sueldo++; //ERROR: Sueldo no es miembro de Persona
     }
}
```

El mensaje mostrado por pantalla al ejecutar este método confirma lo antes dicho respecto a que la versión de Cumpleaños() a la que se llama, ya que es: Incrementada edad de trabajador

Acceso a la clase base

Hay determinadas circunstancias en las que cuando redefinamos un determinado método nos interese poder acceder al código de la versión original. Por ejemplo, porque el código redefinido que vayamos a escribir haga lo mismo que el original y además algunas cosas extras. En estos casos se podría pensar que una forma de conseguir esto sería convirtiendo el objeto actual al tipo del método a redefinir y entonces llamar así a ese método, como por ejemplo en el siguiente código:

```
using System;
class A
{
     public virtual void F()
     {
        Console.WriteLine("A");
     }
}
class B:A
{
     public override void F()
     {
             Console.WriteLine("Antes");
             ((A) this).F();    // (2)
             Console.WriteLine("Después");
     }
     public static void Main()
     {
             B b = new B();
             b.F();
     }
}
```

Pues bien, si ejecutamos el código anterior veremos que la aplicación nunca termina de ejecutarse y está constantemente mostrando el mensaje Antes por pantalla. Esto se debe a que debido al polimorfismo se ha entrado en un bucle infinito: aunque usemos el operador de conversión para tratar el objeto como si fuese de tipo A, su verdadero tipo sigue siendo B, por lo que la versión de F() a la que se llamará en (2) es a la de B de nuevo, que volverá a llamarse a sí misma una y otra vez de manera indefinida.

Para solucionar esto, los diseñadores de C# han incluido una palabra reservada llamada **base** que devuelve una referencia al objeto actual semejante a **this** pero con la peculiaridad de que los accesos a ella son tratados como si el verdadero tipo fuese el de su clase base. Usando base, podríamos reemplazar el código de la redefinición de F() de ejemplo anterior por:

```
public override void F()
{
     Console.WriteLine("Antes");
     base.F();
     Console.WriteLine("Después");
}
```

Si ahora ejecutamos el programa veremos que ahora sí que la versión de F() en B llama a la versión de F() en A, resultando la siguiente salida por pantalla:

```
Antes
A
Después
```

A la hora de redefinir métodos abstractos hay que tener cuidado con una cosa: desde el método redefinido no es posible usar **base** para hacer referencia a métodos abstractos de la clase padre, aunque sí para hacer referencia a los no abstractos. Por ejemplo:

```
abstract class A
{
     public abstract void F();
     public void G()
     {}
}

class B: A
{
     public override void F()
     {
             base.G();       // Correcto
             base.F();       // Error, base.F() es abstracto
     }
}
```

Determinación de tipo. Operador is

Dentro de una rutina polimórfica que, como la del ejemplo anterior, admita parámetros que puedan ser de cualquier tipo, muchas veces es conveniente poder consultar en el código de la misma cuál es el tipo en concreto del parámetro que se haya pasado al método en cada llamada al mismo. Para ello C# ofrece el operador **is**, cuya forma sintaxis de uso es:

```
<expresión> is <nombreTipo>
```

Este operador devuelve **true** en caso de que el resultado de evaluar <expresión> sea del tipo cuyo nombre es <nombreTipo> y **false** en caso contrario[1]. Gracias a ellas podemos escribir métodos genéricos que puedan determinar cuál es el tipo que tienen los parámetros que en cada llamada en concreto se les pasen. O sea, métodos como:

1 Si la expresión vale null se devolverá false, pues este valor no está asociado a ningún tipo en concreto.

```
public void MétodoGenérico(object o)
{
     if  (o is int)          // Si o es de tipo int (entero)...
             // ...Código a ejecutar si el objeto o es de tipo int
     else if (o is string)   // Si no, si o es de tipo string (cadena)...
             // ...Código a ejecutar si o es de tipo string
     //... Ídem para otros tipos
}
```

El bloque **if...else** es una instrucción condicional que permite ejecutar un código u otro en función de si la condición indicada entre paréntesis tras el **if** es cierta (**true**) o no (**false**).

Downcasting

Si se tiene la necesidad de manejar un objeto por su tipo, luego de determinar cuál es el mismo (por ejemplo, con el operador **is**), hay que hacer una conversión del tipo padre al verdadero tipo del objeto, y a esto se le llama **downcasting**

Para realizar un downcasting una primera posibilidad es indicar preceder la expresión a convertir del tipo en el que se la desea convertir indicado entre paréntesis. Es decir, siguiendo la siguiente sintaxis:

```
(<tipoDestino>) <expresiónAConvertir>
```

El resultado de este tipo de expresión es el objeto resultante de convertir el resultado de <expresiónAConvertir> a <tipoDestino>. En caso de que la conversión no se pudiese realizar se lanzaría una excepción del tipo predefinido **System.InvalidCastException**

Otra forma de realizar el downcasting es usando el operador **as**, que se usa así:

```
 <expresiónAConvertir> as <tipoDestino>
```

La principal diferencia de este operador con el anterior es que si ahora la conversión no se pudiese realizar se devolvería **null** en lugar de lanzarse una excepción. La otra diferencia es que as sólo es aplicable a tipos referencia y sólo a conversiones entre tipos de una misma jerarquía (de padres a hijos o viceversa)

Los errores al realizar conversiones de este tipo en métodos genéricos se producen cuando el valor pasado a la variable genérica no es ni del tipo indicado en <tipoDestino> ni existe ninguna definición de cómo realizar la conversión a ese tipo.

Clases y métodos sellados

Una clase sellada es una clase que no puede tener clases hijas, y para definirla basta anteponer el modificador sealed a la definición de una clase normal. Por ejemplo:

```
sealed class ClaseSellada
{  }
```

Una utilidad de definir una clase como sellada es que permite que las llamadas a sus métodos virtuales heredados se realicen tan eficientemente como si fuesen no virtuales, pues al no poder existir clases hijas que los redefinan no puede haber polimorfismo y no hay que determinar cuál es la versión correcta del método a la que se ha de llamar. Nótese que se ha dicho métodos virtuales heredados, pues lo que no se permite es definir miembros virtuales dentro de este tipo de clases, ya que al no poderse heredarse de ellas es algo sin sentido en tanto que nunca podrían redefinirse.

Ahora bien, hay que tener en cuenta que sellar reduce enormemente su capacidad de reutilización, y eso es algo que el aumento de eficiencia obtenido en las llamadas a sus métodos virtuales no suele compensar. En realidad, la principal causa de la inclusión de estas clases en C# es que permiten asegurar que ciertas clases críticas nunca podrán tener clases hijas y sus variables siempre almacenarán objetos del mismo tipo. Por ejemplo, para simplificar el funcionamiento del CLR y los compiladores se ha optado por hacer que todos los tipos de datos básicos excepto **System.Object** estén sellados.

Téngase en cuenta que es absurdo definir simultáneamente una clase como **abstract** y **sealed**, pues nunca podría accederse a la misma al no poderse crear clases hijas suyas que definan sus métodos abstractos. Por esta razón, el compilador considera erróneo definir una clase con ambos modificadores a la vez.

Aparte de para sellar clases, también se puede usar **sealed** como modificador en la redefinición de un método para conseguir que la nueva versión del mismo que se defina deje de ser virtual y se le puedan aplicar las optimizaciones arriba comentadas. Un ejemplo de esto es el siguiente:

```
class A
{
     public abstract F();
}
class B:A
{
     public sealed override F() // F() deja de ser redefinible
     {}
}
```

Propiedades abstractas

Las propiedades participan del mecanismo de polimorfismo igual que los métodos, siendo incluso posible definir propiedades cuyos bloques de código **get** o **set** sean abstractos. Esto se haría prefijando el bloque apropiado con un modificador **abstract** y sustituyendo la definición de su código por un punto y coma. Por ejemplo:

```
using System;

abstract class A
{
     public abstract int PropiedadEjemplo
     {
              set;
              get;
     }
}
class B:A
{
     private int valor;

     public override int PropiedadEjemplo
     {
              get
              {
                       Console.WriteLine("Leído {0} de PropiedadEjemplo", valor);
                       return valor;
              }
              set
              {
                       valor = value;
                       Console.WriteLine("Escrito {0} en PropiedadEjemplo", valor);
              }
     }
}
```

En este ejemplo se ve cómo se definen y redefinen propiedades abstractas. Al igual que **abstract** y override, también es posible usar cualquiera de los modificadores relativos a herencia y polimorfismo ya vistos: virtual, **new** y **sealed**.

Nótese que aunque en el ejemplo se ha optado por asociar un campo privado valor a la propiedad PropiedadEjemplo, en realidad nada obliga a que ello se haga y es posible definir propiedades que no tengan campos asociados. Es decir, una propiedad no se tiene porqué corresponder con un almacén de datos.

ANEXO I

La clase primigenia: System.Object

En .NET todos los tipos que se definan heredan implícitamente de la clase **System.Object** predefinida en la BCL, por lo que dispondrán de todos los miembros de ésta. Por esta razón se dice que **System.Object** es la raíz de la jerarquía de objetos de .NET.

A continuación, vamos a explicar cuáles son estos métodos comunes a todos los objetos:

- **public virtual bool Equals(object o):** Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro. Devuelve **true** si ambos objetos son iguales y **false** en caso contrario.

  La implementación que por defecto se ha dado a este método consiste en usar igualdad por referencia para los tipos por referencia e igualdad por valor para los tipos por valor. Es decir, si los objetos a comparar son de tipos por referencia sólo se devuelve **true** si ambos objetos apuntan a la misma referencia en memoria dinámica, y si los tipos a comparar son tipos por valor sólo se devuelve **true** si todos los bits de ambos objetos son iguales, aunque se almacenen en posiciones diferentes de memoria.

  Como se ve, el método ha sido definido como **virtual**, lo que permite que los programadores puedan redefinirlo para indicar cuándo ha de considerarse que son iguales dos objetos de tipos definidos por ellos. De hecho, muchos de los tipos incluidos en la BCL cuentan con redefiniciones de este tipo, como es el caso de **string**, quien, aun siendo un tipo por referencia, sus objetos se consideran iguales si apuntan a cadenas que sean iguales carácter a carácter (aunque referencien a distintas direcciones de memoria dinámica)

  El siguiente ejemplo muestra cómo hacer una redefinición de **Equals()** de manera que aunque los objetos Persona sean de tipos por referencia, se considere que dos Personas son iguales si tienen el mismo NIF:

```
public override bool Equals(object o)
{
        if (o==null)
                return this==null;
        else
                return (o is Persona) && (this.NIF == ((Persona) o).NIF);
}
```

Hay que tener en cuenta que es conveniente que toda redefinición del método **Equals()** que hagamos cumpla con una serie de propiedades que muchos de los métodos incluidos en las distintas clases de la BCL esperan que se cumplan. Estas propiedades son:

- **Reflexividad**: Todo objeto ha de ser igual a sí mismo. Es decir, x.Equals(x) siempre ha de devolver **true**.
- **Simetría**: Ha de dar igual el orden en que se haga la comparación. Es decir, x.Equals(y) ha de devolver lo mismo que y.Equals(x) .
- **Transitividad**: Si dos objetos son iguales y uno de ellos es igual a otro, entonces el primero también ha de ser igual a ese otro objeto. Es decir, si x.Equals(y) e y.Equals(z) entonces x.Equals(z) .
- **Consistencia**: Siempre que el método se aplique sobre los mismos objetos ha de devolver el mismo resultado.
- **Tratamiento de objetos nulos**: Si uno de los objetos comparados es nulo (**null**), sólo se ha de devolver true si el otro también lo es.

Hay que recalcar que el hecho de que redefinir Equals() no implica que el operador de igualdad **(==)** quede también redefinido.

- **public virtual int GetHashCode():** Devuelve un código de dispersión (hash) que representa de forma numérica al objeto sobre el que el método es aplicado. **GetHashCode()** suele usarse para trabajar con tablas de dispersión, y se cumple que si dos objetos son iguales sus códigos de dispersión serán iguales, mientras que si son distintos la probabilidad de que sean iguales es ínfima.

  En tanto que la búsqueda de objetos en tablas de dispersión no se realiza únicamente usando la igualdad de objetos (método Equals()) sino usando también la igualdad de códigos de dispersión, suele ser conveniente redefinir GetHashCode() siempre que se redefina Equals() De hecho, si no se hace el compilador informa de la situación con un mensaje de aviso.

- **public virtual string ToString():** Devuelve una representación en forma de cadena del objeto sobre el que se el método es aplicado, lo que es muy útil para depurar aplicaciones ya que permite mostrar con facilidad el estado de los objetos.

La implementación por defecto de este método simplemente devuelve una cadena de texto con el nombre de la clase a la que pertenece el objeto sobre el que es aplicado. Sin embargo, como lo habitual suele ser implementar ToString() en cada nueva clase que es defina, a continuación mostraremos un ejemplo de cómo redefinirlo en la clase Persona para que muestre los valores de todos los campos de los objetos Persona:

```
public override string ToString()
{
        string cadena = “”;
        cadena += “DNI = “ + this.DNI + ”\n”;
        cadena +=”Nombre = ” + this.Nombre + ”\n”;
        cadena +=”Edad = ” + this.Edad + ”\n”;
        return cadena;
}
```

Es de reseñar el hecho de que en realidad los que hace el operador de concatenación de cadenas (+) para concatenar una cadena con un objeto cualquiera es convertirlo primero en cadena llamando a su método **ToString()** y luego realizar la concatenación de ambas cadenas.

Del mismo modo, cuando a **Console.WriteLine()** y **Console.Write()** se les pasa como parámetro un objeto lo que hacen es mostrar por la salida estándar el resultado de convertirlo en cadena llamando a su método **ToString();** y si se les pasa como parámetros una cadena seguida de varios objetos lo muestran por la salida estándar esa cadena pero sustituyendo en ella toda subcadena de la forma {<número>} por el resultado de convertir en cadena el parámetro que ocupe la posición <número>+2 en la lista de valores de llamada al método.

- **protected object MemberWiseClone():** Devuelve una copia **shallow copy** del objeto sobre el que se aplica. Esta copia es una copia bit a bit del mismo, por lo que el objeto resultante de la copia mantendrá las mismas referencias a otros que tuviese el objeto copiado y toda modificación que se haga a estos objetos a través de la copia afectará al objeto copiado y viceversa.

  Si lo que interesa es disponer de una copia más normal, en la que por cada objeto referenciado se crease una copia del mismo a la que referenciase el objeto clonado, entonces el programador ha de escribir su propio método clonador pero puede servirse de MemberwiseClone() como base con la que copiar los campos que no sean de tipos referencia.

- **public System.Type GetType():** Devuelve un objeto de clase **System.Type** que representa al tipo de dato del objeto sobre el que el método es aplicado. A través de los métodos ofrecidos por este objeto se puede acceder a metadatos sobre el mismo como su nombre, su clase padre, sus miembros, etc. La explicación de cómo usar los miembros de este objeto para obtener dicha información queda fuera del alcance de este documento ya que es muy larga y puede ser fácilmente consultada en la documentación que acompaña al .NET SDK.

- **protected virtual void Finalize():** Contiene el código que se ejecutará siempre que vaya a ser destruido algún objeto del tipo del que sea miembro. La implementación dada por defecto a **Finalize()** consiste en no hacer nada.

  Aunque es un método virtual, en C# no se permite que el programador lo redefina explícitamente dado que hacerlo es peligroso por razones que se explicarán en el Tema 8: Métodos (otros lenguajes de .NET podrían permitirlo)

Aparte de los métodos ya comentados que todos los objetos heredan, la clase **System.Object** también incluye en su definición los siguientes métodos de tipo:

- **public static bool Equals(object objeto1, object objeto2)** → Versión estática del método **Equals()** ya visto. Indica si los objetos que se le pasan como parámetros son iguales, y para compararlos lo que hace es devolver el resultado de calcular **objeto1.Equals(objeto2)** comprobando antes si alguno de los objetos vale **null** (sólo se devolvería **true** sólo si el otro también lo es)

  Obviamente si se da una redefinición al **Equals()** no estático, sus efectos también se verán cuando se llame al estático.

- **public static bool ReferenceEquals(object objeto1, object objeto2)** → Indica si los dos objetos que se le pasan como parámetro se almacenan en la misma posición de memoria dinámica. A través de este método, aunque se hayan redefinido Equals() y el operador de igualdad **(==)** para un cierto tipo por referencia, se podrán seguir realizando comparaciones por referencia entre objetos de ese tipo en tanto que redefinir de Equals() no afecta a este método. Por ejemplo, dada la anterior redefinición de Equals() para objetos Persona:

```
Persona p = new Persona("José", 22, "83721654-W");
Persona q = new Persona("Antonio", 23, "83721654-W");
Console.WriteLine(p.Equals(q));
Console.WriteLine(Object.Equals(p, q));
Console.WriteLine(Object.ReferenceEquals(p, q));
Console.WriteLine(p == q);
```

La salida que por pantalla mostrará el código anterior es:

```
True
True
False
False
```

En los primeros casos se devuelve **true** porque según la redefinición de Equals() dos personas son iguales si tienen el mismo DNI, como pasa con los objetos p y q. Sin embargo, en los últimos casos se devuelve **false** porque aunque ambos objetos tienen el mismo DNI cada uno se almacena en la memoria dinámica en una posición distinta, que es lo que comparan ReferenceEquals() y el operador **==** (éste último sólo por defecto).